Internship Report: Scimba Feel++ Wrapper

Rayen Tlili

July 19, 2024

Contents

Abstract

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 is conducted in the form of an internship at the University of Strasbourg. It is the continuation of the project conducted with Helya Amiri.[1]

The aim is to establish a wrapper between the two libraries combining machine learning with traditional PDE solvers.

[1] helya.amiri@etu.unistra.fr

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 Objectives

1. **Streamlined Data Exchange**: Extracting the solution trained by the ScimBa solver and evaluate it on a Feel++ generated mesh.

2. **User Empowerment**: Create an interface that allows users to easily solve and export large numbers of PDEs.

3. **Integration of Technologies**: Integrate various technologies such as Docker, Python, and Git to create a reproducible environment for the project, apply machine learning techniques, solve PDEs, and manage source code.

We set the proper environment by using Feel++ as a base and installing ScimBa. A python class is set up with different methods to solve, export and visualize using either solvers.

1. **Comparing the results of both solvers**.

2. **Expanding the use for variable anisotropy and different boundary conditions**.

3. **User-Friendly Interface**.

4. **Provide comprehensive documentation and use examples**.

5. **Gather feedback and identify areas for improvement**.

1.2 Roadmap

.

1. Solve PDEs with a variable diffusion matrix using ScimBa PINNs.
2. Compare the results of both solvers with exact solutions.
3. Compute L^2 and H^1 errors and trace their convergence for both solvers.
4. Add Neumann and Robin boundary conditions.
5. Fine tune a solution informed by Galerkin methods and Machine Learning algorithms.

2 Work environment and tools

2.1 Exploring Feel++

First, both of us explored the Feel++ toolboxes on our own. Feel++ is a 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 conjugate heat transfer
- Fluid-structure interaction
- Electro and magnetostatics
- Thermoelectrics
- Levelset and multifluid

2.1.1 Exploring Feel++ toolboxes

As of the first meeting with the project supervisors, we've taken a look at the different toolboxes Feel++ has to offer in Python: .

1. Getting started with toolboxes in Python

Feel++ toolboxes are availabe as python modules. The following toolboxes are available:

Toolbox	Python Module
coefficient form	feelpp.toolboxes.cfpdes
fluid mechanics	feelpp.toolboxes.fluid
heat transfert	feelpp.toolboxes.heat
solid mechanics	feelpp.toolboxes.solid
electric	feelpp.toolboxes.electric
hdg	feelpp.toolboxes.hdg

An interesting toolbox to start with is the **Coefficient Form PDEs**:

2.1.2 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.

   Boundary Conditions:

   - Dirichlet
   - Neumann
   - Robin

2.2 ScimBa Physics-Informed Neural Networks (PINNs)

Physics-Informed Neural Networks (PINNs)[2] 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.

[2]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 Dockerfile

```
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 Methodology

3.1 Solving the PDE

First, we need to create the right environment with specific options and configurations.

```
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'))
```

In the code above, we import the necessary modules including system-specific parameters, Feel++, and the core components of Feel++ toolboxes. We also import the 'Poisson' class from the 'tools' folder. The environment is initialized with specific options and configuration for coefficient-form PDEs.

The Poisson class takes the following arguments:

```
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)}'
    )
```

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}"
            }
          }
        }
      }
                [. . .]
```

3.2 Computing L2 and H1 errors

3.2.1 Computing the errors

This function iterates over a set of mesh sizes (‘h‘), computes the solution using a specified computational model, and appends the L2 and H1 errors to the dataframe.

```
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
```

3.2.2 Plotting the convergence rate

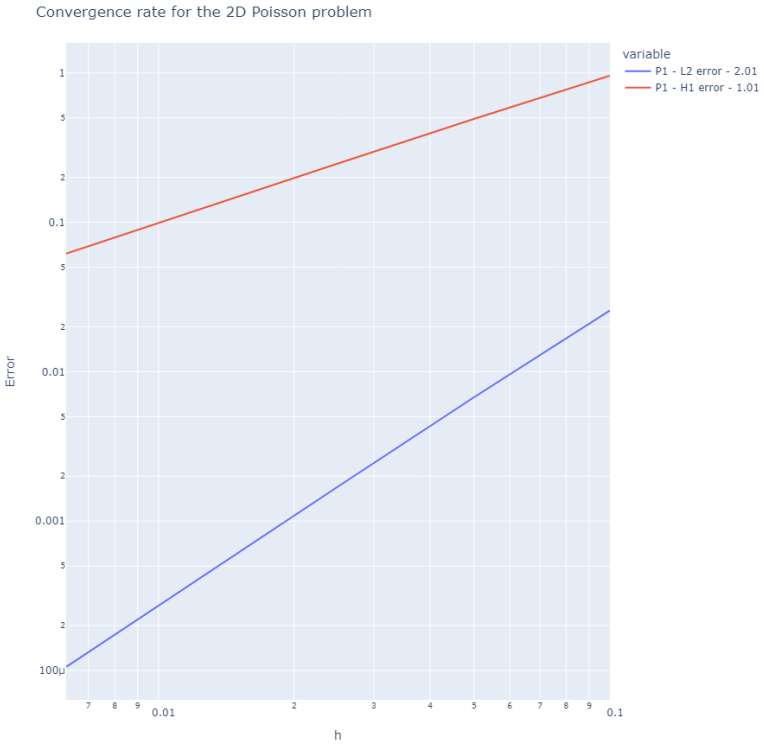
Run the convergence analysis for different mesh sizes and polynomial orders.

```
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
```

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

```python
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
```

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



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 $(1.0, 1.0)$;

