UiT
THE ARCTIC UNIVERSITY OF NORWAY

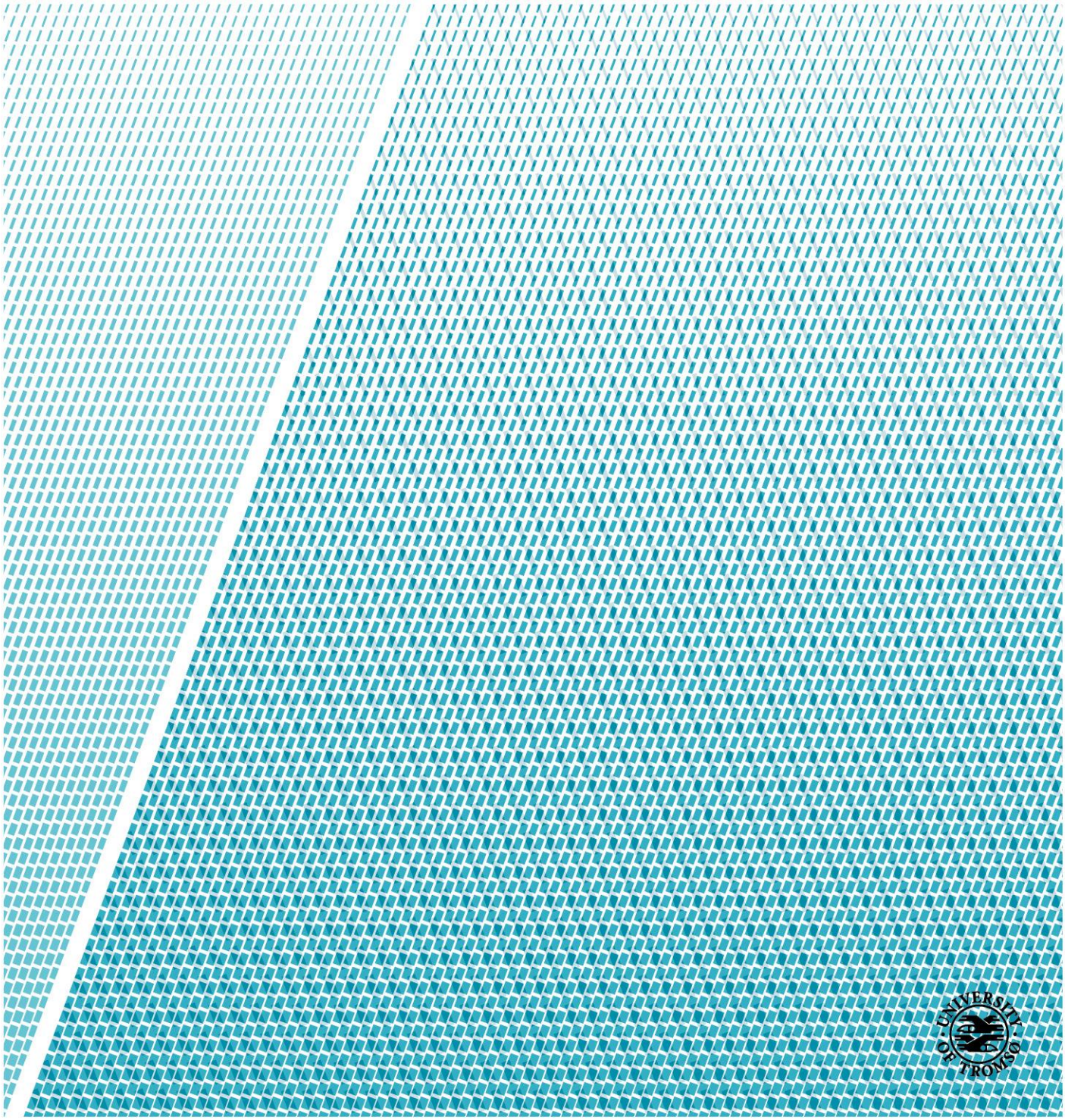
University of Tromsø - The Arctic University of Norway
Faculty of Engineering Science and Technology
Department of Computer Science and Computational Engineering

Master of Science

Computational modelling of perfusion to the heart

Henry Chukwuemeke Afambu

Master's Thesis in Engineering Design - Spring 2019



Supplementary report

for

Master Thesis

Computational modelling of perfusion to the heart

UiT / THE ARCTIC UNIVERSITY OF NORWAY

Henry Chukwuemeke Afambu

Contents

List of Figures

1 Introduction

This thesis supplement presents the translation of the Darcy flow problem (i.e. the governing equations, boundary conditions and parameters) into Python codes that implement the Darcy porous media flow formulation on a unit square porous medium. This supplement also discusses the finite element error analysis carried out to gauge the code's level of accuracy by use of method of manufactured solutions. Examining the numerical error in computation helps to verify that the code correctly solves the governing partial differential equation.

The results of the error analysis show that the Python code implements the Darcy flow model on a unit square porous media with acceptably minimal numerical error.

The error analysis is concluded with a mesh convergence test that shows that in running the Python code with a mesh size [78 x 78] on FEniCS, we obtain a reasonably accurate numerical solution.

Multiple runs of the Python code with an applied mesh size greater than [78 x 78] on FEniCS showed that the accuracy of the numerical solution did not improve significantly compared to the significant amount of computer resources, memory and solution time needed to run the Python code.

2 Darcy flow on a unit square porous medium

The governing equations [1] for Darcy flow whereby the Darcy velocity, $\boldsymbol{w}$, is linearly proportional to the gradient of the fluid pore pressure, p is:

$$\boldsymbol{w} + \mathbf{K} \cdot \nabla p = \mathbf{0} \quad in\ \Omega \tag{1}$$

$$\nabla \cdot \boldsymbol{w} = f \quad in\ \Omega \tag{2}$$

where $\mathbf{K}$ is the permeability of the porous medium Ω, and f is the fluid volumetric source or sink, Ω is the spatial domain of the porous medium, p represents the fluid pressure. ∇p is the gradient of pressure, p.

From equation (1), the Darcy velocity, $\boldsymbol{w}$ is given by:

$$\boldsymbol{w} = -\mathbf{K} \cdot \nabla p \tag{3}$$

Substituting for $\boldsymbol{w}$ into equation (2) gives:

$$\nabla \cdot (-\mathbf{K} \cdot \nabla p) = f \quad in\ \Omega \tag{4}$$

For our Darcy flow problem, we have:

The domain, $\Omega = [0,1] \times [0,1]$ (a 2-dimensional unit square media where $\Omega \subset \mathbb{R}^2$).

Let $\partial\Omega$ represent the boundary of domain Ω.

Assume that the permeability of the unit square porous media, $\mathbf{K}$ is spatially dependent on both x and y and given by:

$$\mathbf{K} = max\left(exp\left(-\left(\frac{\mathrm{y} - 0.5 - 0.1sin(10\mathrm{x})}{0.1}\right)^2\right), 0.1\right) \cdot I$$

(where I = identity matrix).

