Internship Report: Scimba Feel++ Wrapper
MASTER 1, CSMI

Rayen Tlili

August 21, 2024

Contents

Context

This document reports the details of the coupling of Feel++, which uses Galerkin methods for PDE solving, and ScimBa, which uses machine learning methods.

This project builds upon previous work and aims to refine and integrate the tools developed in the previous project to enhance the efficiency and accuracy of PDE solving through the combination of machine learning and traditional methods.

This project was conducted under the supervision of Joubine Aghili and Christophe Prud'homme.

Main Content

1 Introduction

The end product of this work is a program that will solve specific Poisson Partial Differential Equations with user-provided parameters. By combining the resources of ScimBa, a python solver using PINNs machine learning methods and Feel++ which uses Galerkin solving methods by integrating over a spatial domain. Which would then export, visualize and compare the results.
Here are the objectives, approach, and roadmap for the wrapping of ScimBa and Feel++.

1.1 Main Objective

Streamlined Data Exchange: Evaluating solutions trained by ScimBa using Feel++ tools.

1.2 Roadmap

1. Create a Poisson class that uses both solvers to solve Poisson PDEs.
2. Visualize and compare the results of both solvers with exact solutions.
3. Expand the use for variable anisotropy.
4. Compute L^2 and H^1 errors and trace their convergence for both solvers..

2 Work environment and tools

2.1 Exploring Feel++

Feel++ is a comprehensive library that allows manipulation of mathematical objects to solve Partial Differential Equations (PDEs). It also provides toolboxes for physics-based models and their coupling. These toolboxes include applications for fluid mechanics, solid mechanics, heat transfer, and more.

2.1.1 Coefficient Form Toolbox

1. **What are Coefficient Form PDEs?**: The coefficient forms in PDE (Partial Differential Equation) toolboxes encapsulate crucial properties like diffusion, convection, and reaction coefficients. These coefficients are vital for characterizing diverse PDEs such as elliptic, parabolic, or hyperbolic equations, each with its unique coefficient form. For instance, in the Poisson equation, a common elliptic equation, the coefficient form is often expressed as:

$$-\nabla \cdot (c\nabla u) + au = f$$

   - c : represents the diffusion coefficient,
   - a : represents the reaction coefficient,
   - u : is the unknown function, and
   - f : is the source term.

   PDE toolboxes, such as Feel++, offer features for handling different PDEs. They make it easier to define coefficients, set boundaries, discretize problems, and use numerical methods. This helps users to solve complex PDEs, study physical phenomena, and simulate real-world situations more efficiently.

2. **System of PDEs**: Many PDEs can be expressed in a standard form, mainly based on the coefficients' definition. We use the following equation to find this form:
   $u : \Omega \subset R^d \longrightarrow R^n$ with $d = 2, 3$ and $n = 1$ (u is a scalar field) or $n = d$ (u is a vector field) such that

$$d\frac{\partial u}{\partial t} + \nabla \cdot (-c\nabla u - \alpha u + \gamma) + \beta \cdot \nabla u + au = f \text{ in } \Omega$$

- d : damping or mass coefficient
- c : diffusion coefficient
- α : conservative flux convection coefficient
- γ : conservative flux source term
- β : convection coefficient
- a : absorption or reaction coefficient
- f : source term

Parameters μ may depend on the unknown u and on the space variable x, time t, and other unknowns $u_1, \ldots, u_N$.

3. **Coefficients**: We also need to follow certain limitations on coefficient shapes, as detailed in the table below. .

Coefficient	Shape if Scalar Unknown	Shape if Vectorial Unknown
d	scalar	scalar
c	scalar or matrix	scalar or matrix
α	vectorial	scalar or matrix
γ	vectorial	matrix
β	vectorial	vectorial
a	scalar	scalar
f	scalar	vectorial

4. **Initial Conditions**: Initial Initial conditions set the initial values for each unknown variable in the equations. These conditions can be defined using expressions or fields.

2.2 Exploring ScimBa

ScimBa is a Python library designed to solve complex PDEs using Physics-Informed Neural Networks (PINNs), DeepONet, GaussianMixture, Neural-Galerkin, NormalizingFlows, and OdeLearning neural networks each tailored for different numerical and machine learning tasks.

2.2.1 Physics-Informed Neural Networks (PINNs)

Physics-Informed Neural Networks (PINNs)[1] integrate physical laws described by partial differential equations (PDEs) directly into the neural network training process. The key idea behind PINNs is to incorporate the residuals of the PDEs as part of the loss function during training. Specifically, given a PDE of the form:

$$\mathcal{N}[\mathbf{u}(\mathbf{x}, t)] = f(\mathbf{x}, t), \tag{1}$$

where $\mathcal{N}$ is a differential operator and f is a source term, a PINN seeks an approximate solution $\mathbf{u}_\theta$ parameterized by neural network weights θ. The total loss function combines data loss and physics loss:

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{data}}(\theta) + \mathcal{L}_{\text{physics}}(\theta), \tag{2}$$

with

$$\mathcal{L}_{\text{physics}}(\theta) = \frac{1}{N_f} \sum_{j=1}^{N_f} \left| \mathcal{N}[\mathbf{u}_\theta(\mathbf{x}_j, t_j)] - f(\mathbf{x}_j, t_j) \right|^2 . \tag{3}$$

We decided to start using the examples in the ScimBa repository of uses of the Physics-Informed Neural Networks (PINNs). PINNs integrate the underlying physical laws described by PDEs directly into the learning process of neural networks. This is achieved by constructing a loss function that penalizes the network for failing to fit known data and for violating the given physical laws.

[1]Reference: M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations," Journal of Computational Physics, vol. 378, pp. 686-707, 2019.

2.3 A Replicable Environment with Docker

To ensure consistent and replicable results across different environments, Docker was employed. Docker containers allow for the encapsulation of the entire software environment, ensuring that the same setup can be replicated across different machines without dependency issues.

```
1
2  # Start with the Feel++ base image
3  FROM ghcr.io/feelpp/feelpp:jammy
4
5  # Set labels for metadata
6  LABEL maintainer="Rayen Tlili <rayen.tlili@etu.unistra.fr>"
7  LABEL description="Docker image with Feel++ & ScimBa"
8
9  USER root
10
11 # Install system dependencies
12 RUN apt-get update && apt-get install -y \
13     git \
14     xvfb
15
16 # Install Python libraries
17 RUN pip3 install torch xvfbwrapper pyvista plotly panel
       ipykernel matplotlib tabulate nbformat gmsh
18
19
20 # Clone the Scimba repository
21 RUN git clone https://gitlab.inria.fr/scimba/scimba.git /
       workspaces/2024-stage-feelpp-scimba
22
23 # Install Scimba and its dependencies
24 WORKDIR /workspaces/2024-stage-feelpp-scimba/scimba
25 RUN pip3 install scimba
26
27 # Copy the xvfb script into the container for visualization
28 COPY tools/load_xvfb.sh /usr/local/bin/load_xvfb.sh
29 RUN chmod +x /usr/local/bin/load_xvfb.sh
30
31 # Set the script to initialize the environment
32 CMD ["/usr/local/bin/load_xvfb.sh"]
```

Listing 1: Dockerfile for Feel++, Scimba, and Python libraries.

This Dockerfile creates a docker image with Feel++ as a base and installs the dependencies and libraries needed to run ScimBa in that environment.

3 Tools for Solving and Visualizing Solutions

This section details the process of setting up the environment and solving the PDEs, particularly focusing on the Poisson equation. By using Feel++ and ScimBa, we integrate traditional Galerkin methods with machine learning approaches to achieve accurate solutions efficiently.

