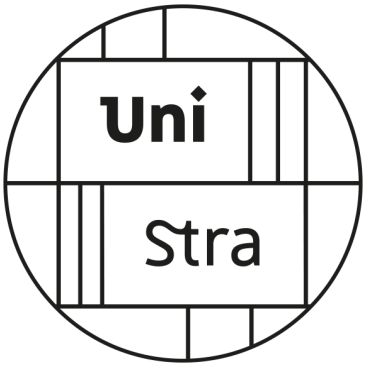


Project report: Coupling ScimBa and Feel++

Helya Amiri[*] & **Rayen Tlili**[†]

Supervised by Christophe Prud'homme, Joubine Aghili

Submitted June 4, 2024

[*]: , helya.amiri@etu.unistra.fr
[†]: , :rayen.tlili@etu.unistra.fr

Contents

Abstract

This project report the details of the integration efforts between ScimBa, emphasizing machine learning, and Feel++, known for its Galerkin methods in PDE solving. The aim is to establish a wrapper between the two libraries combining machine learning with traditional PDE solvers.

Main Content

1 Introduction

A wrapper is a design pattern or a piece of code that serves as an intermediary between other code components. This report presents the objectives, approach, and roadmap for the wrapping of ScimBa and Feel++ libraries.

1.1 Objectives

1. **Streamlined Data Exchange**: Develop a system that streamlines data exchange between ScimBa and Feel++, enabling seamless interaction between the two libraries.
2. **User Empowerment**: Create an interface that allows users to leverage the combined tools of ScimBa and Feel++ effectively.
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.
4. **Add necessary files and dependencies to ScimBa and Feel++**: Contribute to both projects by offering users a platform that unifies machine learning techniques and PDE solving methods to work with and compare.

Once the proper environment has been set up in the docker image, we started working on a program that will solve various PDEs using the Feel++ method and the Scimba method to visualize and compare the results.

1. **Generate multiple results using Feel++ toolboxes**: Using the CFDE toolbox to solve Poisson equations with varying parameters and visualizing them on varying geometries.
2. **Understanding and exporting results using ScimBa**: Explore the ScimBa repository and use examples from Pinns (Physics-informed neural networks) to solve a Poisson equation and visualize the results.
3. **Creating a program that is able to use both as solvers**: One of the primary objectives to reach for this project is to create a program that is able to call upon the Feel++ CFPDE toolbox and the ScimBa machine learning algorithms to solve various PDEs.
4. **Comparing the results of both solvers**: The results of both solvers will be able to be visualized and compared in terms of efficiency and accuracy.
5. **Expand Application Scope**: After successfully solving Poisson equations, expand the application of the program to solve other types of PDEs, further demonstrating the versatility of the integrated system.

6. **Optimize Performance**: Continually optimize the performance of the program, ensuring that it runs efficiently and effectively on various hardware configurations.
7. **User-Friendly Interface**: Develop a user-friendly interface that allows users with varying levels of technical expertise to utilize the program effectively.
8. **Documentation and Training**: Provide comprehensive documentation and training materials to help users understand how to use the program and interpret the results.
9. **Community Engagement**: Engage with the user community to gather feedback, identify areas for improvement, and guide future development efforts.

1.2 Roadmap

.

1. Explore Feel++ and ScimBa documentation.
2. Create a container using docker with a Feel++ base and install ScimBa within it.
3. Solve PDEs using Feel++.
4. Solve PDEs using ScimBa PINNs.
5. Create a Poisson class you can call to solve using Feel++.
6. Add ScimBa as a solver for the class by updating the Poisson2d class to handle ScimBa with parametrized f, g and add a diffusion tensor to the Poisson2d class.
7. Compare the results of both solvers with exact solutions.
8. Compute L^2 and H^1 errors and trace their convergence for both solvers.

2 Exploring Feel++ documentation

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 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**:

2.2 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:
   $u : \Omega \subset \mathbb{R}^d \longrightarrow \mathbb{R}^n$ 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

Figure 1: Shape required by the coefficients

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

3 Getting familiar with ScimBa

3.1 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:

$$\mathcal{N}[\mathbf{u}(\mathbf{x}, t)] = f(\mathbf{x}, t), \tag{3.1}$$

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:

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{data}}(\theta) + \mathcal{L}_{\text{physics}}(\theta), \tag{3.2}$$

with

$$\mathcal{L}_{\text{physics}}(\theta) = \frac{1}{N_f} \sum_{j=1}^{N_f} \left| \mathcal{N}[\mathbf{u}_\theta(\mathbf{x}_j, t_j)] - f(\mathbf{x}_j, t_j) \right|^2 . \tag{3.3}$$