Also assume that porosity, ϕ of the medium $= 0.3$

From equation (4) above, we have source/sink term, f as:

$$f = \nabla \cdot (-\mathbf{K} \cdot \nabla p)$$

Assume that for the Darcy problem, the two given Dirichlet pressure boundary conditions apply:

$$p_D = \begin{cases} 1, & at\ x = 0 \\ 0, & at\ x = 1 \end{cases} \quad on\ \partial\Omega$$

In addition, the unit square medium has impermeable boundaries at $y = 0$ and $y = 1$. All other boundaries are permeable.

Deriving the variational form of governing equation

The variational form of the Darcy flow problem is needed for FEniCS to solve the governing Darcy equations (3) and (4).

The variational form is obtained by first multiplying both sides of the equation (4) by a test function, say v and integrating by parts over the domain, Ω:

$$\int_{\Omega} \nabla \cdot (-\mathbf{K} \cdot \nabla p) v \ \mathrm{d}x = \int_{\Omega} f v \ \mathrm{d}x \tag{5}$$

where $\mathrm{d}x$ is the differential element for integration over the domain Ω. The unknown pressure, p to be approximated is referred to as the trial function in FEniCS.

This gives:

$$\int_{\Omega} (\mathbf{K} \cdot \nabla p) \cdot \nabla v \ \mathrm{d}x \ - \ \int_{\partial\Omega} \mathbf{K} \nabla p \cdot n \, v \, \mathrm{d}s \ = \int_{\Omega} f v \ \mathrm{d}x \tag{6}$$

where ds denotes the differential element for integration over the boundary of Ω (i.e. $\partial\Omega$).

As a result of the Dirichlet boundary condition, the pressure, p is known at the boundary and the test function, v vanishes on the domain's boundary $\partial\Omega$ so:

$$\int_{\partial\Omega} \mathbf{K} \nabla p \cdot n \, v \, \mathrm{d}s = 0$$

Therefore, equation (6) becomes:

$$\int_{\Omega} (\mathbf{K} \cdot \nabla p) \cdot \nabla v \ \mathrm{d}x = \int_{\Omega} f v \ \mathrm{d}x \tag{7}$$

The equation (7) above is known as the weak or variational form of the original system of Darcy's flow equations (1) and (2).

If we require that the above equation holds for all test functions v in some suitable space $\hat{\mathrm{V}}$, called the test space, we obtain a well-defined mathematical problem that uniquely determines the pressure solution, p which lies in some function space V, called the trial space [2].

The variational problem then reads: Find $p \in \mathrm{V}$ such that:

$$\int_{\Omega} (\mathbf{K} \cdot \nabla p) \cdot \nabla v \ \mathrm{d}x = \int_{\Omega} f v \ \mathrm{d}x \qquad \forall v \in \hat{\mathrm{V}} \tag{8}$$

The trial and test spaces are defined as:

$$\mathrm{V} = \{v \ \in \ H^1(\Omega) : v = p_D \ on \ \partial\Omega\},$$

$$\hat{\mathrm{V}} = \{v \ \in \ H^1(\Omega) : v = 0 \ on \ \partial\Omega\}.$$

where $H^1(\Omega)$ is the Sobolev space containing functions v such that v^2 and $|\nabla v|^2$ have finite integrals over domain Ω (which essentially means the functions are continuous).

To obtain the numerical approximation [2] between the mathematical formulation of the above problem and the corresponding FEniCS program, we let p represent the numerical solution of pressure for the discrete problem.

In addition, we let p_e represent the exact pressure solution of the continuous problem. Similarly, we let V denote the discrete finite element function space in which we obtain the numerical solution.

Writing the canonical notation for variational problems as follows: find $p \in V$ such that:

$$a(p, v) = L(v) \qquad \forall v \in \hat{\mathrm{V}}$$

where $a(p, v)$ is known as the bilinear form and $L(v)$ is the linear form of the variational problem.

For the Darcy flow variational form (8), we have:

$$a(p, v) = \int_{\Omega} (\mathbf{K} \cdot \nabla p) \cdot \nabla v \ \mathrm{d}x$$

$$L(v) = \int_{\Omega} f v \ \mathrm{d}x$$

The linear and bilinear forms of the variational problem will be entered into the Python code and run on FEniCS to obtain the numerical solution for pressure. The translation of the Darcy flow problem (i.e. governing equations, boundary conditions and parameters) into Python codes is presented in the next section.

3 The Python Code

Now that the PDE (8) is formulated as a variational problem, we proceed to writing the Python code to solve it in the FEniCS program. This involves:

- Choosing the finite element spaces V and $\hat{V}$ by specifying the domain (the mesh) and the type of function space (polynomial degree and type).
- Expressing and writing PDE (8) as a (discrete) variational problem in FEniCS: find $p \in V$ such that $a(p, v) = L(v)$ for all $v \in \hat{V}$.
- Verifying that the code correctly solves the governing mathematical equation numerically by use of a procedure known as the method of manufactured solutions (MMS).

MMS is a code verification technique that involves using an expression to construct a test problem with a known analytical solution and running the Python code to solve it on FEniCS. The accuracy of the code is then gauged by running the problem on systematically refined grids (mesh). The behavior of the error is also examined against the theoretical order of accuracy inherent in the code's numerical discretization.

Writing the Python Code

The code is written in Python programming language.

The first lines of the code (shown below) are written to import key classes, scientific packages and tools required for the computation of the partial differential equation (8):

```
1.  from fenics import *
2.  import numpy as np
3.  from ufl import nabla_div
4.  import sympy as sym
5.  import matplotlib.pyplot as plt
```

In the next part of the code, a solver function (named `solver`) was created to solve the partial differential equation (8) and produce the numerical solution of pressure, p across the domain, Ω.

Using the solver function [2] has the advantage of enabling us solve a large number of problems where we can vary certain aspects of the problem as we want such as the domain characteristics, boundary conditions and coefficients such as permeability parameters.

The solver is defined and parameterized by the following arguments: the source term (f), the equation for the exact pressure (p_e) inside the domain, porosity of the medium, (phi), the permeability $(\mathbf{K})$, the mesh and the degree of finite element. The script for building `solver` is shown below:

```
6.  def solver(f, p_e, phi, K_val, mesh, degree):
```

Next step is defining the finite element function space (named `V`) for the domain. Since we know pressure is a scalar quantity, `V` is set as a scalar valued function space. The function space `V` is dependent on the 3 arguments – mesh, element type (denoted `P` below) and the degree.

```
7.      V = FunctionSpace(mesh, 'P', degree)
```

The element type `P` implies the Lagrange finite element which if combined with a degree 1 for our 2D problem means we have piecewise linear finite element represented as triangles with nodes at the three vertices [2]. As a result, the computed pressure, p will be continuous across elements and linearly varying in x and y inside each element.