3.1 Solving the PDE

To solve the Poisson equation, we create the environment with Feel++ and configure the settings accordingly. Listing 2 demonstrates the setup and initialization.

```
import sys
import feelpp
import feelpp.toolboxes.core as tb
from tools.Poisson import Poisson

sys.argv = ["feelpp_app"]
e = feelpp.Environment(sys.argv,
                       opts=tb.toolboxes_options("coefficient-form-pdes", "cfpdes"),
                       config=feelpp.localRepository('feelpp_cfpde'))

P = Poisson(dim = 2)
P(  h=0.05,             # mesh size
    order=1,            # polynomial order
    name='u',           # name of the variable u
    rhs='8*pi*pi*sin(2*pi*x)*sin(2*pi*y)',  # right hand side
    diff='{1,0,0,1}', # diffusion matrix
    g='0',              # Dirichlet boundary conditions
    gN='0',             # Neumann boundary conditions
    shape='Rectangle',# domain shape (Rectangle, Disk)
    geofile=None,       # geometry file
    plot=1,             # plot the solution
    solver='feelpp',  # solver
    u_exact='sin(2 * pi * x) * sin(2 * pi * y)',
    grad_u_exact = '{2*pi*cos(2*pi*x)*sin(2*pi*y),2*pi*sin(2*pi*x)*cos(2*pi*y)}'
    )
```

Listing 2: Initialization and Configuration for Poisson Solver.

The program then creates a json model which will be used by the feelpp solver to plot the solution and the convergence rate for the errors.

```
    self.model = lambda order,dim=2,name="u": {
      "Name": "Poisson",
      "ShortName": "Poisson",
      "Models":
      {
        f"cfpdes-{self.dim}d-p{self.order}":
        {
          "equations":"poisson"
        },
        "poisson":{
          "setup":{
            "unknown":{
              "basis":f"Pch{order}",
              "name":f"{name}",
              "symbol":"u"
            },
            "coefficients":{
              "c": f"{diff}:x:y" if self.dim == 2 else f"{diff}:x:y:z",
              "f": f"{rhs}:x:y" if self.dim == 2 else f"{rhs}:x:y:z",
              "a": f"{a}"
            }
          }
        }
      }
                [. . .]
```

Listing 3: CFPDE json model

3.2 Computing L2 and H1 errors

L2 and H1 Errors: These errors are critical for assessing the accuracy of the solution. The L2 error measures the average error across the domain, while the H1 error accounts for both the function value and its gradient, providing a more comprehensive error analysis.

The L2 error is particularly useful for quantifying how well the solution approximates the exact solution in a global sense, capturing the average behavior of the error.

The H1 error is particularly important in problems where the solution's smoothness or the behavior of its gradient is essential, such as in PDEs that involve flux or gradient-driven phenomena.

```python
def runLaplacianPk(P, df, model, measures, verbose=False):
  """Generate the Pk case"""
  meas = dict()
  dim, order, json = model
  for i, h in enumerate(df['h']):
    m= measures[i]   #feel_solver(filename=fn, h=h, json=json,
        dim=dim, verbose=verbose)
    print('measure =' , m)
    for norm in ['L2', 'H1']:
      meas.setdefault(f'P{order}-Norm_poisson_{norm}-error', [])
      meas[f'P{order}-Norm_poisson_{norm}-error'].append(m.get(f
          'Norm_poisson_{norm}-error'))
  df = df.assign(**meas)
  for norm in ['L2', 'H1']:
    df[f'P{order}-poisson_{norm}-convergence-rate']=np.log2(df[f
        'P{order}-Norm_poisson_{norm}-error'].shift() / df[f'P{
        order}-Norm_poisson_{norm}-error']) / np.log2(df['h'].
        shift() / df['h'])

  return df

def runConvergenceAnalysis(P, json, measures, dim=2,hs=[0.1,
    0.05, 0.025, 0.0125],orders=[1],verbose=False):
  df=pd.DataFrame({'h':hs})
  for order in orders:
    df=runLaplacianPk(P, df=df,model=[dim,order,json(dim=dim,
        order=order)], measures = measures,verbose=verbose)
  print('df = ', df.to_markdown())
  return df
```

Listing 4: Computing L2 and H1 errors

To generate and display a plot of convergence rates across mesh sizes for each polynomial order:

```
def plot_convergence(P, df,dim,orders=[1]):
  fig=px.line(df, x="h", y=[f'P{order}-Norm_poisson_{norm}-error
      ' for order,norm in list(itertools.product(orders,['L2','H1
      ']))])
  fig.update_xaxes(title_text="h",type="log")
  fig.update_yaxes(title_text="Error",type="log")
  for order,norm in list(itertools.product(orders,['L2','H1'])):
    fig.update_traces(name=f'P{order} - {norm} error - {df[f"P{
        order}-poisson_{norm}-convergence-rate"].iloc[-1]:.2f}',
        selector=dict(name=f'P{order}-Norm_poisson_{norm}-error')
        )
  fig.update_layout(
          title=f"Convergence rate for the {dim}D Poisson
              problem",
          autosize=False,
          width=900,
          height=900,)
  return fig
```

Listing 5: Plotting the convergence rate

This is the Error convergence rate for a Laplacian problem using Feel++:

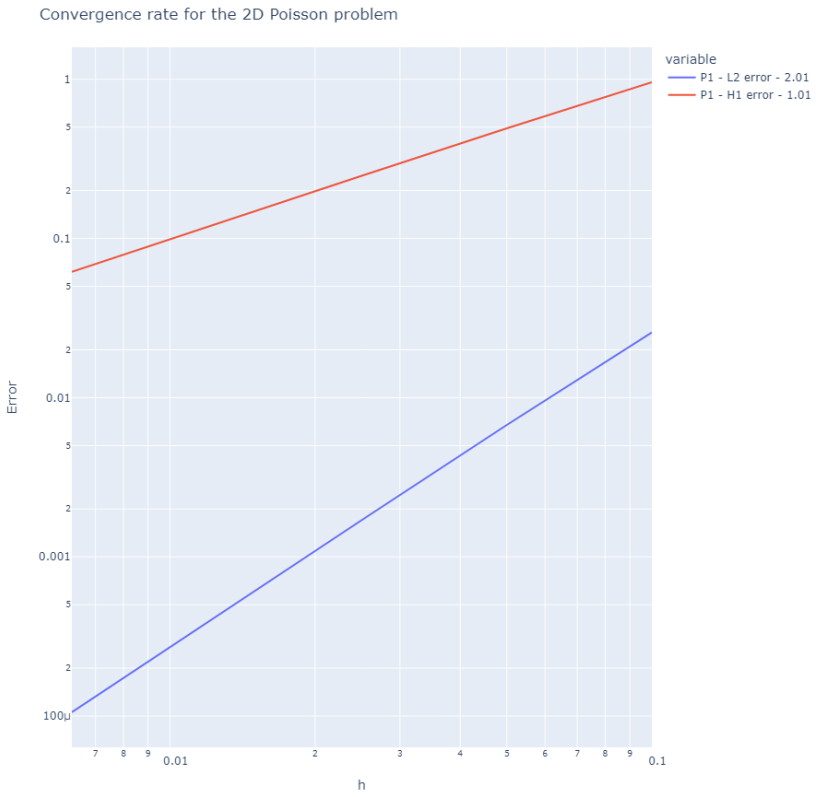


Figure 1: Error Convergence Rate for Laplacian Problem

3.3 Generating visuals using ScimBa

We begin by defining the spatial domain `xdomain` using ScimBa's SpaceDomain module. In this case, we specify a two-dimensional domain using a square domain configuration spanning from $(0.0, 0.0)$ to $(1.0, 1.0)$;