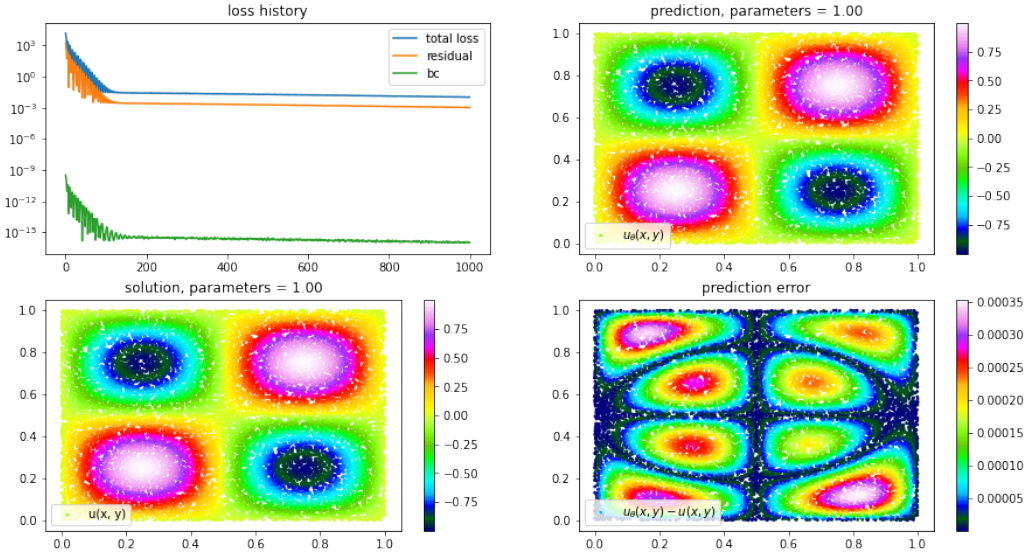
Next, we create an instance of the Poisson equation `pde` in two dimensions, specifying the right-hand side (`rhs`) as well as the boundary condition function:

```

# Laplacien strong Bc on Square with nn
xdomain = domain.SpaceDomain(2, domain.SquareDomain(2, [[0.0,
    1.0], [0.0, 1.0]]))
pde = Poisson_2D(xdomain)
u , pinn = Run_Poisson2D(pde)
```

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
```

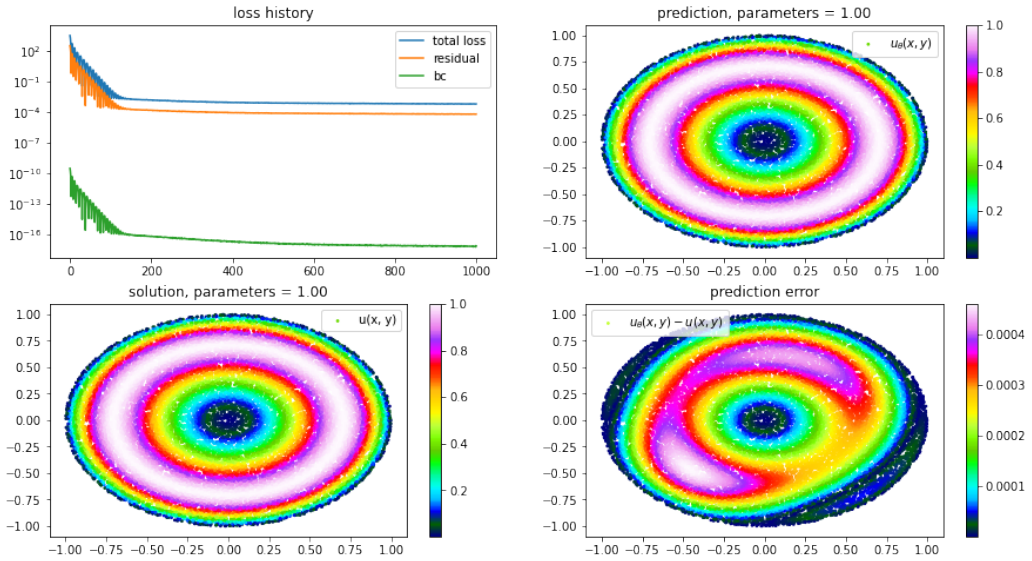
The visual representation generated by the above code snippet is depicted below: .



Then we initiate the visualization process by defining the spatial domain `xdomain` using ScimBa's SpaceDomain module. In this instance, we specify a two-dimensional domain utilizing a disk-based configuration with a center at $(0.0, 0.0)$ and a radius of 1.0.

```
xdomain = domain.SpaceDomain(2, domain.DiskBasedDomain(2, center
    =[0.0, 0.0], radius=1.0))

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))'

pde_disk = Poisson_2D(xdomain, rhs=rhs, g='0', u_exact = u_exact
    )
u, pinn = Run_Poisson2D(pde_disk)
```

The visual representation generated by the provided code snippet is presented below: .



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
```

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.

We then proceed to parsing and extracting the relevant data from the mesh file and solution.

```
if solver == 'scimba':
      import pyvista as pv
      import torch

      u_scimba = self.scimba_solver( h=h, shape=shape, dim=self.
         dim, verbose=True)

      # File path to the .case file
      file_path = 'cfpdes-2d-p1.exports/Export.case'

      # Read the .case file using PyVista
      data = pv.read(file_path)

      # Iterate over each block in the dataset to find
         coordinates
      coordinates = None
      for i, block in enumerate(data):
        if block is None:
          continue
        # Extract the mesh points (coordinates)
        coordinates = block.points
        solution = 'cfpdes.poisson.u'
        solution_expression = block.point_data[solution]

        df = pd.DataFrame(block.point_data)
        print(df.head())

      # Considering only 2d problems:
      num_features = coordinates.shape[1]
      if num_features > 2:
        coordinates = coordinates[:, :2]

      feel_solution = block.point_data['cfpdes.poisson.u']
      u_ex = block.point_data['cfpdes.expr.u_exact']
```

This piece of code:

- Reads the mesh points from the specified mesh file.
- Extracts the solution vector.