The next step involves applying the two Dirichlet boundary conditions below:

$$p_D = \begin{cases} 1, & at\ x = 0 \\ 0, & at\ x = 1 \end{cases} \qquad on\ \partial\Omega$$

Using the FEniCS class `DirichletBC`, the two Dirichlet boundary conditions are defined and collated into `bcs` as shown below:

```
8.      p_L = Constant(1.0)
9.      def boundary_L(x, on_boundary):
10.         return on_boundary and near(x[0], 0)
11.     bc_L = DirichletBC(V, p_L, boundary_L)
12.
13.     p_R = Constant(0.0)
14.     def boundary_R(x, on_boundary):
15.         return on_boundary and near(x[0], 1)
16.     bc_R = DirichletBC(V, p_R, boundary_R)
17.
18.     bcs = [bc_L, bc_R]
```

The next step is to define the variational problem as seen below:

```
19.     p = TrialFunction(V)
20.     v = TestFunction(V)
21.     K = K_val
22.     a = dot(K*grad(p), grad(v))*dx
23.     L = inner(f, v)*dx
```

Including the solve function in `solver` is shown below:

```
22.     p = Function(V)
23.     solve(a == L, p, bcs)
24.     return p
```

The `run_solver` function is defined below to enable us call on `solver` to evaluate the finite element variational problem after entering all parameters specific to the Darcy flow problem.

Since the domain is a 2D unit square, the mesh is specified using the `UnitSquareMesh` function and an arbitrary mesh resolution (50 x 50) is applied.

```
25. def run_solver():
26.     "Run solver to compute and post-process solution"
27.
28.     mesh = UnitSquareMesh(50, 50)
```

Next, the parameters specific to our Darcy flow problem is entered as shown in the code below:

```
29.     phi = Constant(0.3)
30.     d = 2
31.     I = Identity(d)
32.     M = Expression('fmax(0.10, exp(-pow(10.0*x[1]-1.0*sin(10.0*x[0])-5.0,
        2)))', degree=2, domain=mesh)
33.     K_val = M*I
34.     K = K_val
```

Using the method of manufactured solutions (MMS), a simple expression for the exact pressure, p_e (inside the porous media domain, Ω) is arbitrarily manufactured as the solution to our test problem for the system of Darcy flow equations (1) and (2). The manufactured expression for the exact pressure, p_e is to be the analytical solution which we use to verify that our numerically computed solution to the test problem is valid.

The method of manufactured solutions

Assuming the system of Darcy flow equations (1) and (2) is a test problem where the manufactured solution expression for exact pressure, p_e inside the porous media, Ω is given by:

$$p_e(x, y) = 1 - x^2 \qquad in\ \Omega \tag{9}$$

The gradient of exact pressure, p_e (i.e. ∇p_e) is given by:

$$\nabla p_e = \nabla(1 - x^2)$$

$$\nabla p_e = -2x + 0y \tag{10}$$

Both p_e and ∇p_e are entered into the python code using the `Expression` function as shown:

```
35.     p_e = Expression('1 - x[0]*x[0]', degree=2)
36.
37.     grad_p_e = Expression(('-2*x[0]', '0.0'), degree=1, domain=mesh)
```

FEniCS requires that the expression above be written in C++ syntax. The second argument `degree` specifies how the expression is to be treated in the numerical computation. For this problem, the degree in the expression for `p_e` is set at 2 since p_e is quadratic, while the degree is set to 1 in the expression for `grad_p_e` since ∇p_e is linear.

Recalling from equation (4):

$$f = \nabla \cdot (-\mathbf{K} \cdot \nabla p) \quad in\ \Omega$$

Substituting for ∇p from equation (10) into equation (9) above gives:

$$\nabla \cdot \big(-\mathbf{K} \cdot (2x + 0y)\big) = f \tag{11}$$

We have the source/sink term, f entered into the python code as shown below:

```
38.     f = nabla_div(dot(-K, grad_p_e))
```

The `solver` is called on with the appropriate arguments to numerically compute pressure, p:

```
39.     p = solver(f, p_e, phi, K, mesh, 1)
```

Observe that the degree of the finite element is set at 1 which combined with element type `'P'` from line 7 of the Python code implies utilizing the Lagrange finite element so we have piecewise linear finite element represented as triangles with nodes at the three vertices [2].

Since the exact solution of our 2D problem is quadratic and we have a linear piecewise finite element, the error of the numerical computation will be non-zero [2]. In the error analysis, this non-zero error is analyzed and it's behavior is examined against the theoretical order of accuracy inherent in the code's numerical discretization.

If the computational error is sufficiently small (with a finite element degree 1) and shows the expected convergence rate, we accept that the code numerically solves and implements the governing system of Darcy flow equations.

Setting the degree at 2 is also acceptable. The computational error is observed to be even smaller and approaches machine precision ($\times 10^{-16}$). The reason we avoid setting the degree at 2 at this stage is that analyzing the error's behavior becomes problematic as round-off errors accumulate with increased mesh resolution. For this reason, we set the finite element degree to 1.

Recalling equation (3):

$$\boldsymbol{w} = -\mathbf{K} \cdot \nabla p \quad in\ \Omega$$

The Darcy velocity $\boldsymbol{w}$ is numerically computed with the python code as shown below:

```
40.     w_1 = -K*grad(p)
```

Being a vector quantity, we project the numerically computed velocity onto a vector valued function space with the same arguments as that of pressure to obtain the Darcy velocity profile. The python code for this is as shown below:

```
41.     w = project(w_1, VectorFunctionSpace(mesh, 'P', degree=1))
```

To account for the fact that only a fraction of the total media volume is available for flow, the fluid velocity, uf is given by the equation:

$$uf = \frac{Darcy\ velocity, \boldsymbol{w}}{porosity,\ \phi}$$

The fluid velocity is numerically computed and projected onto a vector valued function space as shown below:

```
42.     uf_1 = w_1/phi
43.     uf = project(uf_1, VectorFunctionSpace(mesh, 'P', degree=1))
```

The numerically computed solutions (pressure, Darcy and fluid velocity) are written and saved for visualization as shown below:

```
44.     xdmf = XDMFFile('Pressure_Gradient.xdmf')
45.     p.rename('Pressure', '')
46.     xdmf.write(p)
47.     xdmf.close()
48.
49.     xdmf = XDMFFile('Darcy_Velocity_Profile.xdmf')
50.     w.rename('Darcy_velocity', '')
51.     xdmf.write(w)
52.     xdmf.close()
53.
54.     xdmf = XDMFFile('Fluid_Velocity_Profile.xdmf')
55.     Uf.rename('Fluid_velocity', '')
56.     xdmf.write(uf)
57.     xdmf.close()
```

The program is completed with the python code below:

```
55. if __name__ == '__main__':
56.     run_solver()
```

The python code on line (55) above ensures that the file can be imported (and reused) in other programs so all statements in the main program that are not inside functions should appear within a test `if __name__ == '__main__':`. This test is true if the file is executed as a program, but false if the file is imported. The script on line (56) enables the code to run `solver` and completes the code.

For the full code, please see the online repository (Appendix 1).