Next, we create an instance of the Poisson equation `pde` in two dimensions, specifying the right-hand side (`rhs`) as well as the boundary condition function:

```
xdomain = domain.SpaceDomain(2, domain.SquareDomain(2, [[0.0,
    1.0], [0.0, 1.0]]))
pde = Poisson_2D(xdomain)
u , pinn = Run_Poisson2D(pde)
```

Listing 6: Example for square domain

Finally, we execute the `Run_Poisson2D` function, which solves the Poisson equation defined by `pde`:

```

def Run_Poisson2D(pde, epoch=1000, bc_loss_bool=True, w_bc=10,
    w_res=10):

    # Initialize samplers
    x_sampler = sampling_pde.XSampler(pde=pde)
    mu_sampler = sampling_parameters.MuSampler(
        sampler=uniform_sampling.UniformSampling, model=pde
    )
    sampler = sampling_pde.PdeXCartesianSampler(x_sampler,
        mu_sampler)

    file_name = "test.pth"
    new_training = True
                [. . .]
    # Plot and print the coordinates and values of u
    n_visu = 20000
    reference_solution = True
    trainer.plot(n_visu, reference_solution=True)
    u = pinn.get_w

    return u , pinn
```

Listing 7: Training the solution

The visual representation generated is depicted in Figure 2:

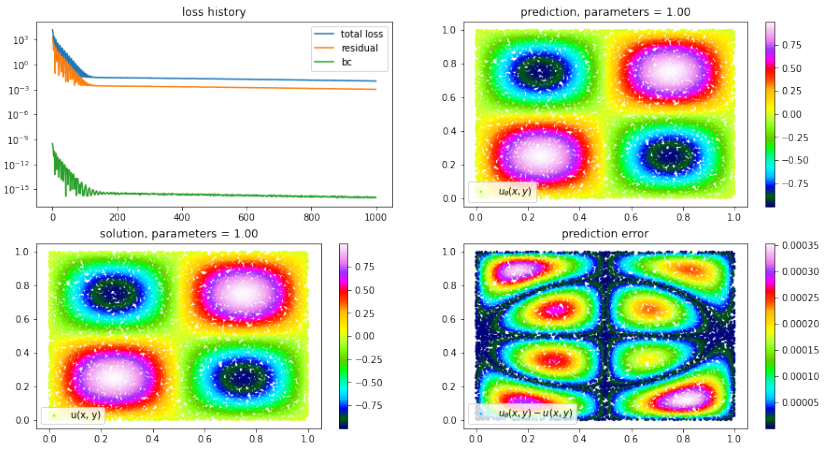


Figure 2: Visualization of Solution using ScimBa

3.4 Extracting the Solution from ScimBa

To extract the solution from ScimBa, we add the `scimba_solver` method to the `Poisson` class. This method allows us to train the function using ScimBa and then extract the solution. The following code demonstrates this process:

```
def scimba_solver(self, h, shape='Rectangle', dim = 2, verbose=False):
    if verbose:
        print(f"Solving a Poisson problem for h = {h}...")

    diff = self.diff.replace('{', '(').replace('}', ')')
    if shape == 'Disk':
        xdomain = domain.SpaceDomain(2, domain.DiskBasedDomain(2, center=[0.0, 0.0], radius=1.0))
    elif shape == 'Rectangle':
        xdomain = domain.SpaceDomain(2, domain.SquareDomain(2, [[0.0, 1.0], [0.0, 1.0]]))
    pde = Poisson_2D(xdomain, rhs=self.rhs, diff=diff, g=self.g, u_exact=self.u_exact)
    u , pinn = Run_Poisson2D(pde, epoch=200)

    return u
```

Listing 8: Method for Extracting Solution from ScimBa.

The `scimba_solver` method performs the following steps:

- Initializes the problem domain and defines the PDE using the specified parameters.
- Trains the neural network using the `Run_laplacian2D` function.
- Extracts the solution function `u` from the trained network.

To read the mesh and evaluate the predicted solution from ScimBa on the mesh points:

```
coordinates_tensor = torch.tensor(coordinates, dtype=torch.
    float64, requires_grad=True)
print(f"Shape of input tensor (coordinates): {coordinates_tensor
    .shape}")

points = coordinates_tensor
labels = torch.zeros(len(points))
data = domain.SpaceTensor(points, labels, boundary=True)
mu = torch.ones((len(points), 1), dtype=torch.float64,
    requires_grad=True)

scimba_solution = []

u_values = u_scimba(data, mu)

for point, u_value in zip(points, u_values):
    print(f"u( {point[0:]} ) = {u_value[0]}")
    u_value_np = u_value.detach().numpy()

    scimba_solution = np.append(scimba_solution, u_value_np[0])


print(f"ScimBa solution: {scimba_solution}")
print(f"Feel++ solution: {feel_solution}")
print(f"Exact solution: {u_ex}")
print("\n Difference |scimba_solution - feel_solution| : ", np.
    abs(scimba_solution - feel_solution))
    [...]
```

Listing 9: Evaluating ScimBa Solution on Mesh Points.

This way, we can effectively train the Poisson function with ScimBa, extract the solution, and evaluate it on the mesh coordinates.

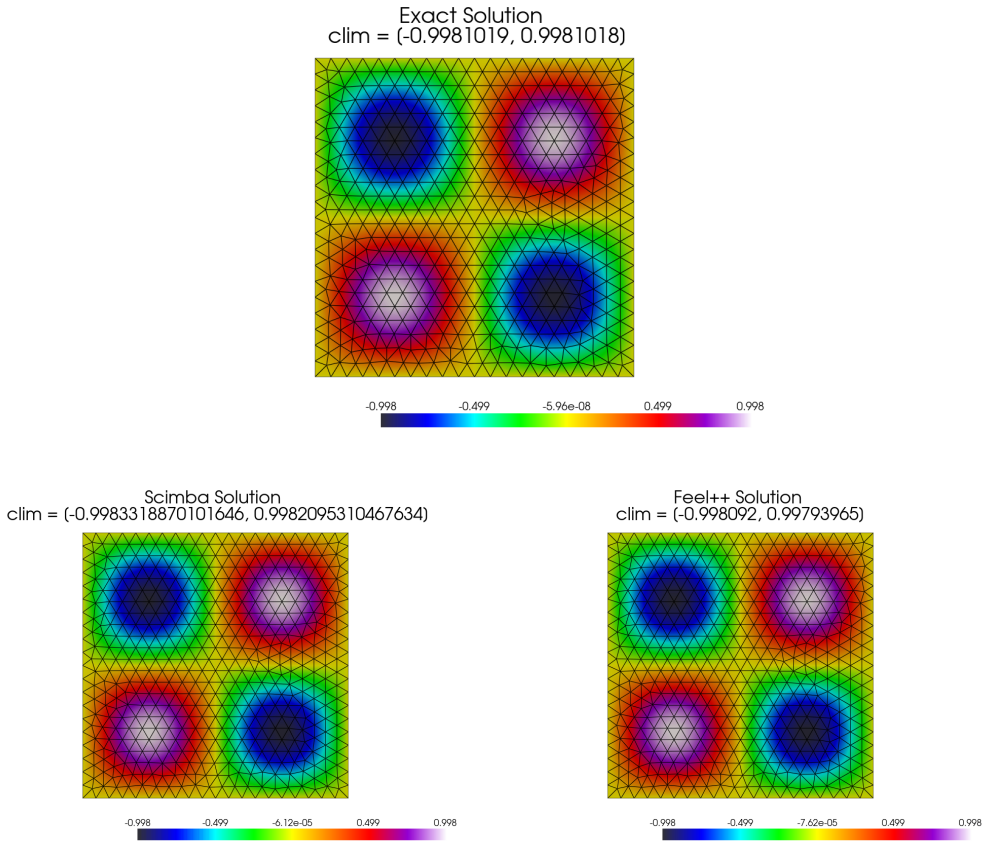
3.4.1 Plotting the two solutions on the same mesh

Once the solution is trained and extracted we want to compare it to the one we found through using Galerkin methods with Feel++:

```
err_feel = np.abs(u_ex - feel_solution) / np.abs(u_ex)
err_scimba = np.abs(u_ex - scimba_solution) / np.abs(u_ex)
clim_err = [np.min(err_feel), np.max(err_feel)]

pv_plot(mesh[0].copy(), err_feel, title=f'|u_exact - u_feel|/|u_exact| \n clim = {clim_err}', clim=clim_err)
pv_plot(mesh[0].copy(), err_scimba, title=f'|u_exact - u_scimba|/|u_exact| \n clim = {[np.min(err_scimba), np.max(err_scimba)]}', clim=[np.min(err_scimba), np.max(err_scimba)])
```

Listing 10: Comparison of Feel++ and ScimBa Solutions on the Same Mesh.



