FEM on Dirichlet Poisson with Spatially Varying Anisotropic Coefficient

Lecture Course: C6.4 Finite Element Method for PDEs

Candidate Number: 1060612

Contents

1 Introduction

In this paper we discuss how to use the Finite Element Method to numerically solve the Dirichlet Poisson equation with a spatially varying anisotropic coefficient on an arbitrary domain $\Omega \subset \mathbb{R}^n$. This means we are solving

$$-\nabla \cdot (\mathbf{K}(\vec{x})\nabla u) = f(\vec{x}), \qquad \vec{x} \in \Omega \tag{1}$$

$$u = 0, \qquad \vec{x} \in \partial\Omega \tag{2}$$

where $\mathbf{K}(\vec{x}) : \Omega \to \mathbb{R}^{d\times d}$ and $\mathbf{K}(\vec{x})$ is a symmetric positive definite matrix. Additionally, for this method to work we must have $||f||_{L^2(\Omega)} < \infty$.

1.1 Step by Step Guide to FEM

This is a step by step[1] guide for solving linear PDEs by FEM.

STEP 1: Multiply the PDE by test function ζ and integrate over domain Ω.

STEP 2: Use integration by parts to reduce the order of the highest order term, thus getting the variational formula.

[1]Inspired by [2]

STEP 3: Decide the type of element to use (e.g linear Lagrange, quadratic Lagrange, quadratic Hermite ...) and generate the mesh over the domain Ω.

STEP 4: Derive the test functions ϕ_i, Then we have $u(\vec{x}) \approx u_h(\vec{x}) = \sum_i u_i \phi_i(\vec{x})$ which is known as the Galerkin Approximation.

STEP 5: Set $\zeta = \phi_j$, where $j = 1, 2, ..., N$ and N is the number test functions. This leads to N equations for the variational formula.

STEP 6: Substitute $u_h(\vec{x})$ into each variational formula to get a linear system of equations for u_i. (The integrals can be calculated numerically or analytically.)

STEP 7: Solve the linear system to get the values of u_i.

For this project, we only consider the linear Lagrange elements. This means our solution will be piecewise linear.

2 Derivation of Variational Formula

Multiply PDE (1) by a weight function ζ and integrate over the domain Ω to obtain

$$\int_\Omega \zeta(\vec{x}) f(\vec{x}) d\vec{x} = -\int_\Omega \zeta(\vec{x}) \nabla \cdot (\mathbf{K}(\vec{x}) \nabla u) d\vec{x}. \tag{3}$$

Using integration by parts we get

$$\int_\Omega \zeta(\vec{x}) f(\vec{x}) d\vec{x} \;=\; \int_\Omega (\mathbf{K}(\vec{x}) \nabla u) \cdot \nabla \zeta(\vec{x}) d\vec{x} - \int_{\partial\Omega} \zeta(\vec{x}) (\mathbf{K}(\vec{x}) \nabla u) \cdot \hat{n} d\vec{x} \tag{4}$$

as shown in (71). Now we set $\zeta(\vec{x}) = 0$ when $\vec{x} \in \partial\Omega$. This leads to

$$\int_\Omega \zeta(\vec{x}) f(\vec{x}) d\vec{x} \;=\; \int_\Omega (\mathbf{K}(\vec{x}) \nabla u) \cdot \nabla \zeta(\vec{x}) d\vec{x}. \tag{5}$$

Therefore, we have taken $u, \zeta \in H_0^1(\Omega) = \{\zeta \in H^1(\Omega) : \zeta = 0 \text{ on } \partial\Omega\}$. Now we find $u \in H_0^1(\Omega)$ such that equation (5) is satisfied for all $\zeta \in H_0^1(\Omega)$. Additionally,

2.1 Galerkin Approximation

Let N_I be the number of nodes inside Ω. Then we aim to get a system of N_I linear equations by substituting $u_h(\vec{x}) = \sum_{i=1}^{N_I} u_i \phi_i(\vec{x})$ for $u(\vec{x})$, and substituting $\zeta = \phi_j(\vec{x})$ where $j \in 1, 2, ..., N_I$. This leads us to

$$\sum_{i=1}^{N_I} u_i \int_\Omega (\mathbf{K}(\vec{x}) \nabla \phi_i(\vec{x})) \cdot \nabla \phi_j(\vec{x}) d\vec{x} = \int_\Omega \phi_j(\vec{x}) f(\vec{x}) d\vec{x} \tag{6}$$

which we can turn into a matrix problem

$$\mathbf{A}\vec{U} = \vec{F} \tag{7}$$

$$\mathbf{A}_{j,i} = \int_{\Omega} (\mathbf{K}(\vec{x})\nabla\phi_i(\vec{x})) \cdot \nabla\phi_j(\vec{x}) d\vec{x} \tag{8}$$

$$\vec{F}_j = \int_{\Omega} \phi_j(\vec{x}) f(\vec{x}) d\vec{x} \tag{9}$$

with $\mathbf{A} \in \mathbb{R}^{N_I \times N_I}$ and $\vec{U}, \vec{F} \in \mathbb{R}^{N_I}$.

A detailed description of methods for the solution of linear systems is beyond the scope of this paper. We will use $\vec{U}$ = numpy.linalg.solve($\mathbf{A}$,$\vec{F}$).

We have discretised equation (5) because computers work with a finite amount of computation. Thus we change the closed set $H_0^1(\Omega)$ into $H_0^1(\Omega)_h$ which contains basis functions ϕ_i that span most of $H_0^1(\Omega)$. We have $\mathbf{A}$ is invertible because Lax-Milgram is satisfied thus the Galerkin approximation is well-posed for any closed subspace $H_0^1(\Omega)_h \subset H_0^1(\Omega)$, which appears as corollary 7.1.1 of [1]. However, this will induce error in our solution since we are now solving a simplified version of the original problem. Additionally, ϕ_i is in $H_0^1(\Omega)_h$ which implies u_h is in $H_0^1(\Omega)_h$. Therefore, u_h fits our Dirichlet boundary conditions.

3 Existence and Uniqueness of Solution

We will use the Lax-Milgram Theorem [7] on the Dirichlet Poisson problem with spatially varying anisotropic coefficient (1) to show that the variational problem has a unique stable solution. For this section we will use the notation $a(\cdot,\cdot) : H_0^1 \times H_0^1 \to \mathbb{R}$, $F(\cdot) : H_0^1 \to \mathbb{R}$:

$$a(u,\zeta) = \int_{\Omega} (\mathbf{K}(\vec{x})\nabla u(\vec{x})) \cdot \nabla\zeta(\vec{x}) d\vec{x} = \int_{\Omega} (\nabla u)^T \mathbf{K} \nabla\zeta d\vec{x} \tag{10}$$

$$F(\zeta) = \int_{\Omega} \zeta(\vec{x}) f(\vec{x}) d\vec{x} \tag{11}$$

For (10) we have used the fact $\mathbf{K}$ is symmetric ($\mathbf{K} = \mathbf{K}^T$). First we show $a(\cdot,\cdot)$ is an inner product space [8] so we can use the Cauchy–Schwarz inequality [1].

Symmetric: We show that $a(u,\zeta) = a(\zeta,u)$ for all $u, \zeta \in H_0^1$.

$$a(u,\zeta) = \int_{\Omega} (\nabla u)^T \mathbf{K} \nabla\zeta d\vec{x} = \int_{\Omega} ((\mathbf{K}\nabla u)^T \nabla\zeta)^T d\vec{x} = \int_{\Omega} (\nabla\zeta)^T \mathbf{K} \nabla u d\vec{x} = a(\zeta,u). \tag{12}$$

This is only valid if $\mathbf{K}$ is symmetric.

Bilinear: As the PDE is linear this is trivial to prove.

We now define

$$\Lambda_{max} = \sup\{\vec{y}^T\mathbf{K}(\vec{x})\vec{y} : \vec{x} \in \Omega, \vec{y} \in \mathbb{R}^d, |\vec{y}| = 1\} = \sup\{\lambda_{max}(\mathbf{K}(\vec{x})) : \vec{x} \in \Omega\}, \quad (13)$$

$$\Lambda_{min} = \inf\{\vec{y}^T\mathbf{K}(\vec{x})\vec{y} : \vec{x} \in \Omega, \vec{y} \in \mathbb{R}^d, |\vec{y}| = 1\} = \inf\{\lambda_{min}(\mathbf{K}(\vec{x})) : \vec{x} \in \Omega\}, \quad (14)$$

where $\lambda_{max}, \lambda_{min}$ denote the maximum and minimum eigenvalue respectively of a matrix. This means Λ_{max} is the maximum value an eigenvalue of $\mathbf{K}(\vec{x})$ can take on the domain Ω. We denote $|\cdot| = ||\cdot||_2$.

Positive Definite: We show that $a(u,u) \geq 0$, and that equality holds when $u = 0$. Now

$$a(u,u) = \int_\Omega (\nabla u)^T\mathbf{K}(\vec{x})\nabla u d\vec{x} \geq \Lambda_{min}\int_\Omega |\nabla u|^2 d\vec{x} \geq 0. \quad (15)$$

As $\mathbf{K}$ is positive definite on Ω we know $\Lambda_{min} > 0$. As $|\nabla u|^2 \geq 0$ we must have equality only holds when $\nabla u = 0$ this implies $u =$ constant. However, $u = 0$ on boundary $(\partial\Omega)$ thus $u = 0$. This means $a(\cdot,\cdot)$ forms an inner product with norm $||u||_a = \sqrt{a(u,u)}$.

Continuity of $a(\cdot,\cdot)$: Show $|a(u,\zeta)| \leq C||u||_{H^1(\Omega)}||\zeta||_{H^1(\Omega)}, \forall u, \zeta \in H_0^1(\Omega)$ and $C \in [0,\infty)$. First we use the Cauchy-Schwarz inequality to get

$$|a(u,\zeta)| \leq ||u||_a||\zeta||_a = \left(\int_\Omega (\nabla u)^T\mathbf{K}(\nabla u)d\vec{x}\right)^{1/2}\left(\int_\Omega (\nabla\zeta)^T\mathbf{K}(\nabla\zeta)d\vec{x}\right)^{1/2}, \quad (16)$$

now using (78) to bound above we get

$$|a(u,\zeta)| \leq \Lambda_{max}\left(\int_\Omega |\nabla u|^2 d\vec{x}\right)^{1/2}\left(\int_\Omega |\nabla\zeta|^2 d\vec{x}\right)^{1/2} \quad (17)$$

and using the fact $\int_\Omega u^2 d\vec{x} \geq 0$ we get

$$|a(u,\zeta)| \leq \Lambda_{max}\left(\int_\Omega u^2 + |\nabla u|^2 d\vec{x}\right)^{1/2}\left(\int_\Omega \zeta^2 + |\nabla\zeta|^2 d\vec{x}\right)^{1/2} \quad (18)$$

$$|a(u,\zeta)| = \Lambda_{max}||u||_{H^1(\Omega)}||\zeta||_{H^1(\Omega)}. \quad (19)$$

Coercivity of $a(\cdot,\cdot)$: Show $a(u,u) \geq \alpha||u||_{H^1(\Omega)}$ with $\alpha \in (0,\infty)$.

$$\begin{aligned} a(u,u) = \int_\Omega (\nabla u)^T\mathbf{K}(\vec{x})(\nabla u)d\vec{x} &\geq \Lambda_{min}\int_\Omega |\nabla u|^2 d\vec{x} \\ &= \frac{\Lambda_{min}}{2}\left(\int_\Omega |\nabla u|^2 d\vec{x} + \int_\Omega |\nabla u|^2 d\vec{x}\right) \\ &\geq \frac{\Lambda_{min}}{2}\left(\int_\Omega |\nabla u|^2 + \frac{1}{d_\Omega}u^2 d\vec{x}\right) \geq \frac{\Lambda_{min}}{2}\min(1,\frac{1}{d_\Omega})||u||_{H^1(\Omega)}. \end{aligned} \quad (20)$$

In equation (20) we have used the Poincaré Inequality [7], which states

$$\int_\Omega u^2 d\vec{x} \leq d_\Omega \int_\Omega |\nabla u|^2 d\vec{x} \tag{21}$$

where d_Ω is constant and $u \in H_0^1(\Omega)$.

In Appendix D we explain how we bounded equations (15), (16) and (20) by Λ_{min} and Λ_{max}.