4 Results

Using Paraview v5.6, the numerical solution for pressure p, the Darcy velocity, $\boldsymbol{w}$ and fluid velocity, uf are shown below:

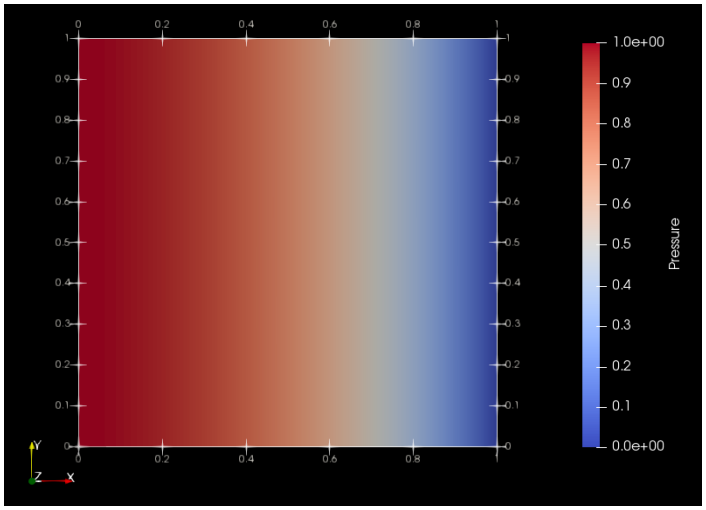


Figure 1: Pressure gradient across the unit square medium

Referencing the figure 1 above, we see that the pressure boundary conditions are enforced so there exists pressure, $p = 1$ at $x = 0$ (left) and zero pressure, p at $x = 1$ (right). The pressure inside the medium varies according to the solution for pressure, $p = 1 - x^2$. As expected, the pressure p does not vary with vertical distance, y.

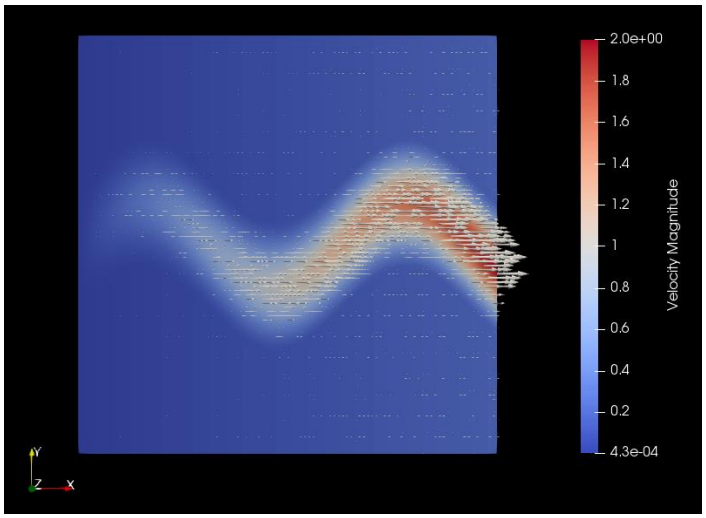


Figure 2: Darcy Velocity profile across the unit square medium

From the figure 2 above, the arrows (glyph) show that the direction of fluid flow going from left ($x = 0$) to right ($x = 1$). This is consistent with the boundary conditions and solution for pressure as the fluid is expected to move from the region of higher pressure to the region lower pressure.

As a result of the higher pressure differential between $x = 0.6$ and $x = 0.8$ of the unit square medium, we observe that the velocity is higher in that region compared to the rest of the medium.

5 Finite Element Error Analysis

Since FEniCS solves the Darcy governing equations by approximating the continuous solution space using discrete elements, there are errors present in the numerical computation. Quantifying the numerical error in finite element computations is necessary to gauge the Python code's level of accuracy. Examining the error helps to verify that the code solves the governing partial differential equation.

The error analysis requires comparing the exact solution for pressure, p_e with the numerically computed pressure, p.

The L_2 Error norm, defined as:

$$E = \sqrt{\int_\Omega (p - p_e)^2 \, dx} \tag{12}$$

is computed on FEniCS for systematically refined mesh (grids) number.

A Python code is presented below as a supplement to the Python code earlier presented in section 3. This new Python code runs `solver` for different mesh sizes and computes the respective L_2 error for each mesh/grid size. This enables us observe the error's behavior against the theoretical order of accuracy inherent in the code's numerical discretization.

Writing the supplementary Python script using the `iterate_solver` function enables computing and storing the L_2 error, E that exists between the exact pressure, p_e and the numerically computed pressure, p for every given mesh (grid) number. The `iterate_solver` function is defined below to collate the respective cell size, h and the degree of freedom, DOF.

In addition, the function `iterate_solver` is defined to collate two additional parameters which will be used for the mesh convergence test discussed in section 5.3:

- GPS: The integral of the square of the gradient of numerical pressure over the domain.

$$GPS = \int_\Omega (\nabla p)^2$$

- GPES: The integral of the square of the gradient of exact pressure over the domain.

$$GPES = \int_\Omega (\nabla p_e)^2$$

```
57.     def iterate_solver():
58.     p_e = Expression('1 - x[0]*x[0]', degree=2)
59.     E = []
60.     h = []
61.     DOF = []
62.     GPS = []
63.     GPES = []
```

In the code below, the mesh (grid) number is set to increase systematically by 4 from a coarse mesh size $[4 \times 4]$ to a very fine mesh size $[380 \times 380]$. In addition, the finite element function space for the domain (named $\mathbb{V}$) is set as seen below:

```
64.     for m in range (4, 380, 4):
65.         mesh = UnitSquareMesh(m, m)
66.         V = FunctionSpace(mesh, 'P', 1)
```

The finite element pressure, p is computed using the piecewise linear finite element for each mesh/grid size and the respective parameters E, h, DOF and GPS are collected.

Since the solution for exact pressure p_e is quadratic, it is interpolated onto a quadratic finite element with degree set at 2 as seen below:

```
67.         p_e_f = interpolate(p_e, FunctionSpace(mesh, 'P', 2))
```

This ensures that the pressure function, p_e is as exact as possible and is used as benchmark for the numerically computed pressure, p.

Next, the problem specific parameters are added and `solver` is called up to compute as seen in the code below. Note that the finite element degree is intentionally set to 1 on line (77) so we obtain a non-zero error that can be studied.

```
68.         phi = Constant(0.3)
69.         d = 2
70.         I = Identity(d)
71.         M = Expression('fmax(0.10, exp(-pow(10.0*x[1]-1.0*sin(10.0*x[0])-
            5.0, 2)))', degree=2, domain=mesh)
72.         K = M*I
73.
74.         grad_p_e = Expression(('-2*x[0]', '0.0'), degree=1, domain=mesh)
75.         f = nabla_div(dot(-K, grad_p_e))
76.
77.         p = solver(f, p_e, phi, K, mesh, degree=1)
```

As the mesh (grid) is systematically increased (refined), the corresponding numerical error in finite element pressure, p is computed for error analysis.