Solution plots

3.5 Varying anisotropy

We define the anisotropy matrix in the Poisson_pinns class that evaluates the "diff" vector given as argument and transforms it into the adequate shape to be treated by the pinns algorithm.

```
def anisotropy_matrix(w, x, mu):
    x1, x2 = x.get_coordinates()
    diff_expr = eval(self.diff, {'x': x1, 'y': x2, 'pi': PI, '
        sin': torch.sin, 'cos': torch.cos, 'exp': torch.exp})
    diff_tensors = [torch.tensor(element, dtype=torch.float64,
        requires_grad=True).unsqueeze(-1) for element in
        diff_expr]
    target_shape = diff_tensors[0].shape
    diff_tensors = [tensor.expand(target_shape) for tensor in
        diff_tensors]

    return torch.cat(diff_tensors, dim=-1)
```

Listing 11: Definition of Anisotropy Matrix in Poisson Class.

3.6 Major setback

By implementing a varying anisotropy the loss function stagnates at certain points, the program seems to also need a lot of training and the result does not vary much with adjustments on the optimizers or the layers of neural networks.

4 Validation

This section focuses on validating the methods used by comparing the results from ScimBa and Feel++ against known exact solutions. The accuracy of these methods is measured using error analysis.

4.1 Generating meshes using Feel++

Feel++ produces geometry files for either a 2D rectangle or a 3D box. The generated file is compatible with Gmsh, facilitating subsequent mesh generation and finite element analysis. The characteristic length h controls the mesh resolution, and we define physical groups for boundaries and domains, which are crucial for setting boundary conditions and material properties in simulations.

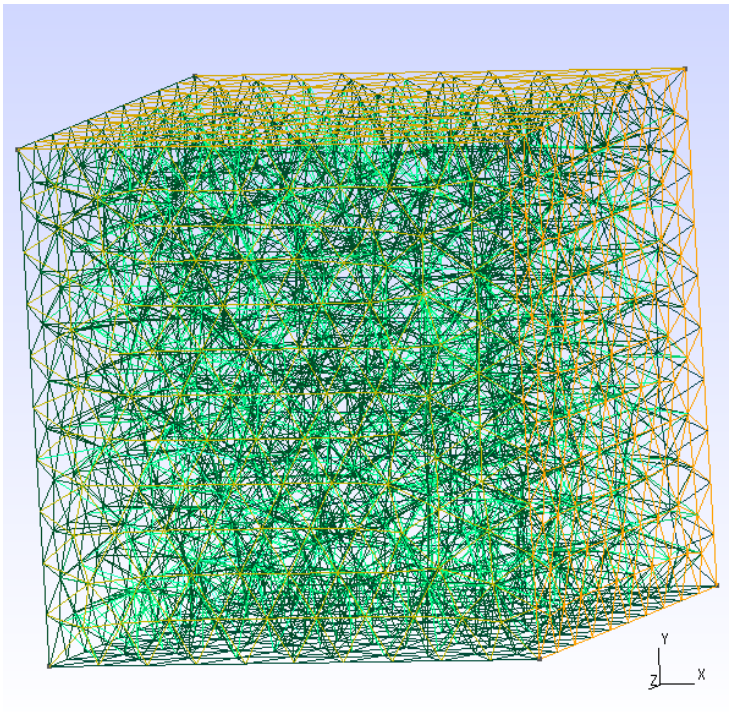


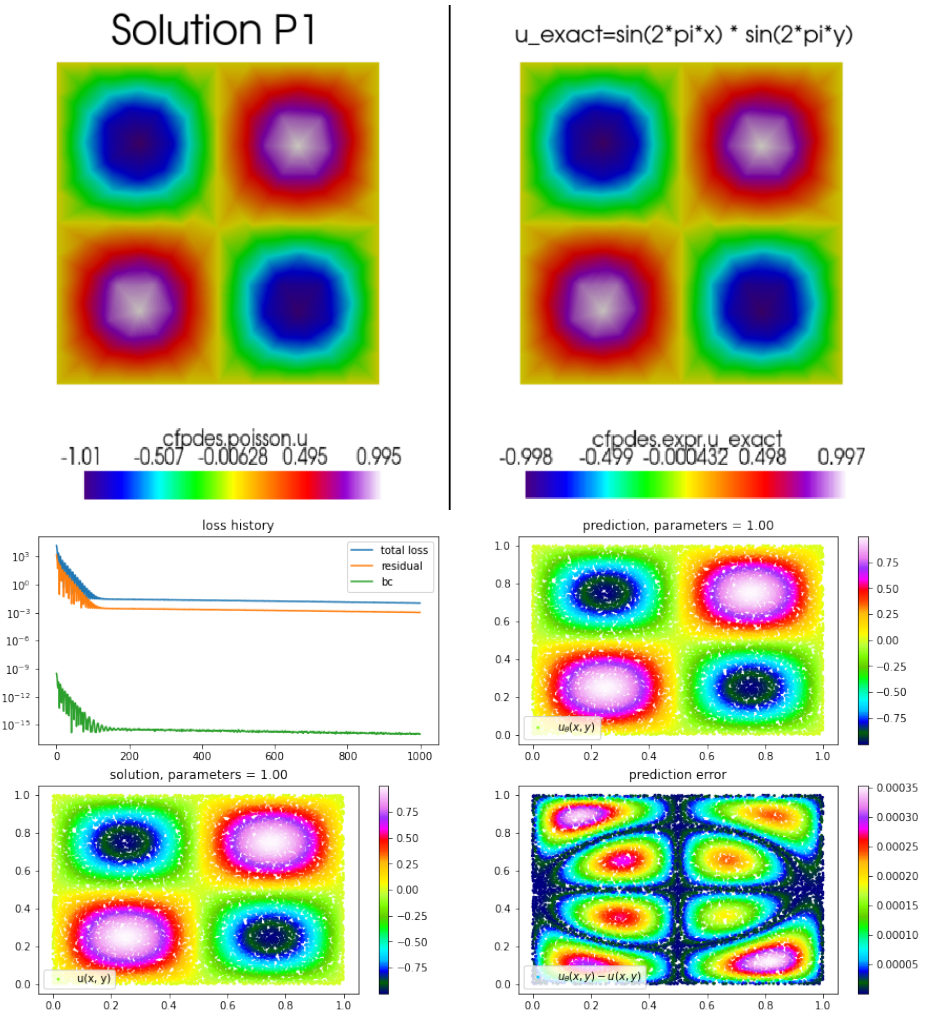
Figure 3: Generated 3D geometry and mesh viewed using gmsh