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

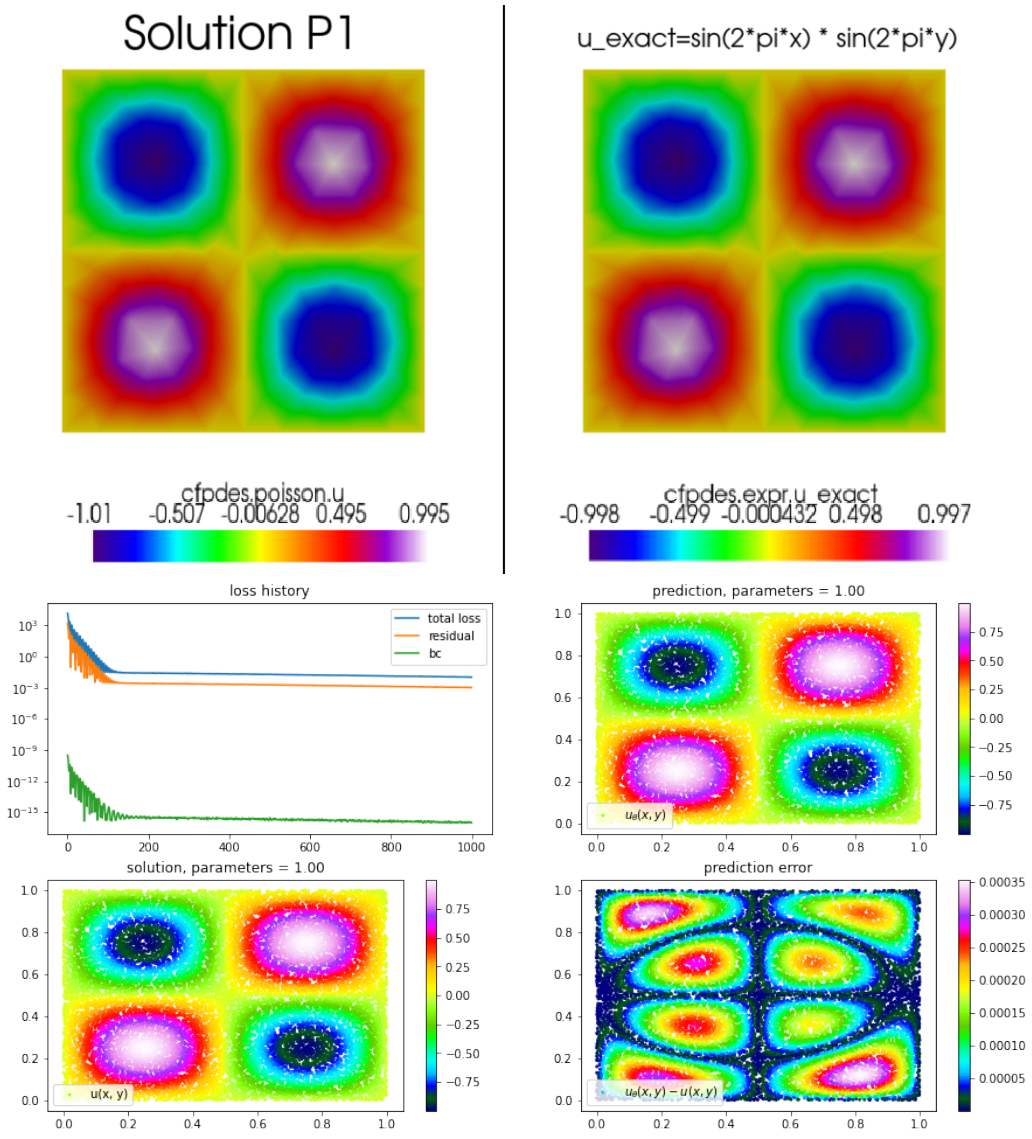
```
coordinates_tensor = torch.tensor(coordinates, dtype=torch.
    float64, requires_grad=True)
print(f"Shape of input tensor (coordinates): {coordinates_tensor
    .shape}")

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))
    [...]
```