Computing the L_2 error norm is made possible using the code below:

```
78.         E1 = errornorm(p_e_f, p, 'L2')
```

5.1 Error behavior as a function of cell size

Since the domain is a unit square, $[0,1] \times [0,1]$, consider figure 3 below to be a simple representation of the domain:

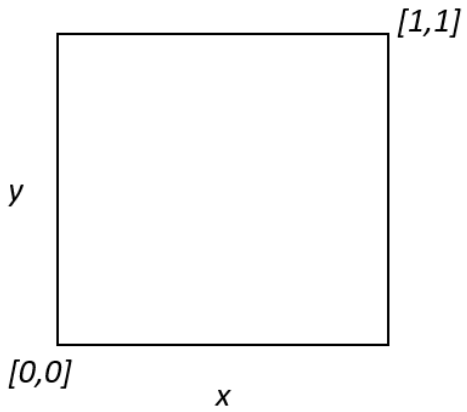


Figure 3: Simple representation of unit square medium

If a straight line is drawn across the unit square domain above, we evaluate its length by use of Pythagoras theorem:

$$Length\ of\ straight\ line = \sqrt{1^2 + 1^2}$$

$$Length\ of\ straight\ line = \sqrt{2}$$

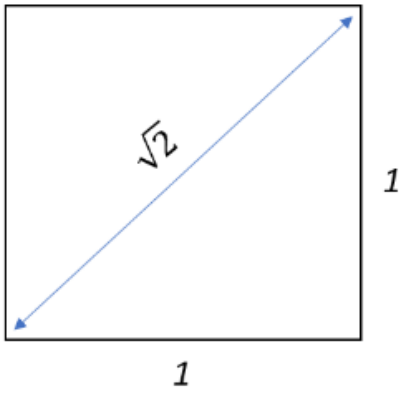


Figure 4: Another representation of unit square medium

If we assume the unit square in figure 4 above to be a unit cell, the length of the blue straight line across the cell is directly proportional to the size of the cell in that as the unit square medium is discretized n times (in the x and y axes) into equal cell sizes h_i (i.e. squares), the blue straight line across the cell is also discretized accordingly.

The size of each finite cell element, h_i is therefore, given by the relationship:

$$h_i = \frac{\sqrt{2}}{n} \tag{19}$$

where n = number of mesh/grid in the x and y axes.

Given the known relationship [3] between error for each mesh (grid), E_i and the corresponding finite element cell size, h_i:

$$E_i\ \alpha\ {h_i}^r \tag{20}$$

where r is the rate of convergence.

The equation (20) can be written as:

$$E_i = Ch_i^{\,r} \tag{21}$$

where C is an unknown constant.

Taking natural log of both sides give:

$$Log_e(E_i) = rLog_e(h_i) + Log_e(C) \tag{22}$$

The above equation (22) can be compared to the equation of a straight line below:

$$\mathrm{y} = m\mathrm{x} + c$$

where

- $Log_e(E_i)$ corresponds to the y-axis of the straight line plot.
- $Log_e(h_i)$ corresponds to the x-axis of the straight line plot.
- r corresponds to the gradient m of the straight line.

For L_2 error, and as the mesh is refined, the convergence rate, r value should approach:

$$r \approx finite\ element\ degree + 1 \tag{23}$$

Since the finite element degree is set at 1 for the numerical computation for pressure, p, the expected error rate of convergence, r should approach 2 as the mesh is refined.

Plotting $Log_e(E_i)$ against $Log_e(h)$ produced the straight line below:

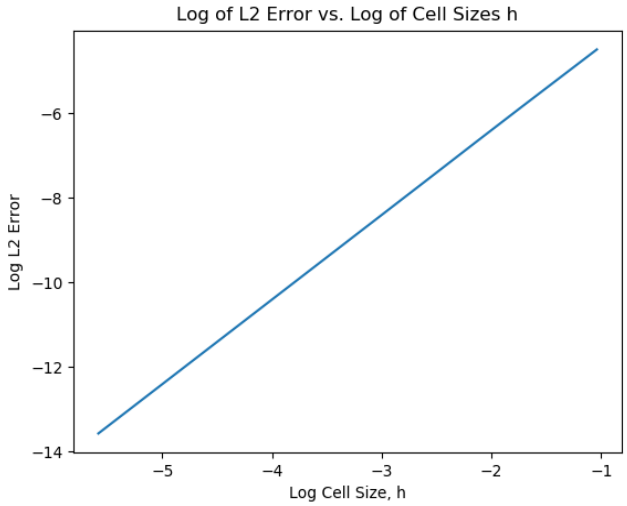


Figure 5: Log L2 Error against Log cell size

The gradient of the above straight line in figure 5 is 1.998.

Since the convergence rate approaches 2, we have a good indication that the Python code implements the governing Darcy flow equations.

5.2 Error behavior as a function of degree of freedom

Plotting the L_2 Error (on a logarithmic scale) against degree of freedom (DOF) produces:

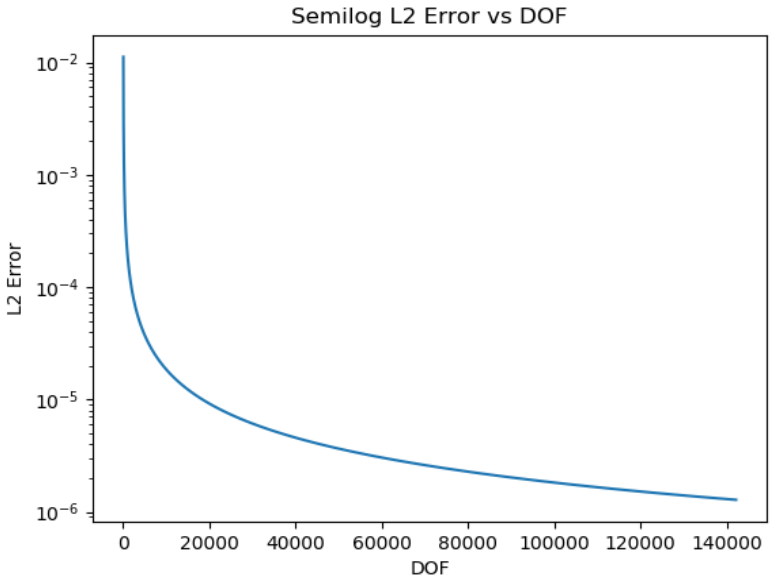


Figure 6: Semi-log of L2 error against DOF

The figure 6 shows that as more degrees of freedom are added to the discretization by refining the mesh/grid, the L_2 error decreases significantly up to a certain point where adding more degrees of freedom yield progressively smaller reductions in L_2 error.

Plotting the $Log_e(E_i)$ against $Log_e(DOF_i)$ produced the plot below:

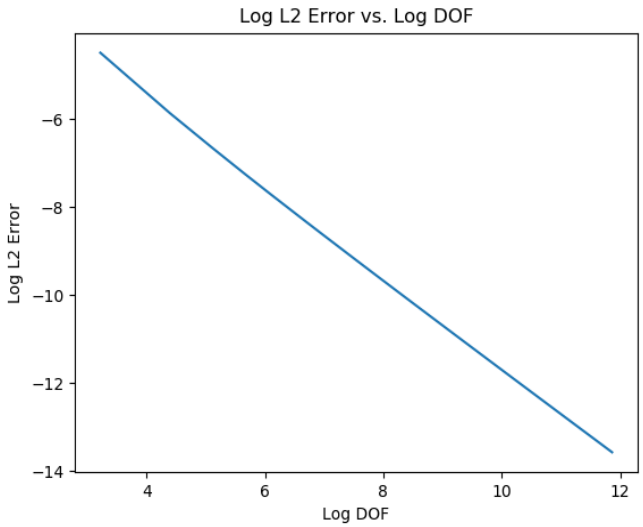


Figure 7: Log L2 Error against Log DOF

The figure 7 above implies that the simulation converges as more degrees of freedom are added to the discretization by mesh refinement (increasing number of mesh/grid).