4.2 Comparing the visuals for a Laplacian problem

4.2.1 On a square domain

Solving the Laplacian problem with Dirichlet boundary conditions on a square domain using Feel++ and ScimBa

```
u_exact = 'sin(2*pi*x) * sin(2*pi*y)'
rhs = '8*pi*pi*sin(2*pi*x) * sin(2*pi*y)'
P(rhs=rhs, g='0', solver ='feelpp', u_exact = u_exact)
P(rhs=rhs, g='0', solver ='scimba', u_exact = u_exact)
```



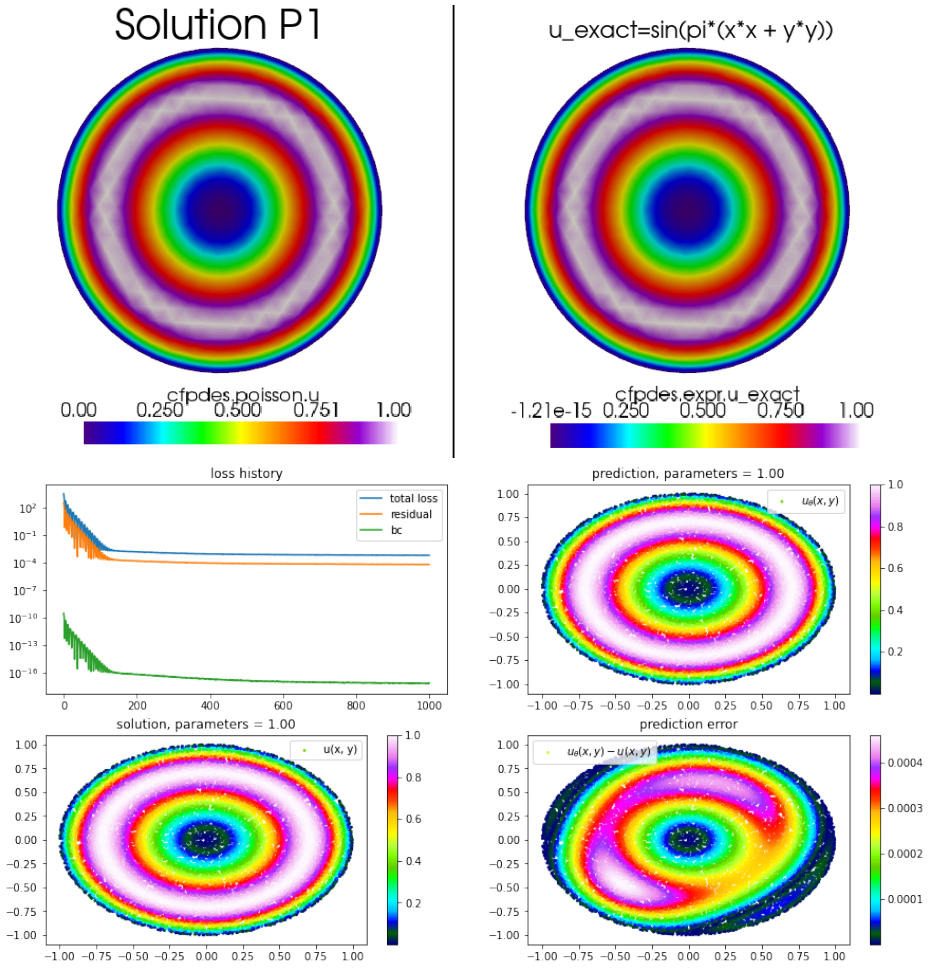
Laplacian on a square domain

4.2.2 On a disk domain

In this part, the focus shifts to solving the Laplacian problem on a disk domain. The exact solution $u = \sin(\pi(x^2 + y^2))$ and its corresponding f are tailored to test the solvers' capabilities in more complex geometrical contexts.

```
u_exact = 'sin(pi*(x*x + y*y))'
rhs = '-4*pi*cos(pi*(x*x + y*y)) + 4*pi*pi*(x*x + y*y)*sin(pi*(x*x + y*y))'
P(rhs=rhs, g='0', shape='Disk', solver='scimba', u_exact=u_exact)
```

Listing 12: Laplacian Problem on Disk Domai



Laplacian on a disk domain

4.3 Visualizing solutions on the same graph

By visualizing both solutions on the same graph, we can directly compare the accuracy and differences between the two methods. The figure below shows the output from the solver, including information about the meshing process and the extracted solution values.

```
0  cfpdes.expr.grad_u_exact
1           cfpdes.expr.rhs
2       cfpdes.expr.u_exact
3          cfpdes.poisson.u
Number of features in coordinates: 3
Number of points: 517

Nodes from export.case: [[0.82477343 0.04606718]
 [0.8300841  0.10191753]
 [0.7806651  0.09123866]
 ...
 [0.8390992  0.3016979 ]
 [0.8367227  0.1427201 ]
 [0.7476512  0.5844005 ]]
```

Figure 4: mesh information visualized on the mesh coordinates.

Figure 4 displays the log information generated during the meshing process. The log provides details about the meshing process, including the available function values, number of nodes and node coordinates.

4.3.1 Comparing error relative to the theoretical exact solution

We consider the following mathematical system:

$$u_{\text{exact}} = y + \left(x(1-x) + y(1-y) \cdot \frac{1}{4} \right)$$

The corresponding Poisson equation with a source term f and boundary condition g is given by:

$$-\Delta u = \frac{5}{2} \quad \text{in} \quad \Omega$$

$$u = y + \left(x(1-x) + y(1-y) \cdot \frac{1}{4} \right) \quad \text{on} \quad \partial\Omega$$

where Δ denotes the Laplace operator, Ω is the domain, and $\partial\Omega$ is the boundary of the domain. Let's see what this does

```
u_exact = 'y + (x*(1-x) + y*(1-y)*0.25) '
P(h = 0.05, rhs='2.5', g='y', solver ='scimba', u_exact =
    u_exact)
```

Listing 13: Poisson Problem on Square Domain with Theoretical Exact Solution.

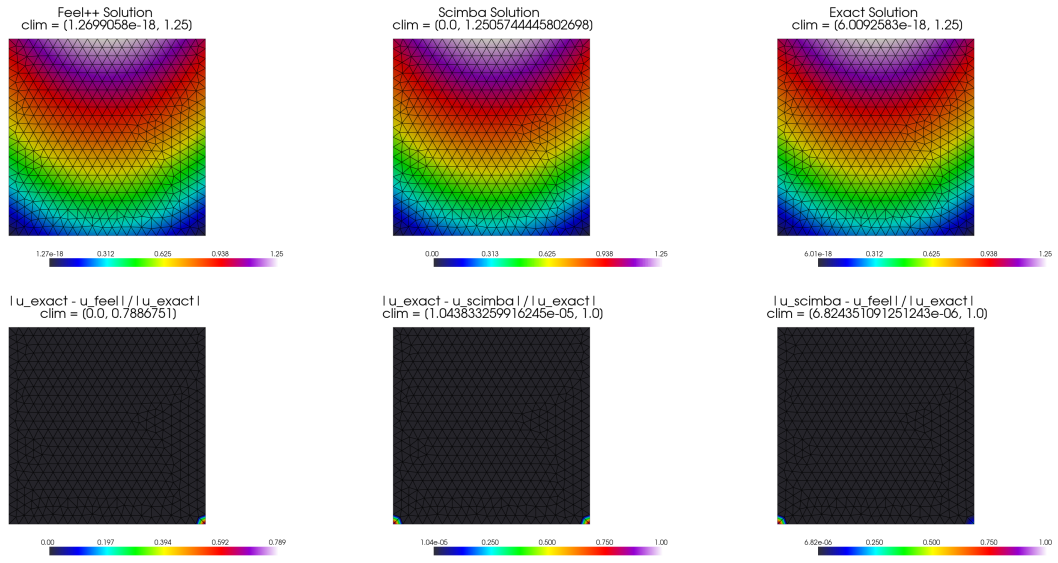


Figure 5: Visuals generated

We can observe slight differences on the bottom on the graph. "clim" represents the minimum and maximum values taken by the solution. The relative error in this case seems to be negligible. We can conclude that both our solvers approximate the solution pretty well, the Feel++ solution seems to be nonetheless a bit more accurate in this case. Here we have h = 0.05, which gives us a 517 point mesh, and scimba is trained with 5000 collocation points.