Continuity of $F(\cdot)$: Show $|F(\zeta)| \leq C||\zeta||_{H^1(\Omega)}, \forall \zeta \in H_0^1(\Omega)$ and $C \in [0, \infty)$. We note $F(\zeta)$ is an inner product on $L^2(\Omega)$. Therefore, we can use the Cauchy-Schwarz inequality to get

$$|F(\zeta)| = |(f, \zeta)_{L^2}| \leq ||f||_{L^2}||\zeta||_{L^2} \leq ||f||_{L^2} \int_\Omega u^2 + |\nabla u|^2 d\vec{x} = ||f||_{L^2}||\zeta||_{H^1(\Omega)}. \tag{22}$$

We have that $||f||_{L^2} < \infty$, this means $f \in L^2(\Omega)$. Thus by the Lax-Milgram theorem the variational formula is well-posed, and has a unique solution for u. Additionally, by Corollary 7.1.1. from [1], the Galerkin Approximation is well-posed.

4 Error Checking

The best way to test the FEM code works is to calculate $||u - u_h||_{H^1}$ which is the Sobolev norm of the error. If we can show that it is small and decreases as the number of nodes in the mesh increases it is a good indication that the code is correct. The drawback with this method is that we need to know the solution u. We have the norm of the error is given by

$$||u - u_h||_{H^1(\Omega)} = ||u - u_h||_{W_2^1(\Omega)} = \left(\sum_{|\alpha|=0}^{1} ||D^\alpha (u - u_h)||^2_{L^2(\Omega)} \right)^{1/2}. \tag{23}$$

By using the definition of the Lebesgue 2-norm, $||\cdot||_{L^2(\Omega)}$, we get

$$||u - u_h||_{H^1(\Omega)} = \left(\int_\Omega \sum_{|\alpha|=0}^{1} |D^\alpha (u(\vec{x}) - u_h(\vec{x}))|^2 d\vec{x} \right)^{1/2}. \tag{24}$$

In our code, we calculate the error over each element and sum them. This leads to an error over the entire domain Ω.

4.1 Convergence Results

For Lagrange elements of order 1, we have the convergence result from [1]

$$||u - u_h||_{H^1(\Omega)} \leq Ch|u|_{H^2(\Omega)} = Ch\left(\int_\Omega \sum_{|\alpha|=2} |D^\alpha u|^2 d\vec{x}\right)^{1/2} \tag{25}$$

for $C < \infty$. We will set $C_1 = C|u|_{H^2(\Omega)} < \infty$, to get

$$||u - u_h||_{H^1(\Omega)} \leq C_1 h. \tag{26}$$

Equation (26) means when the mesh size h is halved the error in the H^1 norm is halved. For extra clarity, we have the following equations

$$|u|_{H^2(a,b)} = \left(\int_a^b \left|\frac{\partial^2 u}{\partial x^2}\right|^2 dx\right)^{1/2}, \tag{27}$$

$$|u|_{H^2(\Omega)} = \left(\int_\Omega \left|\frac{\partial^2 u}{\partial x^2}\right|^2 + \left|\frac{\partial^2 u}{\partial x \partial y}\right|^2 + \left|\frac{\partial^2 u}{\partial y^2}\right|^2 dxdy\right)^{1/2}, \tag{28}$$

where (27) is 1D and (28) is 2D. For (26) we require the sequence of meshes to be shape regular, this is satisfied by our mesh generation algorithm. There is More about Mesh Generation in Appendix C.

4.2 Mesh Size h

We will use the definitions from [1], first we have the diameter h_E of a finite element E

$$h_E = \sup\{||x - y|| : x, y \in E\} \tag{29}$$

where E is the finite element we are currently looking at. As we are dealing with triangular elements, this simplifies to the length of the longest edge. For a mesh $\mathbf{M}$ we take the mesh size h to be

$$h = \sup_{E \in \mathbf{M}} h_E. \tag{30}$$

For 2D this means we take h to equal the length of the triangle with the longest edge. For 1D when nodes are equally spaced the mesh size h is equal to the distance between each node.

4.3 Error in 1D

The formula for 1D error in the space $H^1(a,b)$ is

$$||u-u_h||_{H^1(a,b)} = \left(\int_a^b |u(x)-u_h(x))|^2 + \left| \frac{\partial u}{\partial x}(x) - \frac{\partial u_h}{\partial x}(x) \right|^2 dx \right)^{1/2}. \tag{31}$$

4.4 Error in 2D

The formula for 2D error in the space $H^1(\Omega)$ is

$$||u-u_h||_{H^1(\Omega)} = \left(\int_\Omega |u-u_h|^2 + \left| \frac{\partial u}{\partial x} - \frac{\partial u_h}{\partial x} \right|^2 + \left| \frac{\partial u}{\partial y} - \frac{\partial u_h}{\partial y} \right|^2 dxdy \right)^{1/2}. \tag{32}$$

We will discuss how our code calculates the error. First we separate the integral into each element of the domain Ω. Then we have

$$u_h(x,y) = u_1^{(L)}\phi_1^{(L)}(x,y) + u_2^{(L)}\phi_2^{(L)}(x,y) + u_3^{(L)}\phi_3^{(L)}(x,y) \tag{33}$$

with each $\phi_i^{(L)}$ representing a basis function located at the vertices of the triangle. We use the same method we used in section (8.1) to calculate $\phi_i^{(L)}$. Now using the notation in (55) for u_h we get

$$\begin{aligned} u_h(x,y) &= \vec{\alpha}\cdot\vec{v}^{(L)} + \vec{\beta}\cdot\vec{v}^{(L)}x + \vec{\xi}\cdot\vec{v}^{(L)}y, & (34)\\ \frac{\partial u_h}{\partial x} &= \vec{\beta}\cdot\vec{v}^{(L)}, & (35)\\ \frac{\partial u_h}{\partial y} &= \vec{\xi}\cdot\vec{v}^{(L)} & (36) \end{aligned}$$

where $\vec{v}^{(L)} = \begin{bmatrix} u_1^{(L)} & u_2^{(L)} & u_3^{(L)} \end{bmatrix}^T$. To calculate the error we substitute equations above (34) , (35) and (36) into (32). The 1D case is similar to the steps above.

5 Laplace Equation 1D

Here we solve the 1D version of (1) with boundary conditions $u(a) = u(b) = 0$.

5.1 Global Formulation for 1D Linear Lagrange Element

With Global formulation we calculate the linear system of equations provided by the Galerkin Approximation (8) and (9) by calculating every integral over the domain $\Omega = [a,b]$. We denote the position of the nodes by x_j such that $a = x_0 \le x_1 \le x_2 \le$

$\cdots \leq x_{N-1} \leq x_N = b$. And $\dot{\phi}_i = \frac{d\phi_i}{dx}$. For simplicity we will assume the nodes are evenly spaced, thus $x_j = a + jw$ where $w = \frac{b-a}{N}$. Additionally, we show that this formulation can be considered the same as the finite difference method.

In 1D the linear basis functions and their derivatives for $i = 1, \ldots, N-1$ are

$$\phi_i = \begin{cases} \frac{x_{i+1}-x}{w}, & x_i \leq x \leq x_{i+1} \\ \frac{x-x_{i-1}}{w}, & x_{i-1} \leq x \leq x_i \\ 0, & \text{otherwise} \end{cases} \tag{37}$$

$$\dot{\phi}_i = \begin{cases} -\frac{1}{w}, & x_i < x < x_{i+1} \\ \frac{1}{w}, & x_{i-1} < x < x_i \\ 0, & \text{otherwise} \end{cases} \tag{38}$$

We have $u_h \in H_0^1(a,b)_h$ this implies $u_0 = u_N = 0$ and $H_0^1(a,b)_h = span\{\phi_i, 1 \leq i \leq N-1\}$. This means $\phi_0, \dot{\phi}_0, \phi_N$ and $\dot{\phi}_N$ are not needed, so we go from $N+1$ nodes to $N-1$ equations to solve. Whenever we solve the Dirichlet Poisson problem with linear Lagrange elements, we need to solve a system of N_I (Number of nodes in domain) linear equations.

We now calculate $\mathbf{A}$, it is useful to look at

$$\dot{\phi}_j(x)\dot{\phi}_i(x) = \frac{1}{w^2} \begin{cases} -1, & x_j < x < x_{j+1} \text{ and } i = j+1 \\ -1, & x_i < x < x_{i+1} \text{ and } j = i+1 \\ 1, & x_{i-1} < x < x_{i+1} \text{ and } i = j \\ 0, & \text{otherwise} \end{cases} \tag{39}$$

which we substitute into formula (8) to obtain

$$\mathbf{A}_{j,i} = \int_a^b K(x)\dot{\phi}_i(x)\dot{\phi}_j(x)dx = \frac{1}{w^2} \begin{cases} -\int_{x_j}^{x_{j+1}} K(x)dx, & i = j+1 \\ -\int_{x_i}^{x_{i+1}} K(x)dx, & j = i+1 \\ \int_{x_{i-1}}^{x_{i+1}} K(x)dx, & i = j \\ 0, & \text{otherwise} \end{cases}. \tag{40}$$

Now for $\vec{F}$

$$\vec{F}_j = \int_a^b \phi_j(x)f(x)dx = \int_{x_{j-1}}^{x_{j+1}} \phi_j(x)f(x)dx. \tag{41}$$

To solve we use numerical integration rules, some of which are described here in Appendix B.

We will now take $K = 1$, $\vec{F}_j$ (41) will stay the same and we will numerically calculate it by making the assumption $f(x) = f(x_j)$, for $x_{j-1} < x < x_{j+1}$, taking f

out and integrating over ϕ_j gives us $\vec{F}_j = wf(x_j)$ (since $\int_{x_{j-1}}^{x_{j+1}} \phi_i(x)dx = w$). This is a good approximation if there are enough nodes in the mesh or the mesh size (30) is small.

Additionally, we have $\mathbf{A}$ is

$$\mathbf{A}_{j,i} = \int_a^b \dot{\phi}_j(x)\dot{\phi}_i(x)dx = \frac{1}{w}\begin{cases} -1, & |i-j| = 1 \\ 2, & i = j \\ 0, & \text{otherwise} \end{cases} \tag{42}$$

As can be seen, we have derived the Finite Difference Method for solving the Dirichlet Poisson equation. However, this statement is only valid by assuming $f(x) = f(x_j)$, for $x_{j-1} < x < x_{j+1}$.

In our code, we use the local Lagrange element formulation for the 1D problem. The 2D local formulation is located in section 8.1 it is trivial to convert this to a 1D local formulation.

5.2 Example 1: Dirichlet Poisson 1D

We will use our FEM code to solve the PDE

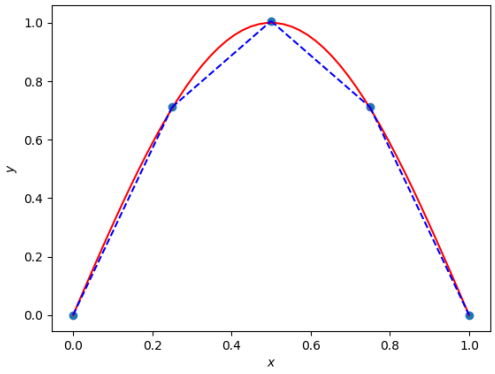
$$-\frac{\partial^2 u}{\partial x^2} = \pi^2 \sin(\pi x), \qquad x \in (0,1) \tag{43}$$

$$u = 0, \qquad x = 0 \text{ or } x = 1 \tag{44}$$

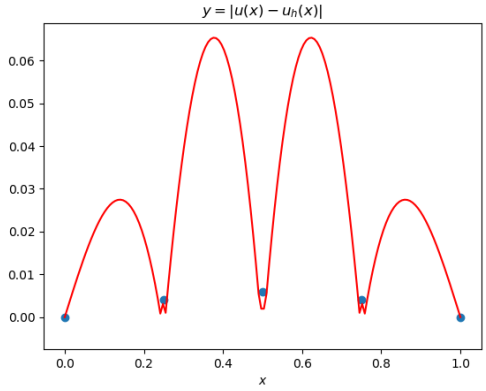
which has exact solution $u(x) = \sin(\pi x)$. Below is Python code with its corresponding output.

```python
#Example_1.py
from PoissonSolver import Poisson1D
import numpy as np; import sympy as sp;
sol = lambda x : np.sin(np.pi*x)
sol_sp = lambda x : sp.sin(np.pi*x)
f = lambda x : np.pi**2*np.sin(np.pi*x)
Example_1 = Poisson1D(a=0, b=1, f=f, N_n=10)  # N_n = 5 #N_n = 10
Example_1.plot1D(sol)
Example_1.plotError(sol)
print(Example_1.Norm(sol_sp))
```

The blue dots in (Fig:1) and (Fig:2) are the positions of nodes. It is the position where $\phi_j = 1$.