5.3 Mesh Convergence Test

Using a too high mesh/grid number is computationally expensive to solve and requires significant computer memory and solution time. A mesh/grid convergence study is important to determine

the optimal point where a reasonably accurate numerical solution is obtained at the expense of using the least computational power, time and memory resources.

Let us assume that for our domain Ω, we do not know what the exact pressure p_e and its gradient ∇p_e are:

$$p_e = Unknown \quad \text{and} \quad \nabla p_e = Unknown$$

In addition, assume we do not know what the numerically computed pressure p and its gradient ∇p are:

$$p = Unknown \quad \text{and} \quad \nabla p = Unknown$$

It is mathematically valid that as numerically computed pressure, p tends to the exact pressure, p_e, then the equation below holds true:

$$\int_{\Omega} (\nabla p)^2 - \int_{\Omega} (\nabla p_e)^2 = 0 \tag{24}$$

By extension, the equation below also holds true:

$$Log_e \int_{\Omega} (\nabla p)^2 - Log_e \int_{\Omega} (\nabla p_e)^2 = 0 \tag{25}$$

Using the `assemble` function, both $\int_{\Omega}(\nabla p)^2$ and $\int_{\Omega}(\nabla p_e)^2$ in equation (24) above are computed:

```
80.         IGPS = assemble(inner(grad(p), grad(p))*dx)
81.         GPS.append(IGPS)
82.         IGPES = assemble(inner(grad(p_e_f), grad(p_e_f))*dx)
83.         GPES.append(IGPES)
```

Plotting $Log_e(\int_{\Omega}(\nabla p)^2)$ against -$Log_e(h)$ gives the figure below:

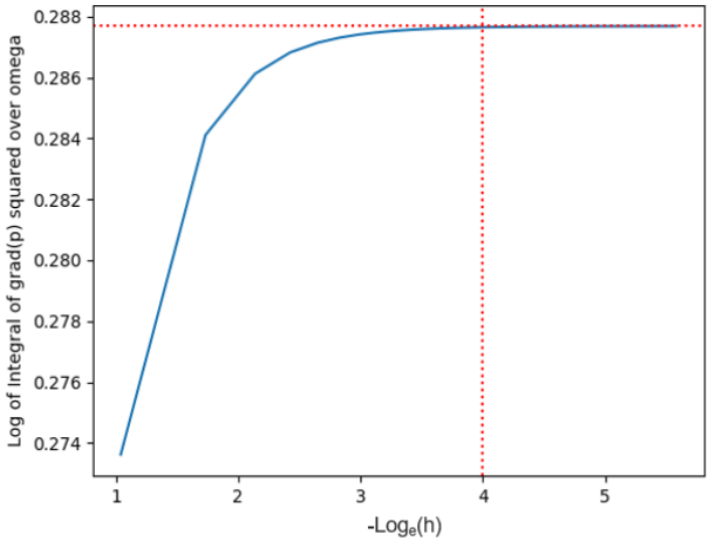


Figure 8: Log GPS against -Log cell size

Observe from the figure 8 above that as finite element cell size decreases due to increase in number of mesh/grid (going from left to right on horizontal axis), the $Log_e(\int_{\Omega}(\nabla p)^2)$ value converges up to a certain point and remains constant (marked by the horizontal red dotted line) as $-Log_e h$ increases.

The plot above implies that from about point $-Log_e h = 4$ (marked by the vertical red dotted line), subsequent increase in the number of meshes/grids (mesh refinement), does not yield significant change in the value of $Log_e(\int_\Omega (\nabla p)^2)$.

Plotting $Log_e(\int_\Omega (\nabla p_e)^2) - Log_e(\int_\Omega (\nabla p)^2)$ against -$Log_e(h)$ gives the figure below:

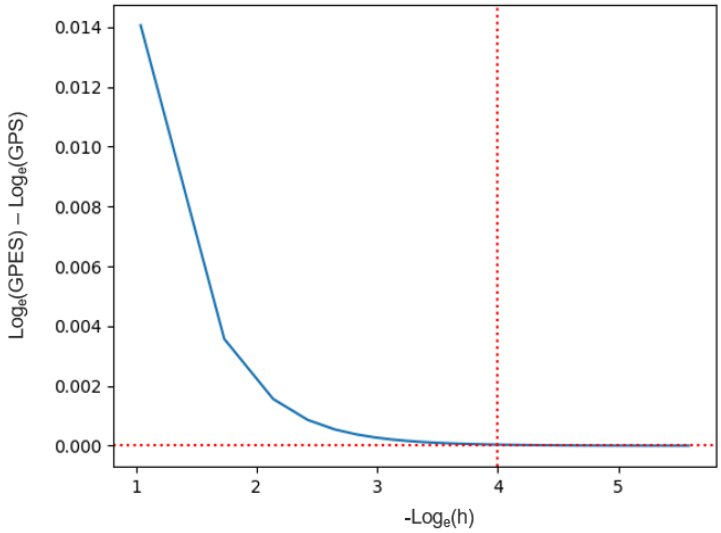


Figure 9: Log (GPES-GPS) against Log cell size

Observe from figure 9 that as the mesh is refined (going from left to right), the difference between $Log_e \int_\Omega (\nabla p_e)^2$ and $Log_e \int_\Omega (\nabla p)^2$ drops to a very small value close to 0 where it remains constant (marked by the horizontal red dotted line) as $-Log_e h$ increases.

As expected, this occurs at the same point $-Log_e h = 4$ (marked by the vertical red dotted line). Beyond this point, subsequent increase in the number of mesh/grid (mesh refinement), does not yield significant change in the $Log_e \int_\Omega (\nabla p_e)^2 - Log_e \int_\Omega (\nabla p)^2$.

Given from the above plot:

$$-Log_e h = 4$$

Taking exponents of both sides give:

$$h = e^{-4}$$

$$h = 0.01831564$$

Using equation (19), we have:

$$h = 0.01831564 = \frac{\sqrt{2}}{n}$$

Solving for n gives:

$$n = \frac{\sqrt{2}}{0.01831564} \approx 77.2$$

Rounding n to the nearest whole number not less than 77.2 gives:

$$n = 78$$

This implies that using a mesh/grid beyond $[78 \times 78]$ does not significantly improve the accuracy of the numerical computation.

6 Testing Python code with different boundary conditions

To reinforce the Python code's ability to solve the governing Darcy flow system of equations for a unit square, a different boundary condition and manufactured solution is employed as follows:

Boundary Condition

Assume that the pressure, p at the boundary of the 2D unit square is known and defined by the set of Dirichlet boundary conditions such that pressure $p = p_D$ where pressure, p_D is given by the expression:

$$p_D = \sin(x)\sin(y) \qquad on\ \partial\Omega$$

Manufactured Solution

Assuming our system of Darcy flow equations (1) and (2) is a test problem where the manufactured solution expression for exact pressure, p_e inside the porous media, Ω is given by:

$$p_e(x,y) = \sin(x)\sin(y) \qquad in\ \Omega \tag{26}$$

The gradient of pressure, p (that is ∇p) is given by:

$$\nabla p_e = \nabla(\sin(x)\sin(y))$$

$$\nabla p_e = \cos(x)\sin(y) + \sin(x)\cos(y) \tag{27}$$

The boundary conditions and manufactured solution are translated into Python codes in the same way as described in section 2.