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

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)
```

.

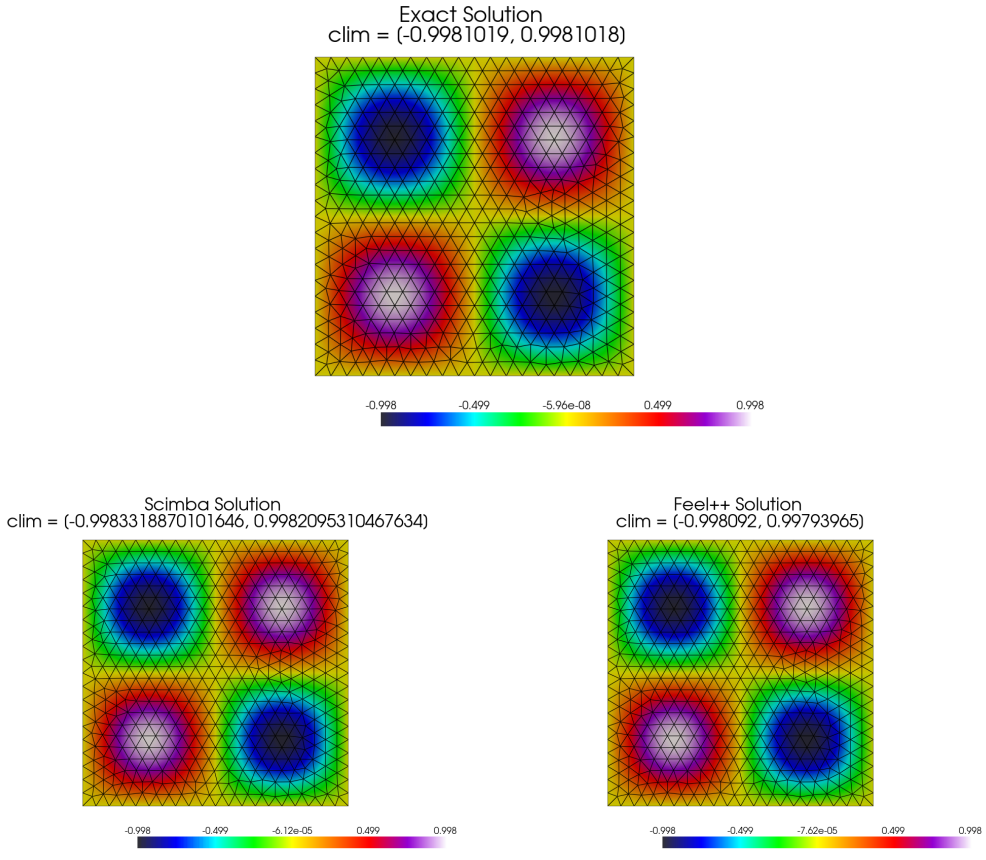


3.4.1 Plotting the two solutions on the same mesh

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)])
```

.



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()

# Evaluate diff elements to tensors
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) if not isinstance(element,
    torch.Tensor) else element.unsqueeze(-1) for element in
    diff_expr]

# Ensure all tensors have the same shape
target_shape = diff_tensors[0].shape
diff_tensors = [tensor.expand(target_shape) for tensor in
    diff_tensors]

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

self.anisotropy_matrix = anisotropy_matrix
```

4 Results

4.1 Generating visuals 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.

We define the method genGeometry within our class to generate the desired geometry:

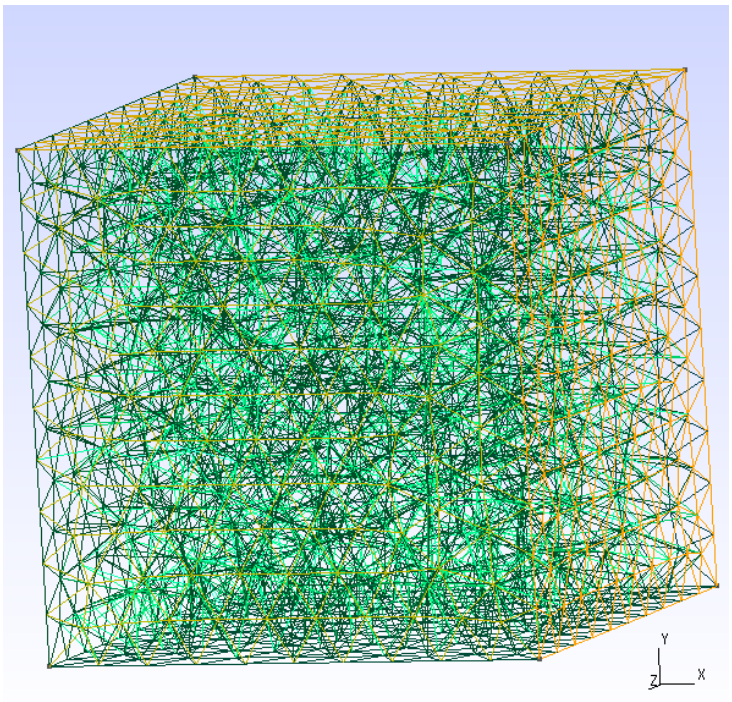
```
1
2   def genGeometry(self, filename, h, shape = 'Rectangle'):
3     """
4     Generate a cube geometry following the dimension  self.dim
5     """
6     geo="""SetFactory("OpenCASCADE");
7     h={};
8     dim={};
9     """.format(h, self.dim)
10    if self.dim==2 :
11      if shape == 'Rectangle':
12        geo+="""
13        Rectangle(1) = {0, 0, 0, 1, 1, 0};
14        Characteristic Length{ PointsOf{ Surface{1}; } } = h;
15        Physical Curve("Gamma_D") = {1,2,3,4};
16        Physical Surface("Omega") = {1};
17        """
18      elif shape == 'Disk':
19        geo += """
20        Disk(1) = {0, 0, 0, 1.0};
21        Characteristic Length{ PointsOf{ Surface{1}; } } = h;
22        Physical Curve("Gamma_D") = {1};
23        Physical Surface("Omega") = {1};
24        """
25
26        [. . .]
27
28    with open(filename, 'w') as f:
29      f.write(geo)
```

Extracting the mesh from the geofile:

```
def getMesh(self, filename, h, shape = 'Rectangle',verbose=False):
    """create mesh"""
    import os
    for ext in [".msh",".geo"]:
        f=os.path.splitext(filename)[0]+ext
        if os.path.exists(f):
            os.remove(f)
    if verbose:
        print(f"generate mesh {filename} with h={h} and dimension={self.dim}")
    self.genGeometry(filename=filename,h=h, shape=shape)
    mesh = feelpp.load(feelpp.mesh(dim=self.dim,realdim=self.dim), filename, h)
    return mesh
```

Generated 3D geometry and mesh viewed using gmsh:

.

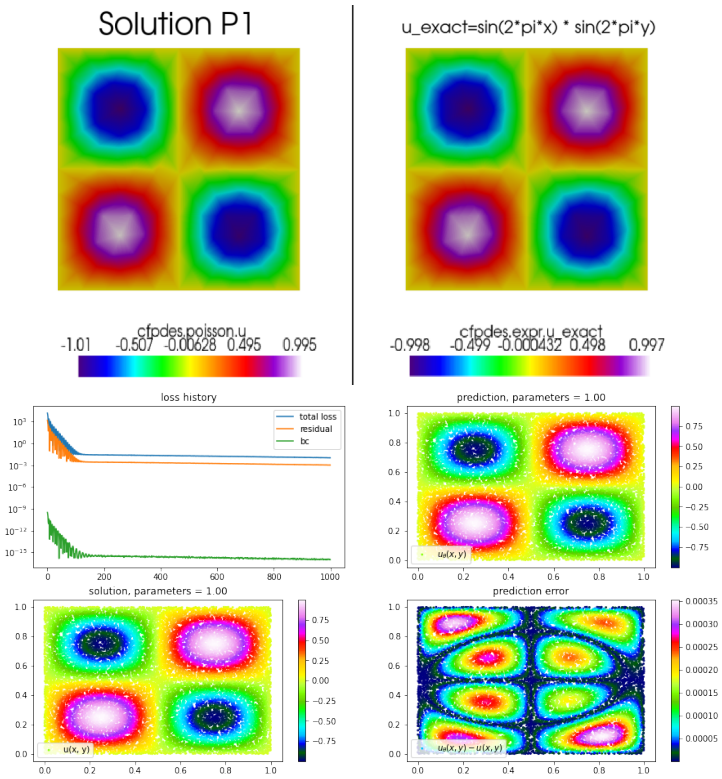


4.2 Comparing the visuals for a Laplacian problem

This segment visualizes the Laplacian problem's solutions on a square domain. Using both the Feel++ and Scimba solvers, we assess the numerical accuracy and visual fidelity of solutions such as $u = \sin(2\pi x)\sin(2\pi y)$, where the right-hand side f complements the exact solution's Laplacian.

```
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 ='scimba', u_exact = u_exact)
```

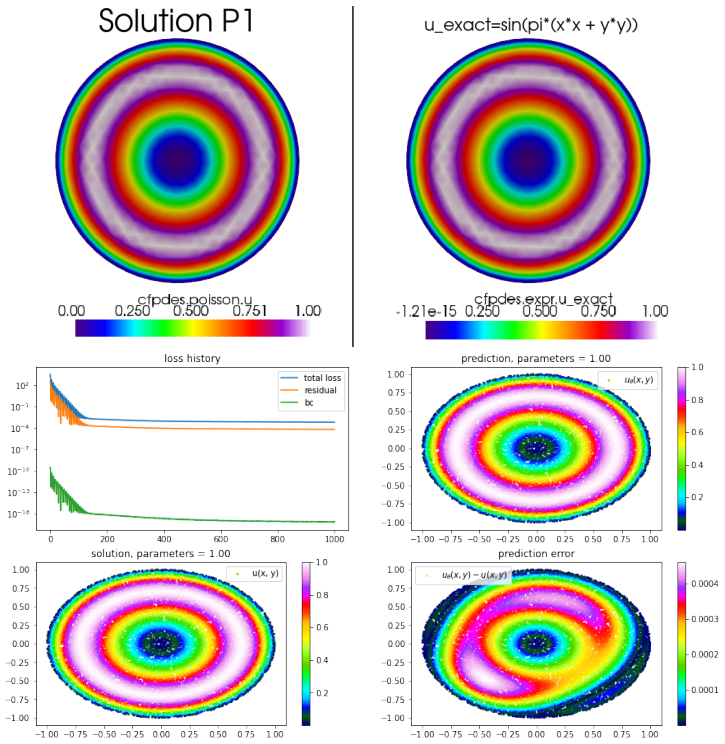
.



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.

```
1
2 u_exact = 'sin(pi*(x*x + y*y))'
3 rhs = '-4*pi*cos(pi*(x*x + y*y)) + 4*pi*pi*(x*x + y*y)*sin(pi*(x
      *x + y*y))'
4
5 P(rhs=rhs, g='0', shape='Disk', solver='scimba', u_exact=u_exact
      )
```

.



4.3 Visualizing solutions on the same graph

This section presents the solution extracted from ScimBa using the Poisson class ScimBa solver. The figure below shows the output from the solver, including information about the meshing process and the extracted solution values.