3.2 Scimba PINNs

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.

```
from lap2D_pinns import Run_laplacian2D, Poisson_2D
from scimba.equations import domain


# Define a square domain
xdomain = domain.SpaceDomain(2, domain.SquareDomain(2, [[0.0, 1.0],
                                                         [0.0, 1.0]]))

# Create an instance of the Poisson problem
pde = Poisson_2D(xdomain)

# Run the training
Run_laplacian2D(pde)
```

The class Poisson 2D is initialized with a given spatial domain (space domain) and sets up the

[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.

problem with one unknown variable and one parameter. The parameter domain is narrowly defined, likely to enforce precise boundary conditions or to stabilize the solution.

```
class Poisson_2D(pdes.AbstractPDEx):
    def __init__(self, space_domain):
        super().__init__(
            nb_unknowns=1,
            space_domain=space_domain,
            nb_parameters=1,
            parameter_domain=[[0.50000, 0.500001]],
        )
```

The Run laplacian2D function encapsulates the entire process of setting up, training, and evaluating a neural network to solve the Laplacian PDE using PINN. This includes data sampling, network configuration, loss calculation, and optimization.

```

def Run_laplacian2D(pde, bc_loss_bool=False, w_bc=0, w_res=1.0):
    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)
```

We will talk further about the files and visuals generated in both cases in the "Results" section.

Other neural networks available on ScimBa:

DeepOnet	fix deeponet tx plots
GaussianMixture	move from os to pathlib in most files
Nets	add reference for discontinuous MLP
NeuralGalerkin	fix multi-residual neural Galerkin
Normalizingflows	fix tests; reduce testing time; fix problem in training_txv; fix bc_sampling in x_sampler
NumericalMethods	fix multi-residual neural Galerkin
OdeLearning	push polygonal
Pinns	interfaces conditions

4 Creating the Docker container

4.1 Docker

Docker is a platform designed to simplify the development, deployment, and execution of applications by using containerization technology. Containers self-sufficient units that include everything needed to run a piece of software, including the code, runtime, libraries, and system dependencies. Creating a Docker container and image for the project offers these key advantages:

- **Portability:** Run the project on any platform supporting Docker.
- **Reproducibility:** Recreate the exact same environment whenever needed.
- **Dependency Management:** Package all dependencies within the Docker image and avoid conflicts with other software on the host system.

Docker is widely used as it provides consistent and reproducible development and production environments.

4.2 Dockerfile

We both worked on creating the Dockerfile adding little by little the dependencies.

```
# Start with the Feel++ base image
FROM ghcr.io/feelpp/feelpp:jammy

# Install system dependencies
RUN apt-get update && apt-get install -y \
    git \
   xvfb
# Install Python libraries
RUN pip3 install torch xvfbwrapper pyvista plotly panel ipykernel
    matplotlib

# Clone the Scimba repository
RUN git clone https://gitlab.inria.fr/scimba/scimba.git
                    /workspaces/2024-m1-scimba-feelpp/scimba

# Install Scimba and its dependencies
WORKDIR /workspaces/2024-m1-scimba-feelpp/scimba
RUN pip3 install scimba

# Copy the xvfb script into the container
COPY tools/load_xvfb.sh /usr/local/bin/load_xvfb.sh
RUN chmod +x /usr/local/bin/load_xvfb.sh

# Set the script to initialize the environment
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. It copies the public ScimBa repository into the 'scimba' folder and installs it. We have also added command lines to automate script that let us run the program 'solve lap.py', that uses Feel++ libraries to solve a Laplacian problem and generates visuals.

5 Methodology

We decided to divide the work. Helya works on making a Feel++ solver and Rayen on a ScimBa one. Given the Feel++ documentation and the Poisson class prototype that gives access to results from the Feel++ solver. Here's the work we did:

5.1 Solving the PDE

First, we need to create the right environment for using the CFPDE toolbox. This involves importing necessary libraries, setting up command-line arguments, and initializing the Feel++ environment with specific options and configurations.

Create the right environment for using the CFPDE toolbox:

```
import sys
import feelpp
import feelpp.toolboxes.core as tb

from tools.solvers import Poisson
sys.argv = ["feelpp_app"]
e = feelpp.Environment(sys.argv,
                       opts=tb.toolboxes_options("coefficient-form-pdes",
                       "cfpdes"),
                       config=feelpp.globalRepository('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 'tools.solvers'. The command-line arguments for the Feel++ environment are set, and the environment is initialized with specific options and configuration for coefficient-form PDEs.