For the full code, please see the online repository (Appendix 2).

The Python code in Appendix 2 is essentially the same as that of Appendix 1 with the only difference being the specified pressure boundary condition, the manufactured solution and medium porosity parameter.

Using Paraview v5.6, the numerical solutions are visualized below:

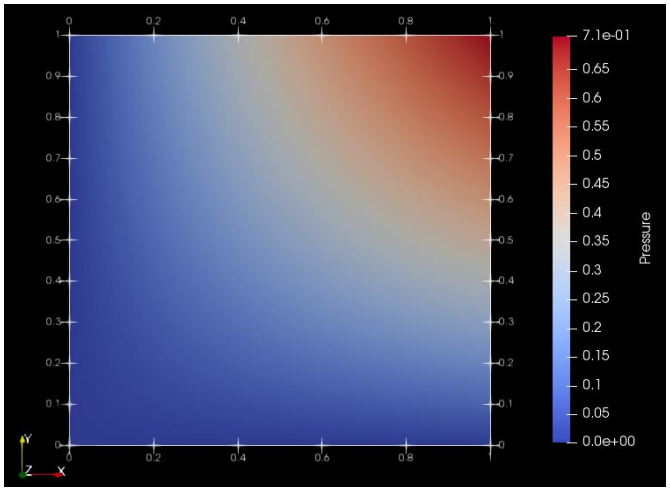


Figure 10: Pressure Gradient across Unit Square Medium

We see from figure 10 that as a result of the specified boundary pressure, the highest pressure in the domain occurs at the top right region of the unit square medium ($x = 1$ and $y = 1$).

The pressure gradient is consistent with the solution for pressure, i. e. $p = \sin(x)\sin(y)$.

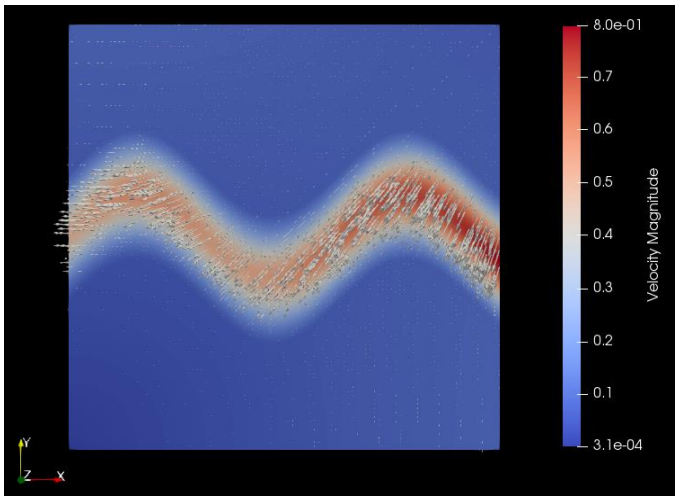


Figure 11: Darcy Velocity Profile across Unit Square Medium

From the velocity profile figure 11 above, observe that the direction of flow is from the region of highest pressure ($x = 1$ and $y = 1$) to the regions of lower pressure within the domain.

6.1 Error Analysis

The numerical error in finite element computations is again quantified to gauge the python code's level of accuracy for the applied boundary condition and manufactured solution. In addition, the error's behavior is examined to verify that the code solved the governing partial differential equations for Darcy flow.

Error Behavior as a function of finite element size

As the mesh/grid is systematically increased (mesh refinement), the corresponding numerical error in finite element pressure, p are computed for analysis. Plotting $Log_e(E_i)$ against $Log_e(h)$ produces the straight line plot below:

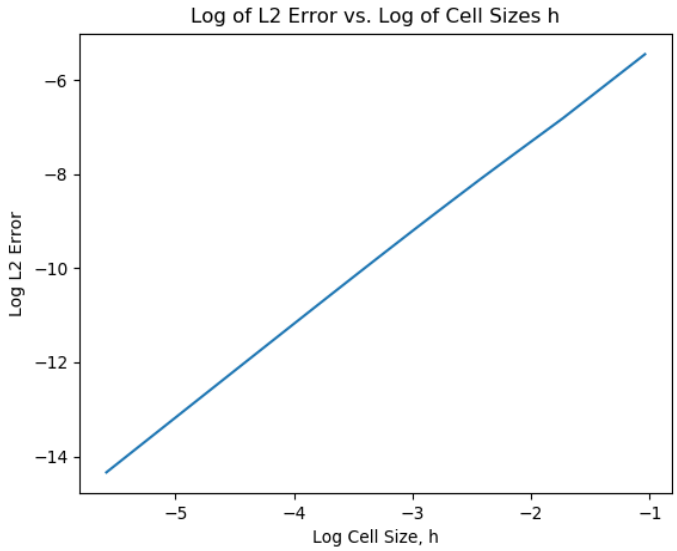


Figure 12: Log L2 Error against Log cell size

The gradient of the straight line in figure 12 is observed to be $\mathbf{1.97}$ and approaches 2 as the mesh is further refined. This is a positive indication that the Python code implements our Darcy problem.

Error Behavior as a function of Degree of Freedom

Plotting the L_2 Error (on a logarithmic scale) against Degree of Freedom (DOF) produces the plot:

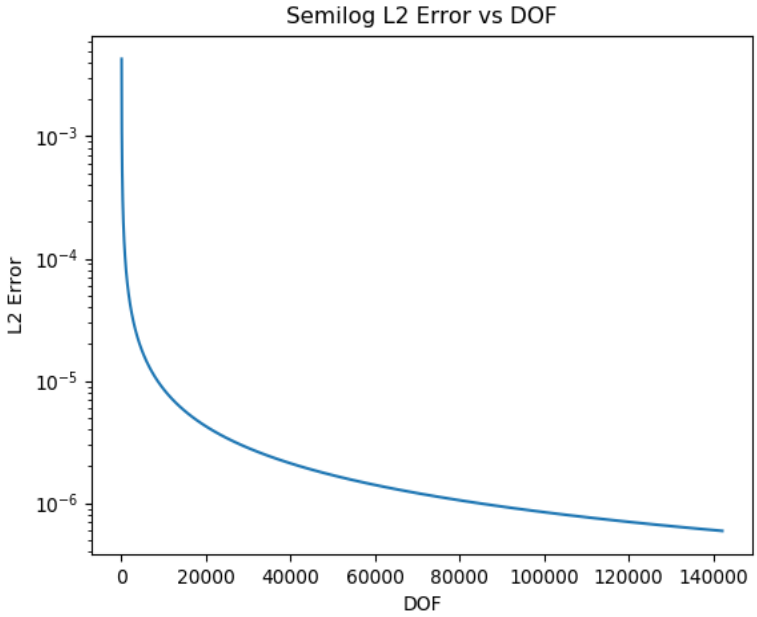


Figure 13: Semi-log of L2 error against DOF

The figure 13 shows that as more degrees of freedom are added to the discretization by mesh refinement, the L_2 error decreases significantly to an extent where adding more degrees of freedom yields progressively smaller reductions in L_2 error.