4.3.2 Comparing absolute errors

We are comparing the solutions provided by both solvers with the exact solution and seeing the residual from the difference in each point of the plot

Feel solver absolute error:

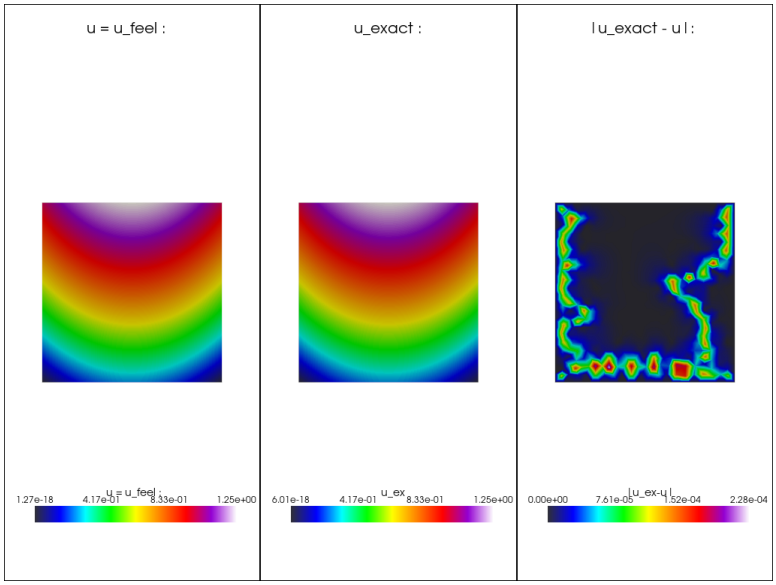


Figure 6: u_feel absolute error

Here we have

$$\|u_{\text{feel}} - u_{\text{exact}}\|_\infty = 0.00022828579$$

The differences seem to be spread out randomly.

This is what we get with a lower hsize = 0.0125 which amounts to 7554 mesh points:

$$\|u_{\text{feel}} - u_{\text{exact}}\|_\infty = 1.50203705e-05$$

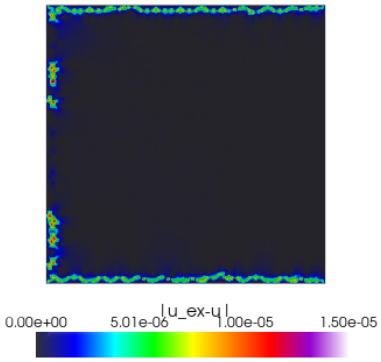


Figure 7: u_feel absolute error for a finer mesh

Scimba solver absolute error:

The scimba solver, once viewed on the same feelpp mesh, gives us an absolute error that's a lot less random and the points with the highest values seem to be the points where the error is also higher.

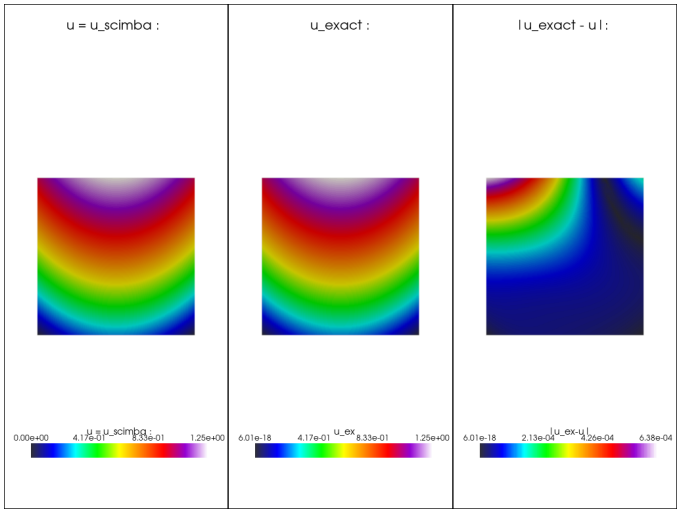


Figure 8: u_scimba absolute error

$$\|u_{\text{scimba}} - u_{\text{exact}}\|_\infty = 0.0010454412097140597$$

Here we can go from 5000 collocation points to 10000 without much change in the divergent values from the exact solution, the solution is much less accurate at 500 collocation points and at less than 200 epochs of training. Over 10000 the program gets too slow.

4.3.3 Error convergence rate

This is the Error convergence rate for the Poisson problem defined above using Feel++ tools.

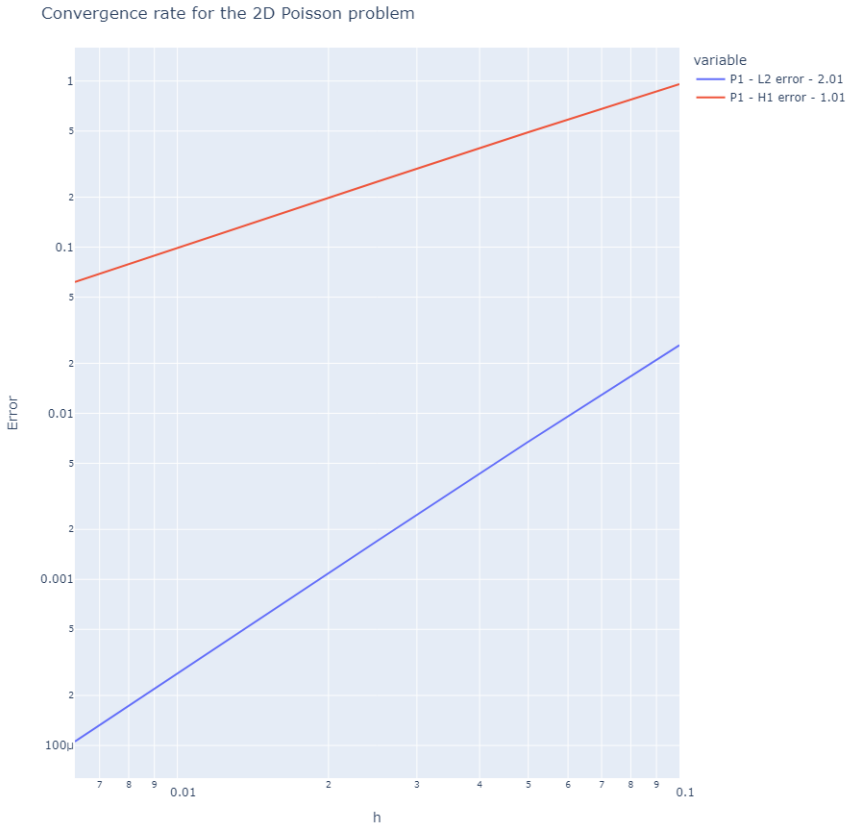


Figure 9: Error rate for the 2D Poisson problem

The graph in Figure 9 displays the convergence rates of L2 and H1 errors for the 2D Poisson problem using P1 polynomial order. The x-axis represents the mesh size (`h`) on a logarithmic scale, the y-axis shows the error values.