```
Info    : Reading 'omega-2.geo'...
Info    : Done reading 'omega-2.geo'
Info    : Meshing 1D...
Info    : [  0%] Meshing curve 1 (Line)
Info    : [ 30%] Meshing curve 2 (Line)
Info    : [ 60%] Meshing curve 3 (Line)
Info    : [ 80%] Meshing curve 4 (Line)
Info    : Done meshing 1D (Wall 0.000309616s, CPU 0.000447s)
Info    : Meshing 2D...
Info    : Meshing surface 1 (Plane, Frontal-Delaunay)
Info    : Done meshing 2D (Wall 0.00482482s, CPU 0.004842s)
Info    : 144 nodes 290 elements
Info    : Writing 'omega-2d.msh'...
Info    : Done writing 'omega-2d.msh'
solution =  [[3.73170049]
 [3.61820254]
 [1.16383112]
 [0.53354154]
 [4.15725008]
 [4.48440937]
 [4.71433899]
 [4.84849596]
 [4.88784788]
 [4.83232442]
 [4.68055629]
...
```

Figure 1: Solution from ScimBa visualized on the mesh coordinates.

The screenshot in Figure 1 displays the log information generated during the meshing process and the resulting solution values computed by the ScimBa solver. The log provides the following details:

- Meshing Information: Details about the meshing process, including the types of meshing (1D, 2D) and performance metrics (e.g., CPU time).

- Solution Values: A list of solution values obtained from the solver, corresponding to the mesh coordinates.

4.3.1 Comparing error relative to the theoretical exact solution

We consider the following mathematical system:

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

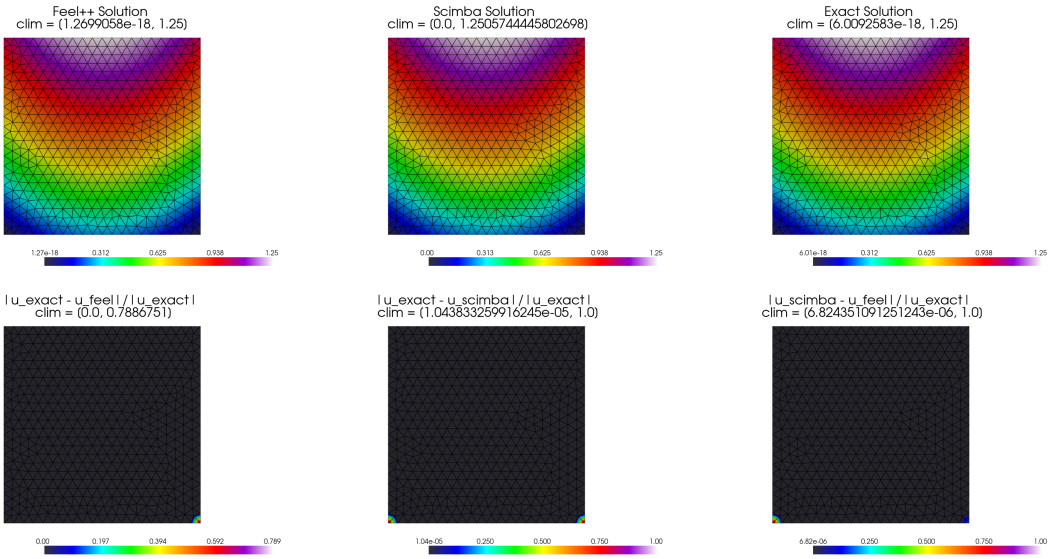
The corresponding Poisson equation with a source term f and boundary condition g is given by:

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

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

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

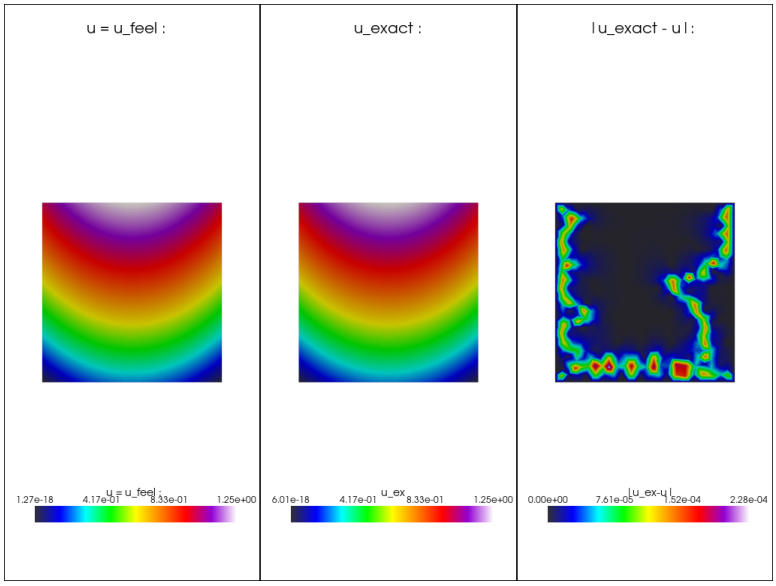
```
u_exact = 'y + (x*(1-x) + y*(1-y)*0.25) '
P(h = 0.05, rhs='2.5', g='y', solver ='scimba', u_exact =
    u_exact)
```



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

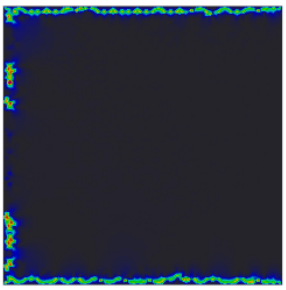


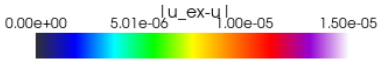
Here we have