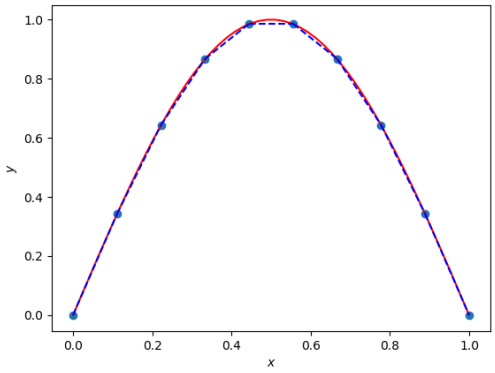


(a) Blue: FEM Solution, Red: True Solution, plot1D

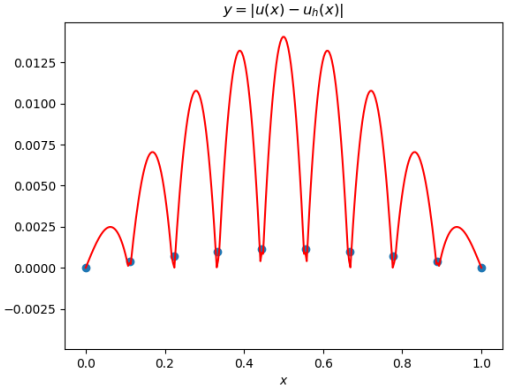


(b) $y = |u(x) - u_h(x)|$, plotError

Figure 1: Output from Example_1.py with $N_n = 5$



(a) Blue: FEM Solution, Red: True Solution, plot1D



(b) $y = |u(x) - u_h(x)|$, plotError

Figure 2: Output from Example_1.py with $N_n = 10$

5.3 Example 2: Dirichlet Poisson 1D with $K = 1/x$

We will use our FEM code to solve a spatially varying anisotropic coefficient problem

$$-\frac{\partial}{\partial x}\left(\frac{1}{x}\frac{\partial u}{\partial x}\right) = x, \qquad x \in (0,2) \tag{45}$$

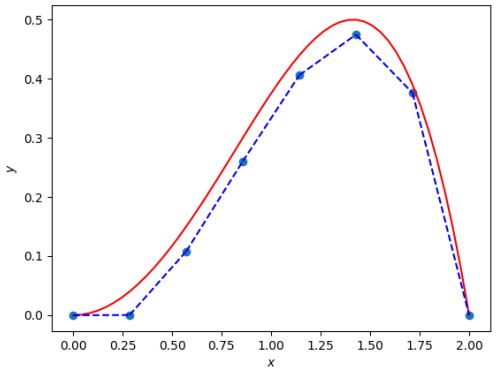
$$u = 0, \qquad x = 0 \text{ or } x = 2 \tag{46}$$

which has exact solution $u(x) = \frac{x^2}{2} - \frac{x^4}{8}$.

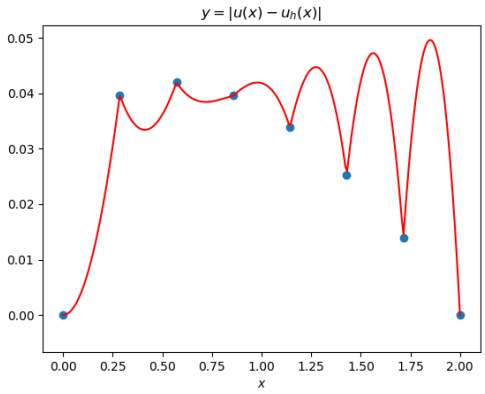
We note that K is not defined at $x = 0$, in the analytic solution we take the limit as $x \to 0$. In the Python program we will change the left boundary to become

$a = 0.0001$ to approximate the left boundary being at zero.

```
# Example_2.py
from PoissonSolver import Poisson1D; import numpy as np
f = lambda x : x
K = lambda x : np.array([[1/x]])
sol = lambda x : (x**2)/2 - (x**4)/8
Example_2 = Poisson1D(a=0.0001, b=2, f=f, N_n=16, K=K) #N_n=8 N_n=16
Example_2.plot1D(sol) #Note sol is not needed for plot1D
Example_2.plotError(sol)
print(Example_2.Norm(sol))
```



(a) Blue: FEM Solution, Red: True Solution, plot1D



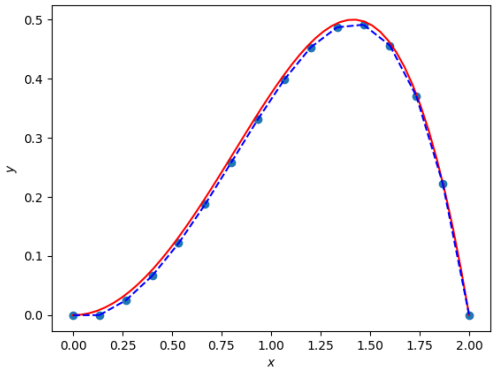
(b) $y = |u(x) - u_h(x)|$, plotError

Figure 3: Output from Example_2.py with $N_n = 8$

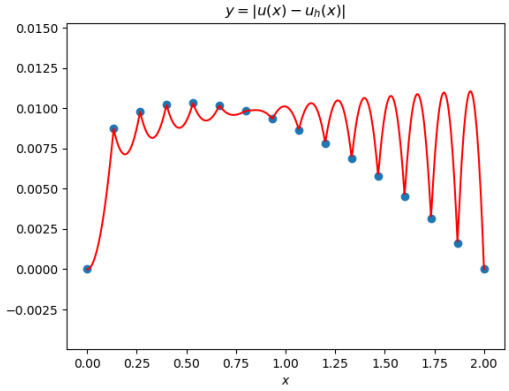
5.4 $||\mathbf{u} - \mathbf{u_h}||_{\mathbf{H^1(a,b)}}$ for Examples 1 and 2

Here we discuss the approximation error in $H^1(a, b)$ ($||u - u_h||_{H^1(a,b)}$) for examples 1 and 2 with different number of nodes N_n.

As can be seen in (Table 1) in both examples the error $||u - u_h||_{H^1(a,b)}$ is halved when the number of nodes N_n is doubled. We have Mesh Size $h = \frac{b-a}{N_n - 1}$, which in this case equals the width w, when N_n is doubled this means Mesh Size h is approximately halved. This is a convergence result (26). Thus giving us a strong indication that our Python code for solving the 1D version of PDE (1) works. Here we calculate the errors by "Example_1.Norm(sol)", we have also truncated the result to fit in the table.



(a) Blue: FEM Solution, Red: True Solution, plot1D



(b) $y = |u(x) - u_h(x)|$, plotError

Figure 4: Output from Example_2.py with $N_n = 16$

6 Finding Solutions to Test

6.1 Solution to Polygon Domains

Here we find solutions to the Poisson PDE (1), by finding a solution u to fit the boundary condition. Then we calculate $f(\vec{x})$ and then we test the FEM code by calculating $||u - u_h||_{H^1}$. First we find the equation of each edge on the domain, for edge i which has vertices (x_1^i, y_1^i) and (x_2^i, y_2^i) we have

$$y = \frac{(y_1^{(i)} - y_2^{(i)})x + x_1^{(i)}y_2^{(i)} - y_1^{(i)}x_2^{(i)}}{x_1^{(i)} - x_2^{(i)}}. \tag{47}$$

This leads to a solution of

$$u(x,y) = \prod_{i=1}^{N}((x_2^{(i)} - x_1^{(i)})y + (y_1^{(i)} - y_2^{(i)})x + x_1^{(i)}y_2^{(i)} - y_1^{(i)}x_2^{(i)}) \tag{48}$$

where N is the number of edges on the domain. Additionally, we can multiply solution (48) by $g(x,y)$ and the solution u will still satisfy the boundary conditions. However, $g(x,y) < \infty$ when (x,y) on $\partial\Omega$.

6.2 Solutions to Non-Polygon Domains

We now find a solution for a domain with curves. One method would be to approximate the domain with a polygon, for example when the domain curves we can ap-

Vertices N_n	Edges	Example 1	Example 2
5	4	0.5515280	0.5137623
10	9	0.2470058	0.2149887
20	19	0.1171741	0.0992568
40	39	0.0571032	0.0478350
80	79	0.0281923	0.0235030
160	159	0.0140077	0.0116527
320	319	0.0069819	0.0058025
640	639	0.0034855	0.0028955
1280	1279	0.0017414	0.0014464

Table 1: $||u - u_h||_{H^1(a,b)}$ for increasing vertices w

proximate this curve with a sequence of straight lines then use (48) to find u.However, this means the solution u becomes complicated.

Another method is to use a coordinate change of basis. Then we use the same idea that was used in the previous section. For example, in polar coordinates we can have the function $u(r, \theta) = (R(\theta) - r)$ which will be zero on $r = R(\theta)$. Instead of calculating the grad operator for polar coordinates it is easier to transform the solution back into Cartesian coordinates, when doing this we multiply u by $(R(\theta) + r)$ to get $u(r, \theta) = (R(\theta)^2 - r^2)$. Therefore, for Example 3 (8.2) we have $R(\theta) = 1$ thus we get $u(x, y) = 1 - r^2 = 1 - x^2 - y^2$.

7 Integrating Over Triangular Domain

Here we calculate the integral

$$\int_E g(x, y)dxdy \tag{49}$$

where E is the triangle domain. With the triangle vertices at $\vec{a}, \vec{b}, \vec{c}$.

7.1 Method 1: Transformation of Coordinates

We will use this change of coordinates

$$\vec{x} = \vec{a} + (\vec{b} - \vec{a})\hat{x} + (\vec{c} - \vec{a})\hat{y}, \tag{50}$$

$$x = a_1 + (b_1 - a_1)\hat{x} + (c_1 - a_1)\hat{y}, \tag{51}$$

$$y = a_2 + (b_2 - a_2)\hat{x} + (c_2 - a_2)\hat{y}. \tag{52}$$

This mean the integral (49) becomes

$$|\mathbf{J}| \int_0^1 \int_0^{1-\hat{x}} g(x(\hat{x}, \hat{y}), y(\hat{x}, \hat{y}))d\hat{y}d\hat{x} \tag{53}$$

where $|\mathbf{J}| = |(b_1 - a_1)(c_2 - a_2) - (c_1 - a_1)(b_2 - a_2)| = 2\times$Area of triangle. It is trivial to see this method could be extended to a tetrahedron.

The advantage of this method is you can get the exact solution (Therefore very accurate). But doing this method numerically requires a lot more computation than taking the approximation that g is constant on the domain E. This gives a good approximation if the size h of the Mesh (30) is small.

7.2 Method 2: Recursive Calculation

The idea of this method is to split the triangle E into 3 triangles via the midpoint, then split those 3 triangles into 3 triangles (Thus we have 9 triangles that all have the same area). The idea of this is to get points $(x_n, y_1), (x_2, y_2), ..., (x_n, y_n)$ on the triangle that are fairly evenly spaced. Then calculate $g(x, y)$ of the points, then the mean, then multiply by area of the triangle E.

For an N^{th} order version of this algorithm we will get $n = 3^{N-1}$ points. Thus we get

$$\int_E g(x, y)dxdy = \frac{|\mathbf{J}|}{2 \cdot 3^{N-1}} \sum_{i=1}^{3^{N-1}} g(x_i, y_i). \tag{54}$$

In our Python implementation we take g is constant on E. This means we are using the first order version of this method.

8 Poisson Equation 2D

8.1 Local Formulation for 2D Linear Lagrange Element

With local formulation we look at each finite element in the mesh. In this paper we will discuss a triangular mesh which is obtained by splitting the domain Ω into finite

elements. We denote each triangle (finite element) in the mesh by E. For each E in the mesh we use the variational formula (5) to obtain a system of linear equations, this gives $\mathbf{A}^{(L)}$ and $\vec{\mathbf{F}}_j^{(L)}$. Then we use a local to global map to add into $\mathbf{A}$ and $\vec{\mathbf{F}}$.