Plotting the $Log_e(E_i)$ against $Log_e(DOF_i)$ produced the plot below:

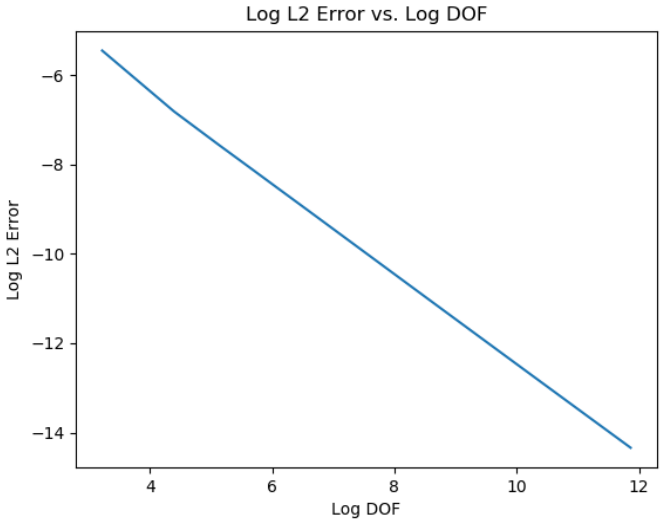


Figure 14: Log L2 Error against Log DOF

The figure 14 implies that the simulation converges as more degrees of freedom are added to the discretization process by mesh refinement (i.e. increasing number of mesh/grid).

Mesh Convergence Test

A mesh convergence study is again carried out to determine the optimal point at which a reasonably accurate numerical solution can be obtained at the expense of using the least computational power, time and memory resources.

Plotting $Log_e(\int_\Omega(\nabla p)^2)$ against -$Log_e(h)$ gave:

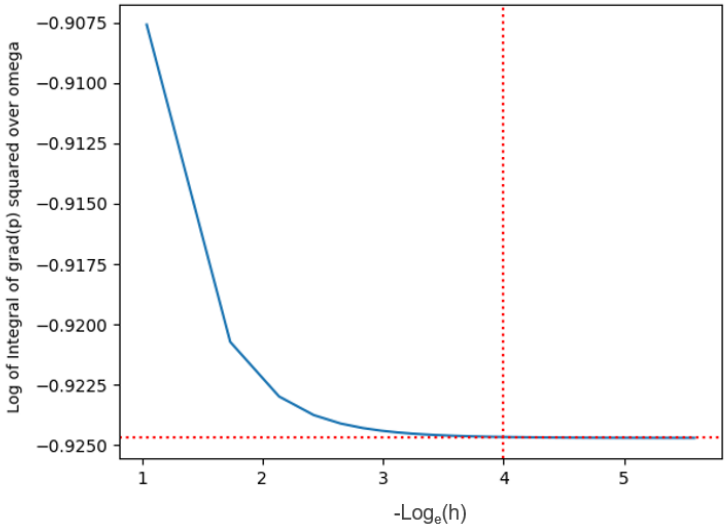


Figure 15: Log GPS against Log cell size

Observe from the above figure 15 that as finite element cell size decreases due to mesh refinement (going from left to right on horizontal axis), $Log_e(\int_\Omega(\nabla p)^2)$ converges up to a certain value where it remains constant (marked by the horizontal red dotted line) as $-Log_e h$ increases.

The plot above shows that from about the point $-Log_e h = 4$ (marked by the vertical red dotted line), subsequent increase in the number of mesh (grids or mesh refinement), does not yield significant changes in the value of $Log_e \int_\Omega(\nabla p)^2$.

Plotting $Log_e \int_\Omega(\nabla p_e)^2 - Log_e \int_\Omega(\nabla p)^2$ against -$Log_e\ (h)$ gives the figure below:

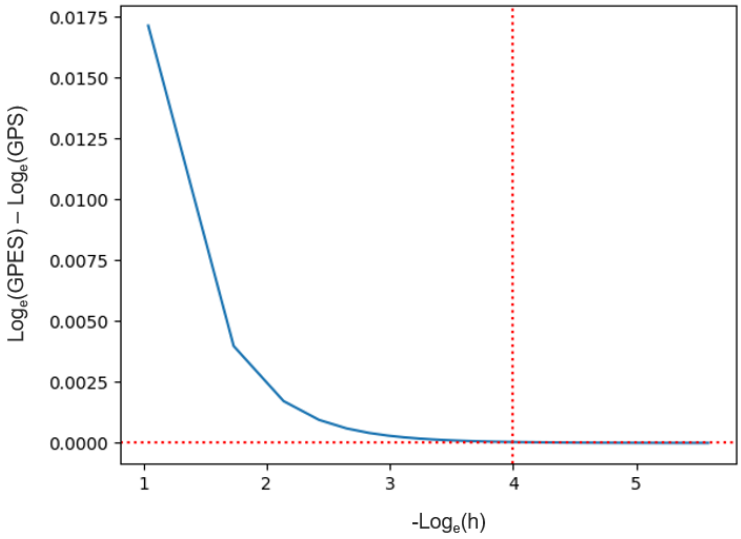


Figure 16: Log (GPES-GPS) against Log cell size

Observe from figure 16 that as the mesh is refined (going from left to right), the difference between $Log_e \int_\Omega (\nabla p_e)^2$ and $Log_e \int_\Omega (\nabla p)^2$ drops to a small value close to 0 where it remains constant (marked by the horizontal red dotted line) as $-Log_e h$ increases.

As expected, this occurs at the same point $-Log_e h = 4$ (marked by the vertical red dotted line) and the mesh/grid size number, n is computed to obtain the value 78.

Following multiple runs, it is observed that when running the Python code with a mesh size above [78 x 78] on FEniCS, the accuracy of the numerical solution does not improve significantly compared to the disproportionately significant amount of computer resources and solution time needed to run the Python code.

7 Conclusion

It is established that the Python code implements the Darcy model on a unit square porous medium with an acceptably minimal numerical error.

The error analysis was concluded with a mesh convergence test that showed that the applying the mesh size [78 x 78] on the Python code, gave a reasonably accurate numerical solution on FEniCS without using too much computer power, solution time and memory resources.

When running the Python code on FEniCS using a mesh size above [78 x 78], it is observed that the accuracy of the numerical solution did not improve significantly compared to the significant amount of computer resources and solution time needed to run the Python code on FEniCS.

This model is extended to simulate Darcy's flow on 3 porous media compartments that exists in the same spatial domain (the multi-compartment Darcy porous flow model).

Based on the results of the mesh convergence test, the mesh/grid number [78 x 78] is applied in the multi-compartment Darcy porous media flow model.

8 References

1. J. Lee, A. Cookson, R. Chabiniok, S. Rivolo, E. Hyde, M. Sinclair, C. Michler, T. Sochi, N. Smith.: Multiscale Modelling of Cardiac Perfusion - Modeling the Heart and Circulatory System, 2015.
2. P. L. Hans, L. Anders: Solving PDEs in Python, The FEniCS Tutorial, Simula, 2017.
3. K. S. Surana, A. D. Joy, J. N. Reddy: Error Estimations, Error Computations and Convergence Rates in FEM for BVPs, 2016.