Each line corresponds to a different combination of polynomial order and error norm, with the convergence rate annotated at the end of the line. The plot demonstrates how the errors decrease as the mesh size is refined, and provides insight into the accuracy and efficiency of the numerical methods employed.

	h	P1-Norm_poisson_L2-error	P1-Norm_poisson_H1-error	P1-poisson_L2-convergence-rate	P1-poisson_H1-convergence-rate
0	0.1	0.001601	0.0468136	nan	nan
1	0.05	0.000407689	0.0233889	1.97343	1.00111
2	0.025	0.00010369	0.0119102	1.9752	0.973622
3	0.0125	2.59007e-05	0.00593982	2.00121	1.00371
4	0.00625	6.44037e-06	0.00295856	2.00778	1.00552

Figure 10: Error rate for the 2D Poisson problem

we can see the different values the errors take for different mesh sizes above. The bigger the number of discrete points (the smaller the mesh size h), the closer to the exact solution we get.

4.3.4 Resolving PDEs with varying anisotropy

We consider the following mathematical system: The exact solution is given by:

$$u_{\text{exact}} = \frac{x^2}{1+x} + \frac{y^2}{1+y}$$

The corresponding Poisson equation with a source term f and diffusion matrix D is given by:

$$-\nabla \cdot (D\nabla u) = f \quad \text{in} \quad \Omega$$

where the source term f and the diffusion matrix D are defined as:

$$f = -\frac{4+2x+2y}{(1+x)(1+y)}$$

$$D = \begin{pmatrix} 1+x & 0 \\ 0 & 1+y \end{pmatrix}$$

The boundary condition g is given by:

$$u = \frac{x^2}{1+x} + \frac{y^2}{1+y} \quad \text{on} \quad \partial\Omega$$

where $\nabla\cdot$ denotes the divergence operator, ∇ denotes the gradient, Ω is the domain, and $\partial\Omega$ is the boundary of the domain.

Solving this PDE, we get this:

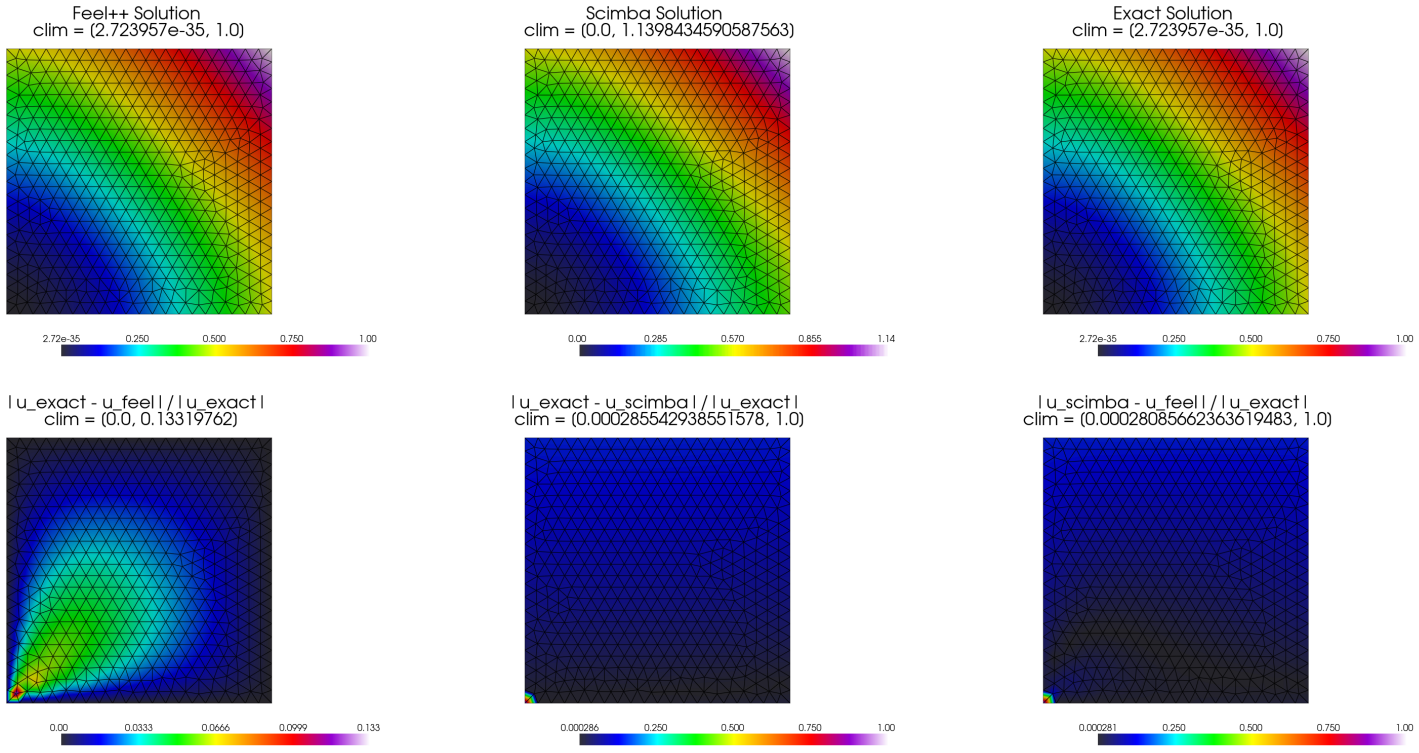


Figure 11: Error Visualization for Varying Anisotropy

The feelpp solver seems to spread the error out a little more but is of lower order than that of the scimba solver.

This time the feelpp error seems to be spread out more systemically Here

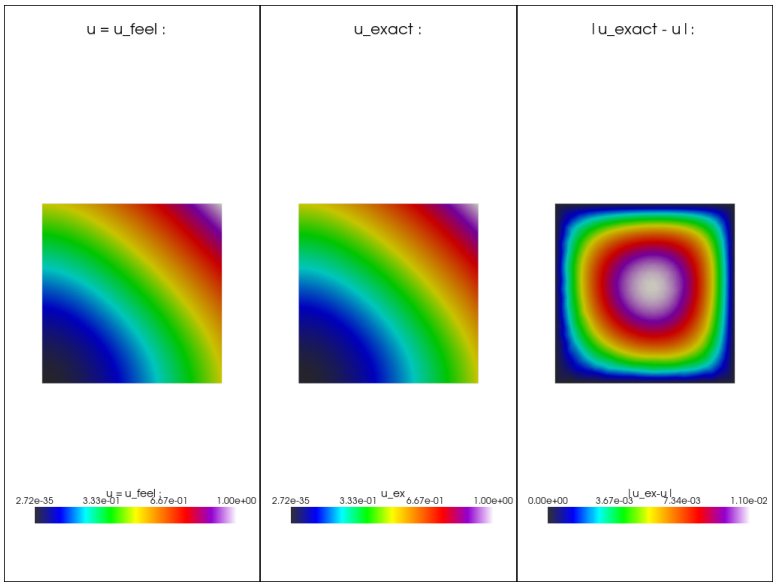


Figure 12: Error Visualization for Varying Anisotropy

we have

$$\|u_{\text{feel}} - u_{\text{exact}}\|_\infty = 0.00022828579$$

The scimba solver is consistent as the points with the highest values are again the points where the error is also higher.