For the linear Lagrange finite element in 2D we have three nodes, for each element to consider, this implies $\mathbf{A}^{(L)} \in \mathbb{R}^{3\times 3}$. First we need to find the equations of the test functions $\phi_j^{(L)}$ which are

$$\phi_1^{(L)} = \alpha_1 + \beta_1 x + \xi_2 y, \qquad \phi_2^{(L)} = \alpha_2 + \beta_2 x + \xi_2 y, \qquad \phi_3^{(L)} = \alpha_3 + \beta_3 x + \xi_3 y, \tag{55}$$

$$\phi_1^{(L)}(x_1, y_1) = 1, \qquad \phi_2^{(L)}(x_1, y_1) = 0, \qquad \phi_3^{(L)}(x_1, y_1) = 0, \tag{56}$$

$$\phi_1^{(L)}(x_2, y_2) = 0, \qquad \phi_2^{(L)}(x_2, y_2) = 1, \qquad \phi_3^{(L)}(x_2, y_2) = 0, \tag{57}$$

$$\phi_1^{(L)}(x_3, y_3) = 0, \qquad \phi_2^{(L)}(x_3, y_3) = 0, \qquad \phi_3^{(L)}(x_3, y_3) = 1. \tag{58}$$

We notice these $\phi_j^{(L)}$ (55) are only valid on the finite element domain. However, we only use these in the current finite element. Now we put equations (55) to (58) into a matrix system and calculate the inverse which gives

$$\begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \\ \beta_1 & \beta_2 & \beta_3 \\ \xi_1 & \xi_2 & \xi_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{59}$$

$$\begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \\ \beta_1 & \beta_2 & \beta_3 \\ \xi_1 & \xi_2 & \xi_3 \end{bmatrix} = \frac{1}{B} \begin{bmatrix} x_2 y_3 - x_3 y_2 & x_3 y_1 - x_1 y_3 & x_1 y_2 - x_2 y_1 \\ y_2 - y_3 & y_3 - y_1 & y_1 - y_2 \\ x_3 - x_2 & x_1 - x_3 & x_2 - x_1 \end{bmatrix} \tag{60}$$

where $B = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$. This leads to $\nabla \phi_i^{(L)} = \begin{bmatrix} \beta_i \\ \xi_i \end{bmatrix}$. Substituting into equation (8) and integrating of finite element gives E

$$\mathbf{A}_{j,i}^{(L)} = \int_\Omega \nabla \phi_i^{(L),T} \mathbf{K}(\vec{x}) \nabla \phi_j^{(L)} d\vec{x} = \begin{bmatrix} \beta_i & \xi_i \end{bmatrix} \int_E \mathbf{K}(\vec{x}) d\vec{x} \begin{bmatrix} \beta_j \\ \xi_j \end{bmatrix}. \tag{61}$$

We have used the fact that $\mathbf{K}(\vec{x}) = \mathbf{K}^T(\vec{x})$. We can use this fact again to simplify to

$$\mathbf{A}^{(L)} = \begin{bmatrix} \beta_1 & \xi_1 \\ \beta_2 & \xi_2 \\ \beta_3 & \xi_3 \end{bmatrix} \int_E \mathbf{K}(\vec{x}) d\vec{x} \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 \\ \xi_1 & \xi_2 & \xi_3 \end{bmatrix}. \tag{62}$$

The integral in (62) means calculating the integral over each component in the matrix.

When we have $\mathbf{K}(\vec{x}) = I$, the integral becomes the area of the triangle, thus we get

$$\mathbf{A}^{(L)} = \frac{|B|}{2} \begin{bmatrix} \beta_1 & \xi_1 \\ \beta_2 & \xi_2 \\ \beta_3 & \xi_3 \end{bmatrix} \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 \\ \xi_1 & \xi_2 & \xi_3 \end{bmatrix}. \tag{63}$$

We note there are other methods for finding $\mathbf{A}^{(L)}$. We could use a change of coordinates for the integral (8) as mentioned in Section 7.1.

The formula for $\mathbf{A}^{(L)}$ can be easily calculated independently for each finite element. Therefore it is trivial to make use of parallel computing. However, the addition of $\mathbf{A}^{(L)}$ into $\mathbf{A}$ needs to be done sequentially otherwise race conditions could occur. A description and example of race conditions can be found in [4].

We will now calculate $\vec{F}^{(L)}$. From (9) we have

$$\vec{F}_j^{(L)} = \int_E \phi_j^{(L)}(x,y) f(x,y) dxdy. \tag{64}$$

In 2D $j \in \{1,2,3\}$. If $f(x,y)$ is an arbitrary function, this formula can not be simplified. Therefore, it must be calculated numerically see Section 7. However, if $f(x,y)$ is given, an analytic solution can be found, but if $f(x,y)$ is complicated it will be easier to numerically calculate the integral.

Now we add the components of $\mathbf{A}^{(L)}$ to $\mathbf{A}$ and $\vec{F}^{(L)}$ to $\vec{F}$. This is done by the local to global map. Furthermore, in our Python implementation we did this by giving each node inside Ω a unique identifier $k \in \{1,2,3,\ldots,N_I\}$ then for each node $\phi_j^{(L)}$ we find the global identifier k_j. This means in the current finite element E we have $\phi_j^{(L)} = \phi_{k_j}$. This leads to $\mathbf{A}_{k_j,k_i} += \mathbf{A}_{j,i}^{(L)}$ and $\vec{F}_{k_j}^{(L)} += \vec{F}_{k_j}$. We will now consider the case where the finite element E has at least one vertex on the boundary. Let $\phi_j^{(L)}$ be a node on the boundary this will give an identifier $k_j = 0$ this tells the Python program not to add row j or column j from $\mathbf{A}^{(L)}$ to $\mathbf{A}$ and not to add row j from $\vec{F}^{(L)}$ to $\vec{F}$. We need to do this because $u_h \in H_0^1(\Omega)$, therefore we know the solution is zero on the boundary.

8.2 Example 3: Dirichlet Poisson On Circle

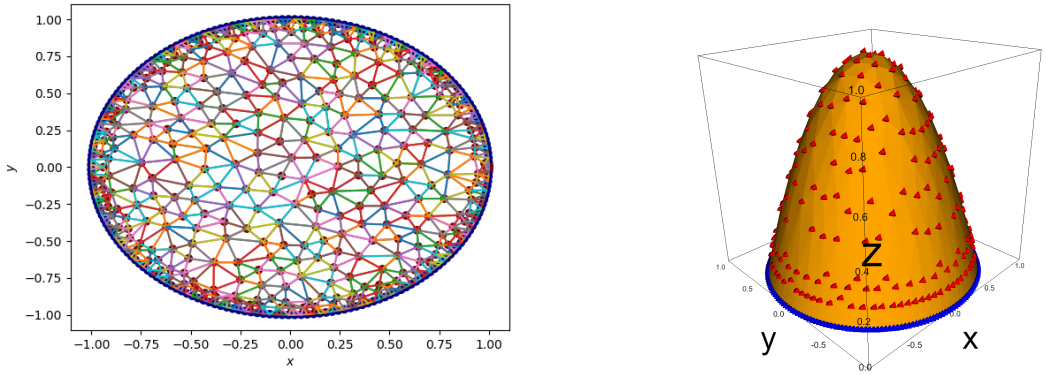
We will use our FEM code to solve the Dirichlet Poisson PDE (1) in 2D where our domain is a unit circle.

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = 4, \text{ for } x^2 + y^2 < 1, \tag{65}$$

$$u = 0, \text{ for } x^2 + y^2 = 1, \tag{66}$$

which has solution $u = 1 - x^2 - y^2$. This PDE example was inspired by [9]. Below we have our Python code and its corresponding output (Figure 5).

```
#Example_3.py
from PoissonSolver import Poisson2D; import numpy as np
f = lambda x, y: 4
sol = lambda x, y: 1-x**2-y**2
Example_3 = Poisson2D("Mesh/Circle_h02.1", f)
Example_3.plotMesh()
Example_3.plot2D(HTML = True)
print(Example_3.Norm(sol, order = 5))
```



(a) Circle Mesh, plotMesh

(b) FEM solution, plot2D

Figure 5: Output from Example_3.py

Where the red dots represent the positions of nodes. It is the position where $\phi_j = 1$.

8.3 Example 4: Dirichlet Poisson On Circle with K

Now we will use our FEM code to solve the Dirichlet Poisson PDE (1) in 2D with a spatially varying anisotropic coefficient $\mathbf{K}$ on a unit circle.

$$-\left(\begin{bmatrix}\frac{\partial}{\partial x} & \frac{\partial}{\partial y}\end{bmatrix}\begin{bmatrix}(x+1.1)^2 & 0\\ 0 & (y+1.1)^2\end{bmatrix}\begin{bmatrix}\frac{\partial u}{\partial x}\\ \frac{\partial u}{\partial y}\end{bmatrix}\right) = f(x,y), \text{ for } x^2+y^2<1 \quad (67)$$

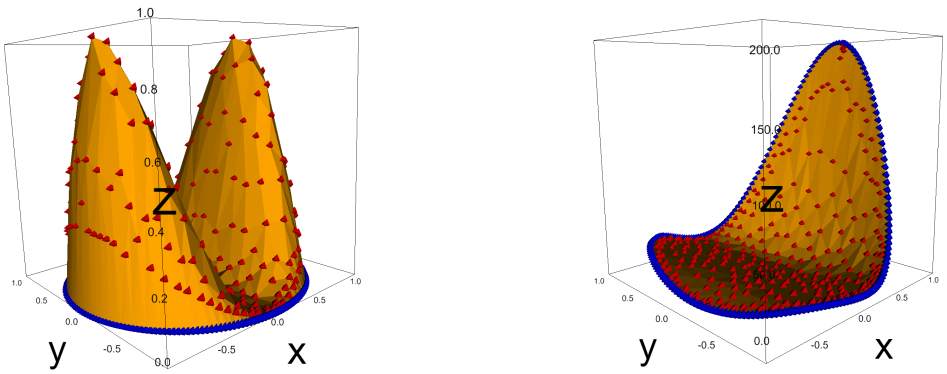
$$u = 0, \text{ for } x^2+y^2=1 \quad (68)$$

We will test our FEM code by setting the solution to be $u = 4x^2(1-x^2-y^2)$, then in our code we symbolically calculate $f(x,y)$ using the calc_f2D function.[2] Notice

[2] $f(x,y) = 80.0x^4 + 140.8x^3 + 48.0x^2y^2 + 35.2x^2y + 43.76x^2 + 35.2xy^2 - 35.2x + 9.68y^2 - 9.68$

that $\mathbf{K}$ is symmetric therefore we can use our formula (62) to calculate $A^{(L)}$ and it is positive definite on the domain. Below we have our Python code and its corresponding output (Figure 6)

```
#Example_4.py
from PoissonSolver import Poisson2D; import numpy as np
from Integration_Tools import calc_f2D
K = lambda x, y: np.array([[(x+1.1)**2,0],[0,(y+1.1)**2]])
sol = lambda x, y: 4*x**2*(1-x**2-y**2)
f = calc_f2D(sol, verbose=False, K = K)
Example_4 = Poisson2D("Mesh/Circle_h02.1", f, K=K)
Example_4.plot2D(HTML = True)
Example_4.plotf(f, HTML = True)
print(Example_4.Norm(sol, order = 5))
```



(a) FEM solution, plot2D

(b) $f(x, y)$, plotf

Figure 6: Output from Example_4.py

8.4 $||\mathbf{u} - \mathbf{u_h}||_{\mathbf{H^1(\Omega)}}$ for Example 3 and 4

Here we discuss the interpolation error in $H^1(\Omega)$ ($||u - u_h||_{H^1(\Omega)}$) for examples 3 and 4 with different mesh sizes. The last two columns refer to the interpolation error in $H^1(\Omega)$.