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

The differences seem to be spread out randomly.

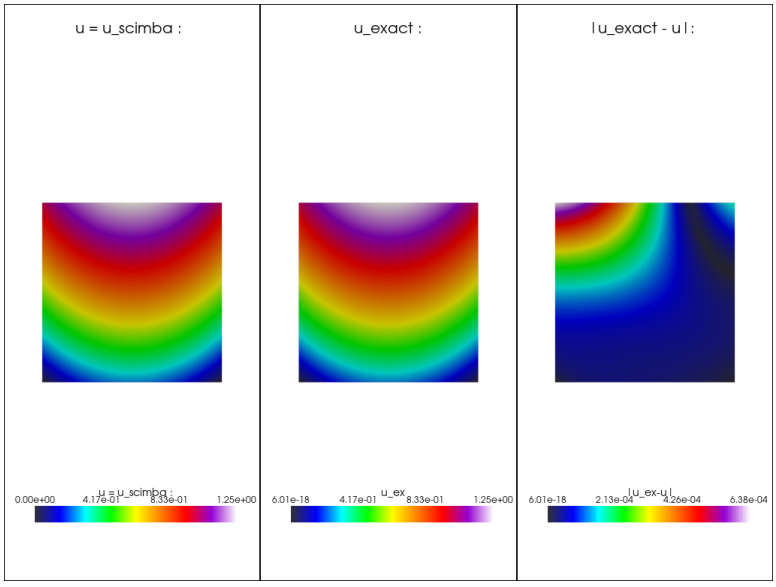
This is what we get with a lower hsize = 0.0125 which amounts to 7554 mesh points:





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

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.

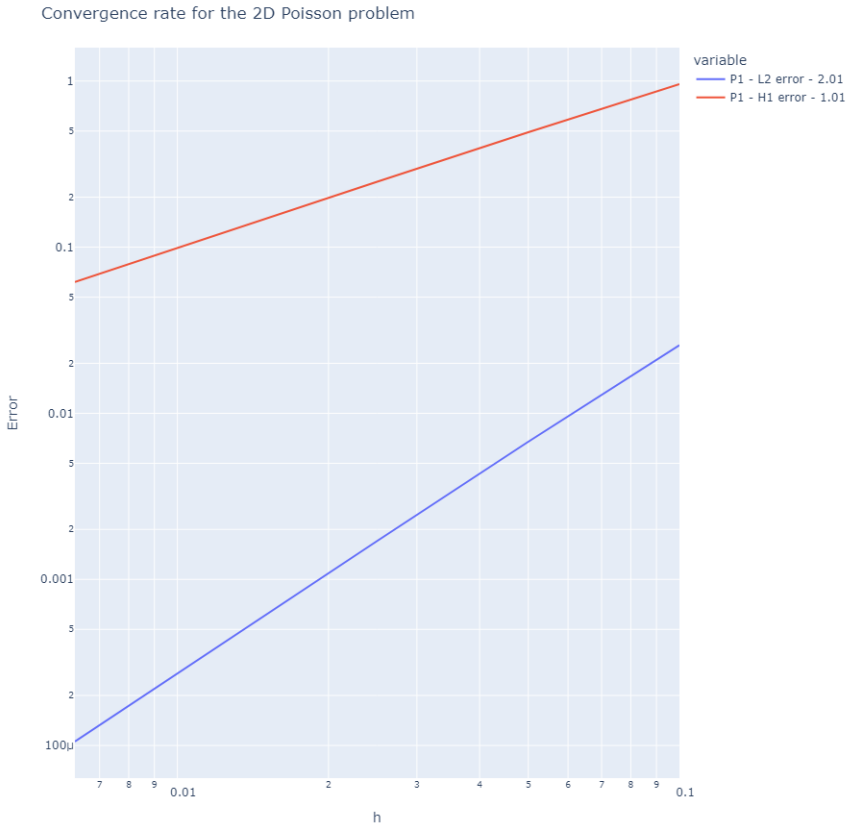


$$\|u_{\text{scimba}} - u_{\text{exact}}\|_\infty = 0.0010454412097140597$$

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. .



The graph in Figure 4.3.3 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

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:

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

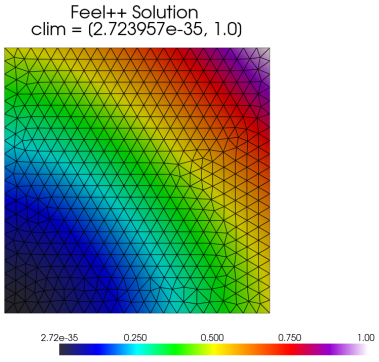
$$D = \begin{pmatrix} 1+x & 0 \\ 0 & 1+y \end{pmatrix}$$

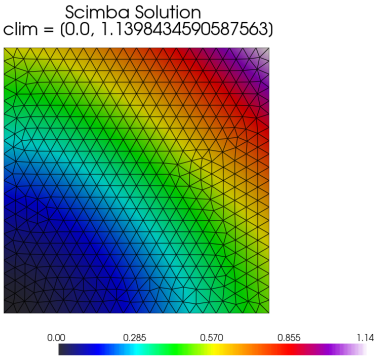
The boundary condition g is given by:

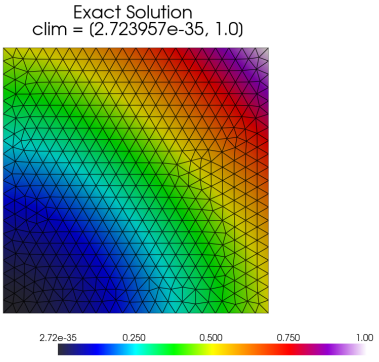
$$u = \frac{x^2}{1+x} + \frac{y^2}{1+y} \quad \text{on} \quad \partial\Omega$$

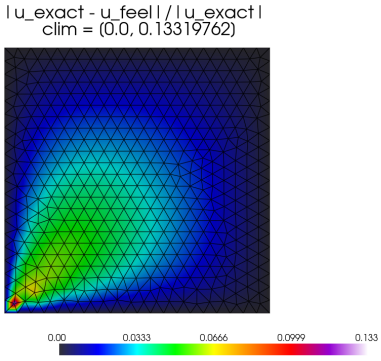
where $\nabla\cdot$ denotes the divergence operator, ∇ denotes the gradient, Ω is the domain, and $\partial\Omega$ is the boundary of the domain.

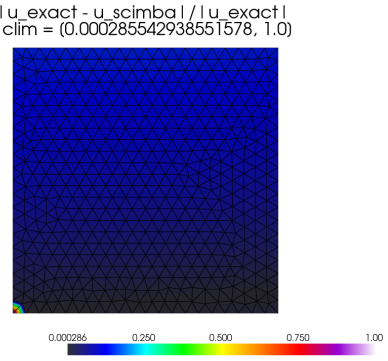
Solving this PDE, we get this:

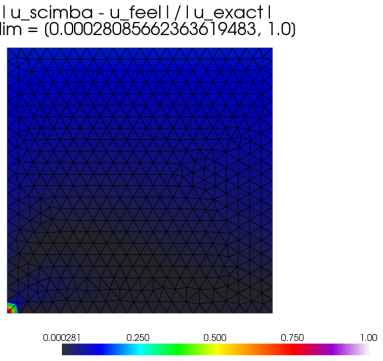






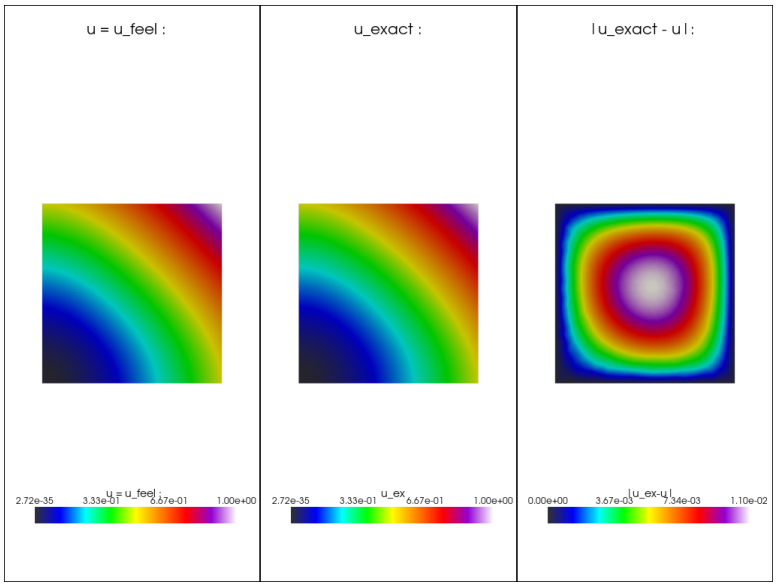






The feelpp solver seems to spread the error out a little more but is of lower order than that of the scimba solver.

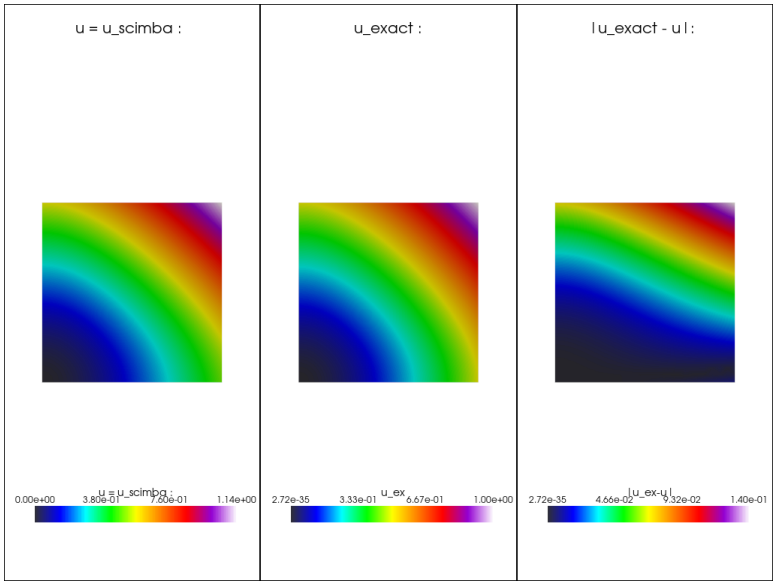
This time the feelpp error seems to be spread out more systemically



Here 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.



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

5 Conclusion

Our ultimate goal is to streamline the process of solving complex mathematical problems and equations by harnessing the combined power of SimBa and Feel++. By integrating these two computational tools, we aim to contribute to two wide-ranged and ongoing projects.