$$\|u_{\text{scimba}} - u_{\text{exact}}\|_\infty = 0.0010454412097140597$$

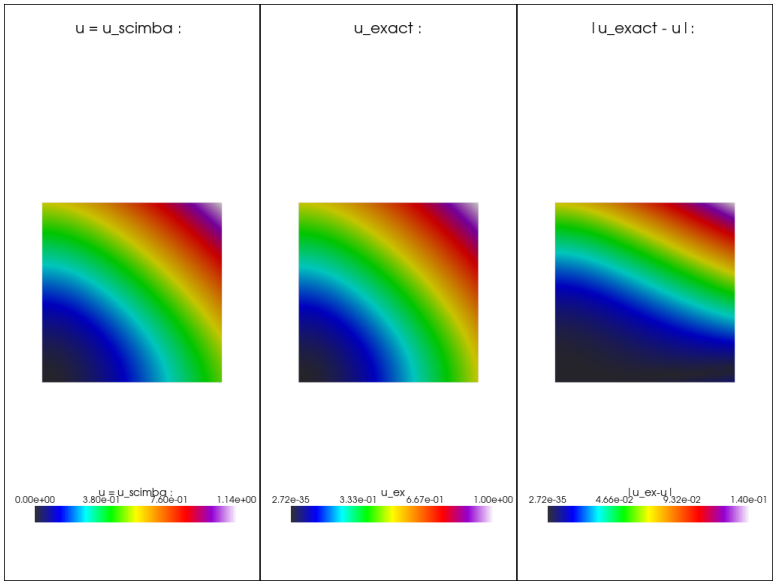


Figure 13: Error Visualization for Varying Anisotropy

4.3.5 Loss Stagnation Issue

In the training process, the loss function initially decreases as expected, but when a variable anisotropy matrix (`diff`) is introduced, the loss stagnates after a certain number of epochs. This indicates that the network is struggling to minimize the loss further, which may be due to the increased complexity introduced by the anisotropy matrix. Potential causes could include the learning rate being too high, leading to overshooting during optimization, or the network architecture not being well-suited to capture the behavior introduced by the anisotropy.

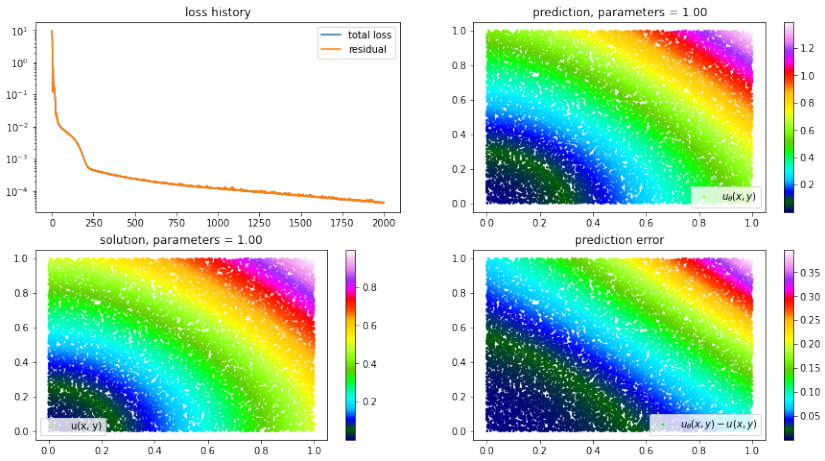


Figure 14: Loss history and prediction errors during the training process.

We have actively worked to address the issue of loss stagnation by experimenting with various approaches, such as adjusting the learning rate and modifying the network architecture. Although these efforts led to a slight decrease in the error, the process remained time-consuming, and the improvements were not substantial. The figure below illustrates the loss history and the corresponding prediction errors, highlighting the challenges we encountered in further minimizing the error.

5 Conclusion

Building on top of the previous project, the internship has given valuable time to adjust and fix the previous program. The goal is to streamline the process of solving complex mathematical problems and equations by wrapping ScimBa and Feel++. By integrating these two tools, we were able to better understand how they individually work and how they can interact together. Working with ScimBa gave us valuable insights in understanding and interpreting Machine Learning methods.

Feel++ provided a comprehensive framework for finite element analysis, offering customizable parameters for solving differential equations and simulating physical processes. By combining ScimBa's training capabilities with Feel++'s solver framework, we can leverage the both tools to solve problems more efficiently. This integration has enabled us to perform comprehensive analyses, visualize results in meaningful ways, and gain deeper insights into the capabilities of both solvers.

Here are key advances on the project as of this report:

1. Environment Setup:
   (a) The Feel++ environment and the CFPDE toolbox were well implemented within the program.
   (b) The Poisson class json model was effectively used to access and solve problems using the Feel++ solver adding further parameters for further complex problems.
   (c) Ability to extract and compare the ScimBa solution.
   (d) The program is able to run smoothly for any VScode user with docker.
2. Solving Poisson Equations:
   (a) Poisson equations were solved for 2D domains with further parameters, including a functioning anisotropy matrix for the ScimBa class.
   (b) The solutions were computed using both Feel++ and ScimBa solvers under the form of vectors that we were able to compute the error of with an exact theoretical solution as well as the errors' convergence rates.

3. Visualization:
   We were able to set up the different relevant functions on the same visual evaluated on the same nodes.
   Those functions include the Feel++ and the ScimBa solutions as well these errors that we aim to minimize :

$$\text{err\_feel} = \frac{|u_{\text{ex}} - \text{feel\_solution}|}{|u_{\text{ex}}|}$$

$$\text{err\_scimba} = \frac{|u_{\text{ex}} - \text{scimba\_solution}|}{|u_{\text{ex}}|}$$

$$\text{err\_sc\_feel} = \frac{|\text{scimba\_solution} - \text{feel\_solution}|}{|u_{\text{ex}}|}$$

   .

Challenges and setbacks:

1. Loss decrease stagnates or is very slow with varying anisotropy with ScimBa.
2. Slow with more complex problems.
3. The program is not ready for a wide ranging use but can be a start towards some creative uses.

Remaining work:

1. Doing further work to decrease the error rate for a problem with a varying anisotropy when using the scimba solver.
2. Implementing and testing Neumann boundary conditions and more complex geometries.
3. Look into implementing the brain hemisphere analogy and apply the program to relevant problems.

One thing I found very interesting working with these tools and the way the program was getting set up, was how it was reminiscent of our popular conception of how the brain hemispheres work.

Scimba can be likened to the left hemisphere of the brain, which handles analytical thinking and logic, utilizing machine learning techniques to optimize complex systems through data-driven algorithms. Conversely, Feel++ resembles the right hemisphere, focusing on creativity and holistic processing, using Galerkin methods to solve partial differential equations by capturing global behaviors in physical systems. The project integrates these tools to enhance solver accuracy and efficiency, with Feel++ as the foundation in the Docker container, mirroring the brain's structure in processing diverse inputs.

This opens up interesting directions for where this project could go were it to mature enough.

Thank You for your time and attention.

Bibliography

References.bib

References

[1] Feel++. (n.d.). *Finite method course*. Retrieved from https://feelpp.github.io/cours-edp/#/

[2] Feel++. (n.d.). *Python Feel++ Toolboxes*. Retrieved from https://docs.feelpp.org/user/latest/python/pyfeelpptoolboxes/index.html

[3] SciML. (n.d.). *Laplacian 2D Disk*. Retrieved from https://sciml.gitlabpages.inria.fr/scimba/examples/laplacian2DDisk.html

[4] ScimBa. (n.d.). *ScimBa Repository*. Retrieved from https://gitlab.inria.fr/scimba/scimba

[5] Feel++. (n.d.). *Feel++ GitHub Repository*. Retrieved from https://github.com/feelpp/feelpp

[6] Wikipedia. (n.d.). *Coupling (computer programming)*. Retrieved from https://en.wikipedia.org/wiki/Coupling_(computer_programming)

[7] SciML. (n.d.). *ScimBa*. Retrieved from https://sciml.gitlabpages.inria.fr/scimba/

[8] Feel++. (n.d.). *Feel++ Documentation*. Retrieved from https://docs.feelpp.org/user/latest/index.html

[9] Feel++. (n.d.). *Quick Start with Docker*. Retrieved from https://docs.feelpp.org/user/latest/using/docker.html