It is expected that when the mesh size h is halved, the value of $||u - u_h||_{H^1}$ for is halved (26). From (Table 2) we can see this is true for example 3. For example 4 we have $||u - u_h||_{H^1}$ is reduced when mesh size h is halved. For these results we have used "tri_integrate" with "order = 5", this means "Example_3.Norm(sol, order = 5)" and the output is truncated to fit the table. The order of the algorithm is explained in section 7.1.

h	File Name	Vertices	Triangles	Edges	Example 3	Example 4
0.4	Circle_h04.1	532	774	1305	0.2519722	1.0086274
0.3	Circle_h03.1	560	830	1389	0.1946745	0.9085195
0.2	Circle_h02.1	639	988	1626	0.1352740	0.7004109
0.1	Circle_h01.1	1130	1970	3099	0.0743830	0.4430134
0.05	Circle_h005.1	3445	6600	10044	0.0376425	0.2324851

Table 2: $||u - u_h||_{H^1(\Omega)}$ for decreasing mesh sizes h

9 Conclusion

In this paper, we used the finite element method to solve the Dirichlet Poisson equation with a spatially varying anisotropic coefficient on an arbitrary domain in one or two dimensions. Firstly, we used Lax-Milgram to prove there existed a unique solution. Then we used Galerkin Approximation to obtain a piecewise linear approximation for the solution. We derived a global approach in one dimension which can be considered equivalent to a finite difference method and we derived a local approach in two dimensions.

From these results a local approach was implemented in Python which used a third party C++ program for mesh triangulation in two dimensions. The Python implementation was tested on generated solutions and agreed with our convergence results. This gives a strong indication our Python code converges to the solution. Additionally, using a local approach leads to fast execution and using sparse matrices leads to low storage requirements. Finally, multiple examples were covered which means the reader can quickly test their own Dirichlet Poisson equation. All the Python code shown in this paper was written by me for this project.

Further research and development would be focused on deriving better numerical integration rules because the majority of execution time is spent on numerical integration. Additionally, translating into C++ or implementing in CUDA (Compute Unified Device Architecture) would lead to faster execution. The CUDA implementation would be a heterogeneous computing approach and would take advantage of the parallel nature of the problem.

References

[1] Patrick E. Farrell. *c6.4 Finite Element Methods for PDEs*. (Last Visited 06/04/2022). 2021. URL: https://people.maths.ox.ac.uk/farrellp/femvideos/notes.pdf.

[2] Professor Piaras Kelly. *Chapter 2: The (Galerkin) Finite Element Method*. (Last Visited 06/04/2022). 2020. URL: https://pkel015.connect.amazon.auckland.ac.nz/SolidMechanicsBooks/FEM/One_Dimensional/02_FE_Method.pdf.

[3] *PEP 8 – Style Guide for Python Code*. (Last Visited 06/04/2022). URL: https://peps.python.org/pep-0008/.

[4] *Race Conditions*. (Last Visited 26/04/2022). URL: https://en.wikipedia.org/wiki/Race_condition.

[5] Jonathan Richard Shewchuk. *Triangle: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator*. (Last Visited 06/04/2022). URL: http://www.cs.cmu.edu/~quake/triangle.html.

[6] Jonathan Richard Shewchuk. "Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator". In: *Applied Computational Geometry Towards Geometric Engineering*. Ed. by Ming C. Lin and Dinesh Manocha. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 203–222. ISBN: 978-3-540-70680-9.

[7] Wick Thomas. *Numerical Methods for Partial Differential Equations*. Jan. 29, 2020, pp. 138, 142. DOI: https://doi.org/10.15488/9248.

[8] Stephen Wiggins. "Elementary Quantum Mechanics". In: (Sept. 2021). (Last Visited 06/04/2022), pp. 12,31. DOI: 10.6084/m9.figshare.12928595.v6. URL: https://figshare.com/articles/book/Elementary_Quantum_Mechanics/12928595.

[9] Wikipedia. *Finite element method*. (Last Visited 06/04/2022). URL: https://en.wikipedia.org/wiki/Finite_element_method.

A Integration by Parts in $\mathbb{R}^n$

Integration by parts is the inverse of product rule. We have the product rule for scalar function g and vector function $\vec{A}$ is

$$\begin{aligned} \nabla \cdot (g\vec{A}) &= g\nabla \cdot \vec{A} + \vec{A} \cdot \nabla g, && (69) \\ \int_\Omega \nabla \cdot (g\vec{A}) d\Omega &= \int_\Omega g\nabla \cdot \vec{A} d\Omega + \int_\Omega \vec{A} \cdot \nabla g d\Omega, && (70) \\ \int_\Omega g\nabla \cdot \vec{A} d\Omega &= \int_{\partial\Omega} g\vec{A} \cdot \hat{n} dS - \int_\Omega \vec{A} \cdot \nabla g d\Omega && (71) \end{aligned}$$

where step (70) to (71) is done by using the divergence theorem and rearranging terms.

B Integrating Over Straight Line

Here we discuss some basic line integration algorithms. In our code, we use method 2, because the straight line integrals are usually over a small domain since the distance between mesh nodes is small. Therefore, method 2 gives a good approximation.

B.1 Method 1: Trapezium Rule

We split the interval into many trapeziums. Then sum the area of the trapeziums. Thus we get

$$\int_a^b f(x)dx \approx w(f(x_1) + f(x_N)) + \frac{w}{2}\left(\sum_{i=2}^{N-1} f(x_i)\right) \quad (72)$$

where $x_1, x_2, ..., x_N$ are equally spaced along the interval $[a, b]$. With $w = \frac{b-a}{N-1}$ and $x_i = a + (i-1)w$.

B.2 Method 2: Average

We create equally spaced points $x_1, x_2, ..., x_N$ between a and b. Then we take the mean of $f(x_1), f(x_2), ..., f(x_N)$ and multiple by the length of the domain $(b-a)$ to get

$$\int_a^b f(x)dx \approx \frac{b-a}{N}\sum_{i=1}^{N} f(x_i). \quad (73)$$

C Mesh-Generation

Mesh Generation in dimensions greater than one is complicated and a detailed description of methods for generation is beyond the scope of this paper. Thus we will use the c code Triangle [5]. For solving the PDE (1) in 2D we need a ".node" file this stores the position of the nodes. And we need a ".ele" file this stores which nodes each element is connected to. The files can be opened using a text editor (e.g. Notepad).

To create a Mesh with Delaunay triangulation. We first create a ".poly" this file describes the boundary of the shape. Then we run the bash code below, for this code example we use the file "Circle.poly".

```
1 gcc triangle.c -o Triangle -lm     #Compile mesh code
2 ./Triangle -q -p -u Circle.poly    #Calculate mesh for a Circle
```

With line 2 generating files "Circle.1.ele" and "Circle.1.node" to be used in our Python code. The "-q" ensures the mesh generates triangles with angles greater than 20 degrees. The "-p" tells the program to do mesh generation on a ".poly" file. The "-u" imposes a user-defined constraint on triangle size, our constraint was ensuring the mesh size (30) $h \leq C$. Where C is the upper limit of the mesh size we required. Furthermore, the "-aα" requires all triangles in the mesh have an area $\leq \alpha$. We only used the "-p -q -u" switches.

Additionally, the paper [6] explains the maths behind the mesh-generation.

D Bounding Matrix with Eigenvalues

We find the bound of $\vec{y}^T\mathbf{K}(\vec{x})\vec{y}$ for $\vec{y} \in \mathbb{R}^d$ and $\vec{x} \in \Omega$. Let us define

$$g(\mathbf{K}(\vec{x}), \vec{y}) = \frac{\vec{y}^T\mathbf{K}(\vec{x})\vec{y}}{|\vec{y}|^2} \tag{74}$$

using the fact distinct eigenvalues have orthogonal eigenvectors and the eigenvectors span the space for symmetric matrices. Theorem 1 from [8]. We expand $\vec{y}$ in terms of the eigenvectors of $\mathbf{K}(\vec{x})$. Thus, $y = \sum_i \zeta_i(\vec{x})v_i(\vec{x})$, where $v_i(\vec{x})$ are orthonormal eigenvectors of $\mathbf{K}(\vec{x})$. Thus we get these results

$$\vec{y}^T\vec{y} = \left(\sum_i \zeta_i(\vec{x})v_i(\vec{x})\right)\left(\sum_j \zeta_j(\vec{x})v_j(\vec{x})\right) = \sum_i \zeta_i^2(\vec{x}) \tag{75}$$

and we get

$$\vec{y}^T\mathbf{K}(\vec{x})\vec{y} = \vec{y}^T\left(\sum_i \zeta_i(\vec{x})\lambda_i(\vec{x})v_i(\vec{x})\right) = \sum_i \zeta_i^2(\vec{x})\lambda_i(\vec{x}) \leq \lambda_{max}(\mathbf{K}(\vec{x}))\sum_i \zeta_i^2(\vec{x}). \tag{76}$$

Using the same notation for λ_{min} and λ_{max} in (13) and (14) we get

$$\Lambda_{min} \leq \lambda_{min}(\mathbf{K}(\vec{x})) \leq \quad g(\mathbf{K}(\vec{x}), \vec{y}) \quad \leq \lambda_{max}(\mathbf{K}(\vec{x})) \leq \Lambda_{max}, \tag{77}$$

$$\Lambda_{min}|\vec{y}|^2 \leq \quad \vec{y}^T\mathbf{K}(\vec{x})\vec{y} \quad \leq \Lambda_{max}|\vec{y}|^2. \tag{78}$$

E Running The Code

To run the code we need to have these libraries: NumPy, Ipyvolume, SciPy, SymPy, webbrowser (Installed with default Python).

These modules can be easily installed by pip or conda. The best way to solve the PDE (1) with your choice of $f(\vec{x})$ and domain Ω is to copy code from the Examples given. Additionally, the functions and classes in these files have Docstrings and follow Python PEP8 style [3].

The examples need to be run in the same directory as "PoissonSolver.py", "Integration_Tools.py", "LocalTensor_1D.py" and "LocalTensor_2D.py". Or these files need to be put in a system PATH directory.

Included with my submission are extra mesh domains that can be used. When using the command line to view Ipyvolume surface plots, the program will download important files and open the output in the default web browser. (This is done as Ipyvolume is meant for interactive Python environments like Jupyter Notebook).

F Code File: Integration_Tool.py

```
#Integration_Tools.py
import numpy as np
import sympy as sp
from scipy.integrate import dblquad


def integrate(f, a, b, N):
    """
    Calculates the numerical value of the integral:
    \int_a^b f(x) dx,
    When N increases, the accuracy of the solution gets better

    Parameters
    ----------
    f : function
```

```
16          The function to integrate
17      a : float
18          The lower bound of the integral
19      b : float
20          The upper bound of the integral
21      N : int
22          The number of nodes to calculate the integral from
23      Returns
24      -------
25      float
26      The value of the integral
27
28      Examples
29      --------
30      >>> integrate(lambda x : x,0,1,5) #0.5
31      0.5
32      """
33      f = np.vectorize(f)
34      x = np.linspace(a, b, N)
35      fofx = f(x)
36      area = np.sum(fofx)*(b-a)/(N)
37      return area
38
39
40  def integrate_Trap(f, a, b, dx):
41      """
42      Calculates the inegral by Trapezium Rule
43      NOT USED: As too slow
44      """
45      f = np.vectorize(f)
46      if (a == b):
47          return 0
48      x = np.arange(a, b, dx)
49      fx = f(x)
50      area = dx*(fx[0]+fx[-1]+2*np.sum(fx[1:-1]))+(b-x[-1])*(fx[-1]+f(
        b))
51      return area/2
52
53
54  def tri_integrate_change_of_cords(f, a, b, c, N):
55      """
56      Calculates the inegral by Change of Cords
57      NOT USED: As too slow
```

```
    """
    # Calc dx
    f = np.vectorize(f)
    dx = np.sum(np.abs(a-b))/N

    def x(u, v): return a[0] + u*(b[0]-a[0]) + v*(c[0]-a[0])
    def y(u, v): return a[1] + u*(b[1]-a[1]) + v*(c[1]-a[1])
    det_J = np.abs((b[0]-a[0])*(c[1]-a[1])-(c[0]-a[0])*(b[1]-a[1]))
    return det_J*dblquad(lambda u, v: f(x(u, v), y(u, v)), 0, 1,
   lambda x: 0, lambda x: 1-x)[0]


def tri_integrate_No_Area(f, a, b, c, N):
    pos = (a+b+c)/3
    if N == 1:
        return f(*pos)
    else:
        return (tri_integrate_No_Area(f, a, b, pos, N-1) +
                tri_integrate_No_Area(f, b, c, pos, N-1) +
                tri_integrate_No_Area(f, c, a, pos, N-1))


def tri_integrate(f, a, b, c, N=1):
    """
    Using a recursive method, we calculate the integral over the
    triangle with points at a, b, c.
    And we calculate the integral
    \int_{E} f(x,y) dE

    Parameters
    ----------
    f : function 2D
        The function to be integrated
    a, b, c : numpy.ndarray, numpy.ndarray, numpy.ndarray,
        Each a, b, c represents the position of a vertex of a
   triangle
    N : numpy.ndarray, optional
        The larger N, the more accurate the numerical approximation,
        It calculates the integral using (3**(N-1)) nodes
        Defualt is N=1

    Returns
    -------
```

```
    float
        The numerical approximation of the triangle integral

    Examples
    --------
    >>> tri_integrate(lambda x,y:  x*y, np.array([0,0]),
    >>>               np.array([0,1]),np.array([1,0]),10) #0.0416
    ish
    0.04173668551409812

    """
    Area_Element = np.abs((a[0]*(b[1]-c[1]) +
                           b[0]*(c[1]-a[1]) +
                           c[0]*(a[1]-b[1]))/2)
    return tri_integrate_No_Area(f, a, b, c, N)*Area_Element/(3**(N
    -1))


def calc_f2D(sol_u, verbose=False, K= lambda x,y : np.array([[1, 0],
    [0, 1]])):
    """
    Using sympy we calculate the function f(x,y) from the sol_u, for
    the
    Dirichlet Poisson with Spatially Varying Anisotropic Coefficient

    Parameters
    ----------
    sol_u : str
        The solution of the PDE, with a sympy compatible functions
        i.e. use sympy.sin(x) instead of numpy.sin(x)
    verbose : bool, optional
        if true shows the answer for f(x,y), defaut is false
    K : function numpy.ndarray 2x2, optional
        Spatially Varying Anisotropic Coefficient, default is
        function I_2 (Identity Matrix)

    Returns
    -------
    function
    for f(x,y)
    """
    x, y = sp.symbols(r'x y')
    u = sol_u(x, y)
```

```
    K_sp = K(x,y)
    grad_u = sp.Matrix([[u.diff(x)], [u.diff(y)]])
    inside = K_sp*grad_u
    ans = -sp.simplify(inside[0].diff(x)+inside[1].diff(y))
    if verbose:
        print(ans)
    return sp.lambdify([x, y], ans)
```