ScimBa offers accessible tools for visualization, computation, and mapping of mathematical models, providing us valuable insights into the behavior of complex systems during our work in this project. Its flexibility and scalability aided us in understanding and interpreting Machine Learning methods.

Nevertheless, Feel++ provides 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 strengths of both tools to solve problems more efficiently.

This integration allowed us to perform comprehensive analyses, visualize results in meaningful ways, and gain deeper insights into both solvers.

The project demonstrates the use of the CFPDE toolbox, leveraging both Feel++ and ScimBa frameworks, for solving Poisson equations across two domains.The methodology involved setting up the environment, defining and solving Poisson problems, and generating visual representations of the results. The following key points summarize the outcomes:

1. Environment Setup:
   (a) The Feel++ environment was initialized with the necessary configuration for using the CFPDE toolbox.
   (b) The Poisson class prototype was effectively used to access and solve problems using the Feel++ solver.
2. Solving Poisson Equations:
   (a) Poisson equations were solved for 2D domains with different parameters, including mesh size, diffusion matrices, and right-hand side functions.
   (b) The solutions were computed using both Feel++ and ScimBa solvers.

3. Visualization:

   (a) Visuals generated using Feel++ displayed the solution on both 2D and 3D geometries, including complex shapes like disks and cubes.

   (b) ScimBa also provided visuals for different domain configurations, such as square and disk-based domains.

Challenges and setbacks:

1. Using time more appropriately with supervisors.
2. Learning to use github in a more efficient way.
3. Limited use examples.
4. Heavy files.

Remaining work:

1. Computing error convergence rates.
2. Extracting solution directly from Scimba.
3. Comparing convergence rates of both solvers.
4. Implement the wrapper on other PDEs, domains in higher dimensions.

.

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