G Code File: PoissonSolver.py

```
#PoissonSolver.py
import numpy as np #
import sympy as sp #
import matplotlib.pyplot as plt
import ipyvolume as ipv
import webbrowser
from scipy.sparse import csc_matrix
from scipy.sparse import lil_matrix
from scipy.sparse.linalg import spsolve
from scipy import interpolate


#Custom Modules
from LocalTensor_1D import *
from LocalTensor_2D import *
from Integration_Tools import *


#PoissonSolver.py
def IPV_Show_Solution(file_name, save_HTML = True, open_HTML = True,
    open_offline = True):
    """
    Displays 2D surface plots, by saving them as HTML files and then
    opening,
    or if in juypter notebook displays graphs directly below code.

    Parameters
    ----------
    file_name : str
        The name of the html to be saved, .html not needed added in
   code
    save_HTML : bool, optional
        If true views Ipyvolume plot via html, if false views plot
```

```
    below code
        Default is True
    open_HTML : bool, optional
        If true opens HTML after it has been created.
        Default is True
    open_offline : bool, optional
        if True, use local urls for required js/css packages and download all
        js/css required packages (if not already available), such that the html
        can be viewed with no internet connection. Online version doesn't work,
        as can't fetch data.
        Default is True
    Returns
    -------
    None
    """
    if save_HTML:
        ipv.save(file_name + ".html", offline = open_offline)
        if open_HTML:
            print("Opening in default webbrowser")
            webbrowser.open(file_name + ".html")
            print("Opened " + file_name + ".html" + " check default browser")
    else:
        ipv.show()

def Read_Mesh_File(Mesh_File_Name):
    """
    Reads the mesh file (.poly) and formats the information for the class Poisson2D.

    Parameters
    ----------
    Mesh_File_Name : str
        The directory of file from current location of running program.
        File to read should be (.poly) created by Mesh Generation,
        https://www.cs.cmu.edu/~quake/triangle.poly.html
    Returns
    -------
    Tuples, 3 numpy.ndarray
```

```
        Mesh_Nodes, Mesh_Elements, Is_Inner_Point
        Example output for a square
        Mesh_Nodes = [[0.1, 0.1], [1, 0.1], [1, 1], [0.1,
    1],[0.5,0.5]]
        Mesh_Elements = [[0,1],[1,2],[2,3],[3,4],[4,0]]
        Is_Inner_Point = [0,0,0,0,0]
    """

    # Open Contents of file
    nodes_file = open(Mesh_File_Name+".node", "r")
    elements_file = open(Mesh_File_Name+".ele", "r")

    # Format file information into floats and ints
    str_nodes = [str_node.split()
                 for str_node in nodes_file.read().splitlines()
    ][1:-1]
    nodes = np.array(str_nodes, dtype=float)

    str_elements = [str_element.split()
                    for str_element in elements_file.read().
    splitlines()][1:-1]
    elements = np.array(str_elements, dtype=int)

    nodes_file.close()
    elements_file.close()

    # Format into array, so program can use
    Is_Inner_Point = np.array(1-nodes[:, 3], dtype=int)  # Could use
    bool
    # + np.array([1,1]) #Can add value to translate the mesh
    Mesh_Nodes = nodes[:, [1, 2]]
    Mesh_Elements = elements[:, [1, 2, 3]] - 1
    return Mesh_Nodes, Mesh_Elements, Is_Inner_Point


def Get_Norm_1D(Mesh_Nodes, Mesh_Elements, FEM_sol_All, sol, D=1):
    """
    Gets the error ||u-u_h||_{H^1} in 1D

    Parameters
    ----------
    Mesh_Nodes : numpy.ndarray
        Stores location of nodes
```

```
    Mesh_Elements : numpy.ndarray
        Stores element links, with pointers pointing to Nodes
    FEM_sol_All : numpy.ndarray
        Stores the solution for u_h calculated by the Finite Element
    Method
    sol : function
        Function for the actual solution of the PDE
    Returns
    -------
    ||u-u_h||_{H^1} in 1D
    """

    # Calculate u_x
    x_sp = sp.symbols('x')
    sol_x = sol(x_sp).diff(x_sp)
    u_x = sp.lambdify([x_sp], sol_x)

    Total_Error = 0
    # Loop over all elements in domain
    # Split integral over all elements in domain
    for element_k in Mesh_Elements:
        # Get positions of nodes in current element
        pos_k = Mesh_Nodes[element_k]
        u_k = FEM_sol_All[element_k]
        Matrix_Coef = np.array([[1, *pos] for pos in pos_k])
        Coef_ = np.linalg.solve(Matrix_Coef, np.identity(len(pos_k))
    )
        '''
        Coef_ = [[\alpha_1, \alpha_2],
                 [\beta_1 , \beta_2 ]]

        \phy_1^{L} = \alpha_1 + \beta_1 x
        \phy_2^{L} = \alpha_2 + \beta_2 x

        u_h = \phy_1^{L}*u_1 + \phy_2^{L}*u_2
        u_h = \alpha_1*u_1 + \alpha_2*u_2 + (\beta_1*u_1 + \beta_2*
    u_2)*x
        '''
        A_c, B_c = np.dot(Coef_[0], u_k), np.dot(Coef_[1], u_k)
        def u_h(x): return A_c + B_c*x  # Gives the FEM
    approximation of u_h
        u_h_x = B_c
        def to_int(x): return (sol(x)-u_h(x))**2 + D*((u_x(x)-u_h_x)
```

```
    **2)
145         Total_Error += integrate(to_int, pos_k[0, 0], pos_k[1, 0],
    10)
146     return Total_Error**0.5
147
148
149 def Get_Norm_2D(Mesh_Nodes, Mesh_Elements, FEM_sol_All, sol, D=1,
    order = 1):
150     """
151     Gets the error ||u-u_h||_{H^1} in 2D
152
153     Parameters
154     ----------
155     Mesh_Nodes : numpy.ndarray
156         Stores location of nodes
157     Mesh_Elements : numpy.ndarray
158         Stores element links, with pointers pointing to Nodes
159     FEM_sol_All : numpy.ndarray
160         Stores the solution for u_h calculated by the Finite Element
     Method
161     sol : function
162         Function for the actual solution of the PDE
163     D : int
164         Calculates ||u-u_h||_{H^D}, only works for 0 and 1, Default
    is 1
165     order : int
166         The order to take the tri_integrate function. Default is 1
167     Returns
168     -------
169     ||u-u_h||_{H^1} in 2D
170     """
171     # Calculate u_x, u_y
172     x_sp, y_sp = sp.symbols('x y')
173     sol_x = sol(x_sp, y_sp).diff(x_sp)
174     sol_y = sol(x_sp, y_sp).diff(y_sp)
175     u_x = sp.lambdify([x_sp, y_sp], sol_x)
176     u_y = sp.lambdify([x_sp, y_sp], sol_y)
177
178     Total_Error = 0
179     # Loop over all elements in domain
180     # Split integral over all elements in domain
181     for element_k in Mesh_Elements:
182         # Get positions of nodes in current element
```

```
183         pos_k = Mesh_Nodes[element_k]
184         u_k = FEM_sol_All[element_k]  # Coeficent value
185         Matrix_Coef = np.array([[1, *pos] for pos in pos_k])
186         Coef_ = np.linalg.solve(Matrix_Coef, np.identity(len(pos_k))
    )
187         A_c, B_c, C_c = np.dot(Coef_[0], u_k), np.dot(
188             Coef_[1], u_k), np.dot(Coef_[2], u_k)
189         '''
190         Coef_ = [[\alpha_1, \alpha_2, \alpha_3],
191                  [\beta_1 , \beta_2 , \beta_3 ],
192                  [\zeta_1 , \zeta_2 , \zeta_3 ]]
193 
194         \phy_1^{L} = \alpha_1 + \beta_1 x + \zeta_1 y
195         \phy_2^{L} = \alpha_2 + \beta_2 x + \zeta_2 y
196         \phy_3^{L} = \alpha_3 + \beta_3 x + \zeta_3 y
197 
198         u_h = \phy_1^{L}*u_1 + \phy_2^{L}*u_2 + \phy_3^{L}*u_3
199         u_h = \alpha dot u_k + \beta dot u_k * x + \zeta dot u_k * y
200         '''
201         def u_h(x, y): return A_c + B_c*x + C_c*y
202         u_h_x = B_c
203         u_h_y = C_c
204         def to_int(x, y): return (sol(x, y)-u_h(x, y))**2 + D * \
205             ((u_x(x, y)-u_h_x)**2+(u_y(x, y)-u_h_y)**2)
206         Total_Error += tri_integrate(to_int, *pos_k, order)
207     return Total_Error**0.5
208 
209 
210 def Get_Equation(Mesh_Nodes, Mesh_Elements, Is_Inner_Point,
    Calc_Local_A_F, f, K):
211     """
212     Gets the error ||u-u_h||_{H^1} in 2D
213 
214     Parameters
215     ----------
216     Mesh_Nodes : numpy.ndarray
217         Stores location of nodes
218     Mesh_Elements : numpy.ndarray
219         Stores element links, with pointers pointing to Nodes
220     Is_Inner_Point : numpy.ndarray
221         Stores if the point is on the boundary(0) or inside the
    domain(1)
222     Calc_Local_A_F : function
```

```
223         The function for calculating Local Load Vector f^{L} and
    Local
224         stiffness matrix A^{L}
225     f : function
226         The function f(\vec{x}) in the PDE to solve
227     K : numpy.ndarray
228         The Spatially Varying Anisotropic Coefficient in the PDE
229     Returns
230     -------
231     tuple 2
232         A_Global_Sparse : scipy.sparse.lil.lil_matrix
233             Global stiffness matrix
234         F_Global : numpy.ndarray
235             Global load vector
236     """
237     N_IP = np.sum(Is_Inner_Point)  # Number of Inner Points
238     N_n = len(Mesh_Nodes)  # Number of nodes
239     # Inner_Point_Index creates an index of all the points inside
    the Boundary
240     Inner_Point_Index = np.array(
241         [np.sum(Is_Inner_Point[0:i]) for i in range(N_n)])
242
243     # Initialise Sparse matrix A (Stiffness Matrix) and F,
244     # Sparse Global Stiffness Matrix
245     A_Global_Sparse = lil_matrix((N_IP, N_IP))
246     F_Global = np.zeros([N_IP, 1])
247
248     # Loop over all elements in Mesh
249     for element_k in Mesh_Elements:
250         # element_k, Global Index of edges of current element e.g.
    [3,4,5]
251
252         # Get positions of nodes in current element
253         pos_k = Mesh_Nodes[element_k]
254
255         # Use to check if completly inside domain
256         pos_k_I = Is_Inner_Point[element_k]
257         # Is this local node in inner domain
258         # Locates local Index of points inside domain
259         Inner_Element_Local_Index = np.where(pos_k_I == 1)
260         d = sum(pos_k_I)  # Number of local points inside domain
261
262         # Calc IP (Inner Points) index (Index of points inside
```

```
    domain)
263         IP_Index = Inner_Point_Index[element_k]
264         # Remove elements on boundary
265         IP_Index = IP_Index[Inner_Element_Local_Index]
266         # Can be considered as the local-to-global map
267 
268         # Calculate Local Tensors
269         A_Local, F_Local = Calc_Local_A_F(pos_k, f, K)
270 
271         # Remove boundary elements as we already know the answer
272         F_Local = F_Local[Inner_Element_Local_Index]
273         A_Local = A_Local[Inner_Element_Local_Index[0],
274                           :][:, Inner_Element_Local_Index[0]]
275         # Now each collum and row line up with the IP_Index
276 
277         # Now Add to global matrix
278         for j in range(d):
279             F_Global[IP_Index[j]] += F_Local[j]
280             for i in range(d):
281                 # Depends if you can calculate it straight
282                 A_Global_Sparse[IP_Index[j], IP_Index[i]] += A_Local
    [j, i]
283     return A_Global_Sparse, F_Global
284 
285 
286 class Poisson1D():
287     """
288     Class that is used to solve the 1D Dirichlet Poisson with a
    Spatially
289     Varying Anisotropic Coefficient problem, using the Finite
    Element Method
290 
291     Equation is:
292 
293     -\dot{(K\dot{u})} = f(x) in domain
294     u = 0 on boundary
295 
296     Attributes
297     ----------
298     a : float
299         position of left boundary
300     b : float
301         position of right boundary
```

```
302     f : function
303         The function representing the source term
304     N_n : int
305         The number of nodes in the mesh
306     uniform : bool, optional
307         if true the nodes in the mesh will be evenly spaced
308     K : numpy.ndarray 1x1 function, optional
309         The Spatially Varying Anisotropic Coefficient
310 
311     Methods
312     -------
313     plot1D(self, sol = None):
314         Displays the plot of the FEM calculation,
315     plotError(self,sol):
316         Displays the plot of the error |u-u_h|
317     Norm(self, sol, k=1):
318         Calculates the error of the FEM calculation, ||u-u_h||_{H^k
    },
319         only valid when k=1 or k=0
320     u_h(x):
321         Linear interpolate between points
322     """
323 
324     def __init__(self, a, b, f, N_n, uniform=True, K=None):
325         self.a = a
326         self.b = b
327         self.N_n = N_n
328         N_e = N_n - 1  # Number of Elements
329 
330         # Calculate Mesh
331         self.Mesh_Nodes = np.reshape(
332             np.array([a, *(np.sort(np.random.rand(N_n-2))*(b-a)+a),
    b]), (-1, 1))
333         if uniform:
334             self.Mesh_Nodes = np.reshape(np.linspace(a, b, N_n),
    (-1, 1))
335         # Create array of nodes which are linked
336         self.Mesh_Elements = np.array([[i, i+1] for i in range(0,
    N_e)])
337         Is_Inner_Point = np.array(
338             [0, *np.ones(N_n-2), 0], dtype=np.int32)  # 0 means on
    boundary
339 
```

```
340         # Chose correct local calculation
341         Calc_Local_A_F = Local_A_F_1D
342         if K != None:
343             Calc_Local_A_F = Local_A_F_1D_with_K
344 
345         # Get the linear system to solve
346         A_Global_Sparse, F_Global = Get_Equation(self.Mesh_Nodes,
    self.Mesh_Elements,
347                                                 Is_Inner_Point,
    Calc_Local_A_F, f, K)
348 
349         # Solve System of linear equations
350         self.FEM_sol = spsolve(csc_matrix(A_Global_Sparse), F_Global
    )
351         # Solution that includes boundary data
352         self.FEM_sol_All = np.zeros(N_n)
353         self.FEM_sol_All[np.where(Is_Inner_Point == 1)] = self.
    FEM_sol
354 
355         # Create function u_h to linear interpolate
356         self.u_h = interpolate.interp1d(np.reshape(
357             self.Mesh_Nodes, (1, -1))[0], self.FEM_sol_All)
358 
359     def plot1D(self, sol=None):
360         """
361         Parameters
362         ----------
363         sol : function, optional
364             If function given for sol, plots the solution in red
365 
366         Returns
367         -------
368         None
369         """
370         x = np.linspace(self.a, self.b)
371         if sol != None:
372             # Plots true solution if true solution given
373             plt.plot(x, sol(x), color="Red")
374         # Plots position and value of nodes
375         plt.scatter(self.Mesh_Nodes, self.FEM_sol_All)
376         # Displayes Links between nodes
377         for Mesh_Element in self.Mesh_Elements:
378             pos_ = np.array([self.Mesh_Nodes[Mesh_Element[i]]
```

```
                                for i in range(2)])
            x = pos_[:, 0]
            y = np.array([self.FEM_sol_All[Mesh_Element[i]] for i in
    range(2)])
            plt.plot(x, y, linestyle='dashed', color="blue")
        plt.xlabel(r"$x$")
        plt.ylabel(r"$y$")
        plt.show()

    def plotError(self, sol):
        """
        Parameters
        ----------
        sol : function
            Solution of the PDE problem

        Returns
        -------
        None
        """
        x = np.linspace(self.a, self.b, 30*self.N_n)
        x_Mesh = np.reshape(self.Mesh_Nodes, (1, -1))[0]
        plt.scatter(x_Mesh, np.abs(self.FEM_sol_All-sol(x_Mesh)))
        plt.plot(x, np.abs(self.u_h(x)-sol(x)), color="Red")
        plt.xlabel(r"$x$")
        plt.title(r"$y=|u(x)-u_h(x)|$")
        plt.show()

    def Norm(self, sol, k=1):
        """
        Parameters
        ----------
        sol : function
            Solution of the PDE problem
        k : int, optional
            Calculates ||u-u_h||_{H^k}, only valid when k=1 or k=0

        Returns
        -------
        float
            ||u-u_h||_{H^k}, only valid when k=1 or k=0
        """
        return Get_Norm_1D(self.Mesh_Nodes, self.Mesh_Elements,
```

```
                                self.FEM_sol_All, sol, k)


class Poisson2D():
    """
    Class that is used to solve the 2D Dirichlet Poisson with a Spatially
    Varying Anisotropic Coefficient problem, using the Finite Element
    Method

    Equation is:

    -\nabla \cdot (K\nabla{u}) = f(x) in domain
    u = 0 on boundary

    Attributes
    ----------
    Mesh_File_Name : str
        The directory of file from current location of running program.
        File to read should be (.poly) created by Mesh Generation,
        https://www.cs.cmu.edu/~quake/triangle.poly.html
    f : function
        The function representing the source term
    K : numpy.ndarray 2x2 function, optional
        The Spatially Varying Anisotropic Coefficient, default is None
        if None K gets treated as identity matrix. Uses different code to
        make computation faster.

    Methods
    -------
    plotMesh(self, links = True):
        Displays the mesh to be used in the FEM calculation
    plot2D(self, surface = True, HTML = True):
        Displays the plot of the solution calculated by FEM.
        REQUIRES ipyvolume
    plotf(self, surface = True, nodes = True, HTML = True):
        Displays the plot of the source term f on the domain
        REQUIRES ipyvolume
    Norm(self, sol, k=1):
```

```
459         Calculates the error of the FEM calculation, ||u-u_h||_{H^k
    },
460         only valid when k=1 or k=0
461     Get_h(self):
462         Calculates the largest side length of a traingle in the
    domain.
463         i.e. it gets the mesh size.
464     u_h(x):
465         Linear interpolate between points. NOT DONE
466     """
467     # Need function to calculate mesh size
468 
469     def __init__(self, Mesh_File_Name, f, K=None):
470         # N_n is number of nodes
471         # Calculate Mesh
472         self.Mesh_Nodes, self.Mesh_Elements, self.Is_Inner_Point =
    Read_Mesh_File(
473             Mesh_File_Name)
474         N_n = len(self.Mesh_Nodes)
475         self.f = f
476         # Chose correct local calculation
477         Calc_Local_A_F = Local_A_F_2D
478         if K != None:
479             Calc_Local_A_F = Local_A_F_2D_with_K
480 
481         #
482         A_Global_Sparse, F_Global = Get_Equation(self.Mesh_Nodes,
    self.Mesh_Elements,
483                                                 self.Is_Inner_Point
    , Calc_Local_A_F,
484                                                 f, K)
485 
486         # Solve System
487         self.FEM_sol = spsolve(csc_matrix(A_Global_Sparse), F_Global
    )
488         # np.reshape(FEM_sol,(1,-1))[0]
489         self.FEM_sol_All = np.zeros(N_n)  # Includes Boundary data
490         self.FEM_sol_All[np.where(self.Is_Inner_Point == 1)] = self.
    FEM_sol
491 
492     def plotMesh(self, links=True):
493         """
494         Parameters
```

```
        ----------
        links : bool, optional
            If true plots lines between the nodes
            Default is True
        Returns
        -------
        None
        """
        plt.scatter(self.Mesh_Nodes[:, 0],
                    self.Mesh_Nodes[:, 1], c=self.Is_Inner_Point,
                    cmap = "jet")
        if links:
            for Mesh_Element in self.Mesh_Elements:
                pos_ = np.array([self.Mesh_Nodes[Mesh_Element[i]]
                                 for i in range(3)])
                x = [*pos_[:, 0], pos_[0, 0]]
                y = [*pos_[:, 1], pos_[0, 1]]
                plt.plot(x, y)
        plt.xlabel(r"$x$")
        plt.ylabel(r"$y$")
        plt.show()

    def plot2D(self, surface=True, HTML = True):  # , sol = None):
        """
        Parameters
        ----------
        surface : bool, optional
            If true plots the surface of the solution u_h,
            Default is True
        HTML : bool, optional
            If true saves the output plot to a html file, then
            views the file with default webbrowser. Change to false
            if in Juypter Notebook and graph displays below code.
            Default is True
        Returns
        -------
        None
        """
        # if sol != None:
        # Plot wireframe of solution
        #plt.plot(x, sol(x), color = "Red")
        x = self.Mesh_Nodes[:, 0][np.where(self.Is_Inner_Point == 1)
]
```

```
537         y = self.Mesh_Nodes[:, 1][np.where(
538             self.Is_Inner_Point == 1)]  # transpose
539         z = self.FEM_sol
540         x_0 = self.Mesh_Nodes[:, 0][np.where(self.Is_Inner_Point ==
    0)]
541         y_0 = self.Mesh_Nodes[:, 1][np.where(self.Is_Inner_Point ==
    0)]
542         z_0 = np.zeros(len(x_0))
543 
544         fig = ipv.figure()
545         scatter_inner = ipv.scatter(x, y, z)
546         scatter_outer = ipv.scatter(x_0, y_0, z_0, color="blue")
547         if surface:
548             ipv.plot_trisurf(self.Mesh_Nodes[:, 0], self.Mesh_Nodes
    [:, 1],
549                              self.FEM_sol_All, triangles=self.
    Mesh_Elements,
550                              color='orange')
551         # Plot flat surface
552         #ipv.show()
553         IPV_Show_Solution("Surface_Plot", HTML)
554 
555     def plotf(self, surface=True, nodes=True, HTML = True):
556         """
557         Parameters
558         ----------
559         surface : bool, optional
560             If true plots flat surface between nodes
561             Default is True
562         nodes : bool, optional
563             If true does a scatter plot for the nodes
564             Default is True
565         HTML : bool, optional
566             If true saves the output plot to a html file, then
567             views the file with default webbrowser. Change to false
568             if in Juypter Notebook and graph displays below code.
569             Default is True
570         Returns
571         -------
572         None
573         """
574         fig = ipv.figure()
575         if nodes:
```

```
576             # Get nodes inside the boundary
577             x = self.Mesh_Nodes[:, 0][np.where(self.Is_Inner_Point
    == 1)]
578             y = self.Mesh_Nodes[:, 1][np.where(self.Is_Inner_Point
    == 1)]
579             z = self.f(x, y)
580 
581             # Get nodes on the boundary
582             x_0 = self.Mesh_Nodes[:, 0][np.where(self.Is_Inner_Point
     == 0)]
583             y_0 = self.Mesh_Nodes[:, 1][np.where(self.Is_Inner_Point
     == 0)]
584             z_0 = self.f(x_0, y_0)
585             scatter_inner = ipv.scatter(x, y, z)
586             scatter_outer = ipv.scatter(x_0, y_0, z_0, color="blue")
587         if surface:
588             ipv.plot_trisurf(self.Mesh_Nodes[:, 0], self.Mesh_Nodes
    [:, 1],
589                              self.f(self.Mesh_Nodes[:, 0],
590                                     self.Mesh_Nodes[:, 1]),
591                              triangles=self.Mesh_Elements, color='
    orange')
592         # Plot flat surface
593         #ipv.show()
594         IPV_Show_Solution("f_plot", HTML)
595 
596     def Norm(self, sol, k=1, order = 1):
597         """
598         Parameters
599         ----------
600         sol : function
601             Solution of the PDE problem
602         k : int, optional
603             Calculates ||u-u_h||_{H^k}, only valid when k=1 or k=0
604         order : int
605             The order to take the tri_integrate function. Default is
     1
606         Returns
607         -------
608         float
609             ||u-u_h||_{H^k}, only valid when k=1 or k=0
610         """
611         return Get_Norm_2D(self.Mesh_Nodes, self.Mesh_Elements,
```

```
                            self.FEM_sol_All, sol, k, order)

    def Get_h(self):
        """
        Parameters
        ----------
        None
        Returns
        -------
        float
            max side length of triangle in mesh
        """
        max_h = 0
        #Will repeat some links twice, but since simple calculation
        #efficiency is not important
        for element_k in self.Mesh_Elements:
            # Get positions of nodes in current element
            pos_k = self.Mesh_Nodes[element_k]
            #pos_k = [[x_1,y_1],[x_2,y_2],[x_3,y_3]]
            L1 = np.linalg.norm(pos_k[0]-pos_k[1])
            L2 = np.linalg.norm(pos_k[0]-pos_k[1])
            L3 = np.linalg.norm(pos_k[0]-pos_k[1])
            max_h = np.max([L1,L2,L3,max_h])
        return max_h

    def u_h(self, x, y):
        return 0
```

H Code File: LocalTensor_1D.py

```
# LocalTensor1D.py Start
import numpy as np
from Integration_Tools import *


def Local_F_1D(x_j, x_jp1, f):
    """
    Calculates the local load vector for Galerkin Approximation in 1D
    Note, does not depend on K (Spatially Varying Anisotropic Coefficient)

    Parameters
```

```
    ----------
    x_j : float
        Left point of the element
    x_jp1 : float
        Right point of the element
    f : function
        The source term function of the PDE

    Returns
    -------
    numpy.ndarray 2
        The loacl load vector F

    Examples
    --------
    >>> Local_F_1D(0.5,0.6,lambda x : x)
    array([0.02648148, 0.02851852])

    """
    # Calculate the 2 phy's for the straight line finite element
    # Only valid on finite element
    phy_k = [lambda x: ((x_jp1-x)/(x_jp1-x_j)),
             lambda x: ((x-x_j)/(x_jp1-x_j))]
    # Then Calculate F_Local
    F_Local = np.array([integrate(lambda x: phy_k[0](x)*f(x), x_j, x_jp1, 10),
                        integrate(lambda x: phy_k[1](x)*f(x), x_j, x_jp1, 10)])
    # Slow consider finding analytic
    return F_Local


def Local_A_F_1D(pos_k, f, K):
    """
    Calculates the local load vector and the local stiffness matrix
    for Galerkin Approximation in 1D,

    Parameters
    ----------
    pos_k : numpy.ndarray
        Contains the nodes of the finite element
    f : function
        The source term function of the PDE
```

```
    K :
        Spatially Varying Anisotropic Coefficient
        NOT USED in this function, assumed to be 1
    Returns
    -------
    tuple 2,
        A_Local : numpy.ndarray 2x2
            Stores the local stiffness tensor
        F_Local : numpy.ndarray 2
            The loacl load vector F

    Examples
    --------
    >>> Local_A_F_1D(pos_k = numpy.array([[0.5],[0.6]]),f = lambda x
    : x, K = None)
    (array([[ 10., -10.],[-10.,  10.]]), array([0.02648148,
   0.02851852]))

    """
    # Case K = 1
    x_jp1 = pos_k[1][0]
    x_j = pos_k[0][0]

    F_Local = Local_F_1D(x_j, x_jp1, f)

    A_jj = 1/(x_jp1-x_j)
    A_Local = np.array([[A_jj, -A_jj], [-A_jj, A_jj]])

    return A_Local, F_Local


def Local_A_F_1D_with_K(pos_k, f, K):
    """
    Calculates the local load vector and the local stiffness matrix
    for Galerkin Approximation in 1D, with a Spatially Varying
    Anisotropic Coefficient

    Parameters
    ----------
    pos_k : numpy.ndarray
        Contains the nodes of the finite element
    f : function
        The source term function of the PDE
```

```
    K : numpy.ndarray 1x1 function
        Spatially Varying Anisotropic Coefficient
    Returns
    -------
    tuple 2,
        A_Local : numpy.ndarray 2x2 float
            Stores the local stiffness tensor
        F_Local : numpy.ndarray 2 float
            The loacl load vector F

    Examples
    --------
    >>> Local_A_F_1D_with_K(pos_k = numpy.array([[0.5],[0.6]]),
                            f = lambda x : x, K = lambda x :  numpy.
    array([[x]]))
    (array([[ 5.5, -5.5], [-5.5,  5.5]]), array([0.02648148,
    0.02851852]))
    """
    x_jp1 = pos_k[1][0]
    x_j = pos_k[0][0]

    F_Local = Local_F_1D(x_j, x_jp1, f)

    K_int = integrate(K, x_j, x_jp1, 10)  # integral of K over
    domain
    width = 1/(x_jp1-x_j)
    A_Local = np.array([[K_int, -K_int], [-K_int, K_int]])*(width
    **2)
    return A_Local, F_Local
```

I Code File: LocalTensor_2D.py

```
#LocalTensor_2D.py
import numpy as np
from Integration_Tools import *


def Local_F_2D(pos_k, Coef_, f):
    """
    Calculates the local load vector for Galerkin Approximation in 2
    D
    Note, does not depend on K (Spatially Varying Anisotropic
    Coefficient)
```

```
10
11      Parameters
12      ----------
13      pos_k : numpy.array, 3x2 float
14          The nodes of the finite element triangle
15      Coef_ : numpy.array, 3x3 float
16          Has the coefficents for the each of the phi's
17      f : function
18          The source term function of the PDE
19
20      Returns
21      -------
22      numpy.ndarray 3 float
23          The loacl load vector F
24
25      Examples
26      --------
27      >>> pos_k = numpy.array([[0,0],[0,1],[1,0]])
28      >>> Coef_ = numpy.array([[1., -1., -1.], [ 0.,  0.,  1.], [ 0.,
        1.,  0.]])
29      >>> Local_F_2D(pos_k, Coef_, f = lambda x, y : x*y)
30      array([0.00901793, 0.01846771, 0.01846771])
31      """
32
33      # Only on triangle domain
34      phy_k = [lambda x, y: np.dot(Coef_[0], np.array([1, x, y])),
35               lambda x, y: np.dot(Coef_[1], np.array([1, x, y])),
36               lambda x, y: np.dot(Coef_[2], np.array([1, x, y]))]
37
38      F_Local = np.array([tri_integrate(lambda x, y: phy_k[0](x, y)*f(
    x, y), *pos_k, 3),
39                          tri_integrate(lambda x, y: phy_k[1](x, y)*f(
    x, y), *pos_k, 3),
40                          tri_integrate(lambda x, y: phy_k[2](x, y)*f(
    x, y), *pos_k, 3)])
41
42      return F_Local
43
44
45  def Local_A_F_2D(pos_k, f, K):
46      """
47      Calculates the local load vector and the local stiffness matrix
48      for Galerkin Approximation in 2D,
```

```
49
50      Parameters
51      ----------
52      pos_k : numpy.ndarray 3x2
53          Contains the nodes of the triangle finite element
54      f : function
55          The source term function of the PDE
56      K :
57          Spatially Varying Anisotropic Coefficient
58          NOT USED in this function, assumed to be 1
59      Returns
60      -------
61      tuple 2,
62          A_Local : numpy.ndarray 3x3 float
63              Stores the local stiffness tensor
64          F_Local : numpy.ndarray 2 float
65              The loacl load vector F
66
67      Examples
68      --------
69      >>> pos_k = numpy.array([[0,0],[0,1],[1,0]])
70      >>> Local_A_F_2D(pos_k,f = lambda x, y : x*y, K = None)
71      (array([[ 1. , -0.5, -0.5], [-0.5,  0.5,  0. ], [-0.5,  0. ,
    0.5]]),
72       array([0.00901793, 0.01846771, 0.01846771]))
73      """
74      # Case K = Identity_2
75      Matrix_Coef = np.array([[1, *pos_k[0]], [1, *pos_k[1]], [1, *
    pos_k[2]]])
76      Coef_ = np.linalg.solve(Matrix_Coef, np.array([[1, 0, 0],
77                                                     [0, 1, 0],
78                                                     [0, 0, 1]])).T
79      '''
80      Coef_ = [[\alpha_1, \beta_1, \zeta_1],
81               [\alpha_2, \beta_2, \zeta_2],
82               [\alpha_3, \beta_3, \zeta_3]]
83      '''
84
85      F_Local = Local_F_2D(pos_k, Coef_, f)
86
87      # Case K = Identity_2
88      Area_Element = np.abs((pos_k[0, 0]*(pos_k[1, 1]-pos_k[2, 1]) +
89                             pos_k[1, 0]*(pos_k[2, 1]-pos_k[0, 1]) +
```

```
                            pos_k[2, 0]*(pos_k[0, 1]-pos_k[1, 1]))/2)

    A_Local = Area_Element*np.matmul(Coef_[:, [1, 2]], Coef_[:, [1, 2]].T)
    # Case K = Identity_2 End

    return A_Local, F_Local


def Local_A_F_2D_with_K(pos_k, f, K):
    """
    Calculates the local load vector and the local stiffness matrix
    for Galerkin Approximation in 2D, with a Spatially Varying
    Anisotropic Coefficient

    Parameters
    ----------
    pos_k : numpy.ndarray 3x2 float
        Contains the nodes of the finite element
    f : function
        The source term function of the PDE
    K : numpy.ndarray 2x2 function
        Spatially Varying Anisotropic Coefficient
    Returns
    -------
    tuple 2,
        A_Local : numpy.ndarray 3x3 float
            Stores the local stiffness tensor
        F_Local : numpy.ndarray 3 float
            The loacl load vector F

    Examples
    --------
    >>> pos_k = numpy.array([[0,0],[0,1],[1,0]])
    >>> K = lambda x,y : numpy.array([[x,0],[0,y]])
    >>> Local_A_F_2D_with_K(pos_k,f = lambda x, y : x*y,K=K)
    (array([[ 0.33333333, -0.16666667, -0.16666667],
            [-0.16666667,  0.16666667,  0.        ],
            [-0.16666667,  0.        ,  0.16666667]]),
     array([0.00901793, 0.01846771, 0.01846771]))
    """
    Matrix_Coef = np.array([[1, *pos_k[0]], [1, *pos_k[1]], [1, *pos_k[2]]])
```

```
    Coef_ = np.linalg.solve(Matrix_Coef, np.array([[1, 0, 0],
                                                   [0, 1, 0],
                                                   [0, 0, 1]])).T
    '''
    Coef_ = [[\alpha_1, \beta_1, \zeta_1],
             [\alpha_2, \beta_2, \zeta_2],
             [\alpha_3, \beta_3, \zeta_3]]
    '''

    F_Local = Local_F_2D(pos_k, Coef_, f)
    K_int = np.array([[tri_integrate(lambda x, y: K(x, y)[j, i], *
pos_k, 3)
                       for i in range(2)]
                      for j in range(2)])

    A_Local = np.matmul(np.matmul(Coef_[:, [1, 2]], K_int), Coef_[:,
    [1, 2]].T)

    return A_Local, F_Local
    # End Case K = K(x)
```