We have set up to solving the Poisson equation with different parameters using the CFPDE toolbox by creating a Poisson Class that takes the following arguments:

```
P = Poisson(dim = 2)
P(h=0.08,  rhs='-1.0-1*y*x+y*y', g='0', order=1, geofile='geo/disk.geo',
    plot='2d.png')
P(h=0.1,  rhs='-1.0-2*y*x+y*y', g='0', order=1, plot='f2.png')

P = Poisson(dim = 2)
P(h=0.1, diff='{1.0,0,0,x*y}', rhs='1', plot='d1.png')
P(h=0.1, diff='{1+x,0,0,1+y}', rhs='1', plot='d2.png')

P = Poisson(dim = 3)
P(h=0.08, diff='{1,0,0,0,x+1,0,0,0,1+x*y}', g = 'x', rhs='x*y*z',
geofile = 'geo/cube.geo', plot='3d.png')
```

In the two first examples, we create a 'Poisson' solver object for a 2D problem with a specific right-hand side expression and geometry file, and save the output plot. We then solve two

more 2D problems with different diffusion coefficients and save the plots. Finally, we create a ‘Poisson‘ solver object for a 3D problem with a specified diffusion matrix, boundary condition, and geometry fileg’.

Lastly, we demonstrate how to add a solver flag when calling the ‘Poisson‘ class. This allows us to specify which solver to use (either ‘feelpp‘ or ‘scimba‘).

Adding solver flag when calling the Poisson Class:

```
def __call__(self,
             h,                                          # mesh size
             order=1,                                    # polynomial order
             name='Potential',                           # name of the variable
             rhs='8*pi*pi*sin(2*pi*x)*sin(2*pi*y)',      # right hand side
             diff='{1,0,0,1}',                           # diffusion matrix
             g='0',
             geofile=None,
             plot=None,
             solver='feelpp'):                           # or solver ='scimba'
    """
```

In the code above, we define the ”call” method for the ”Poisson” class with parameters for mesh size, polynomial order, variable name, right-hand side expression, diffusion matrix, boundary conditions, geometry file, plot filename, and solver flag. The solver flag can be either ”feelpp” or ”scimba”, indicating which solver to use for the problem.

Solving using Feel++ and ScimBa

```
P( rhs='-1.0-4*y*x+y*y', g='x', order=1, solver='feelpp')
P( rhs='-1.0-4*y*x+y*y', g='x', order=1, solver ='scimba')
```

5.2 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, filename, h, json, dim=2, verbose=False):
    if verbose:
        print(f"Solving the laplacian problem for hsize = {h}...")

    diff = self.diff.replace('{', '(').replace('}', ')')
    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)
    network, pde = Run_laplacian2D(pde)

    # Extract solution function u
    u = network.forward

    # Get mesh points
    geo_to_msh(filename, f"omega_{dim}d.msh", mesh_size=h)
    mesh = f"omega_{dim}d.msh"
    my_mesh = mesh2d(mesh)
    my_mesh.read_mesh()
    coordinates = my_mesh.Nodes

    # Evaluate the network on mesh points
    input_tensor = torch.tensor(coordinates, dtype=torch.double)
    mu = torch.tensor([[0.5]]).repeat(input_tensor.size(0), 1)
    solution_tensor = u(input_tensor, mu)

    # Convert the tensor to a NumPy array
    solution_array = solution_tensor.detach().numpy()

    # Update model with the solution array if needed
    self.model["PostProcess"][f"cfpdes_{self.dim}d_p{self.order}"]["Exports"]
    ["expr"]["u"] = solution_array

    return solution_array
```

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.
- Reads the mesh points from the specified mesh file.
- Evaluates the network on the mesh points and converts the solution tensor to a NumPy

array.

- Updates the model with the extracted solution array.

To read the mesh and evaluate the solution on the coordinates, we use the `mesh2d` class as shown below:

```
class mesh2d:
    def __init__(self, filename):
        self.filename = filename
        self.Nnodes = 0
        self.Nodes = None

    def read_mesh(self):
        with open(self.filename, 'r') as f:
            line = f.readline()
            while line:
                if line.startswith('$Nodes'):
                    self.read_nodes(f)
                if line.startswith('$Elements'):
                    self.read_elements(f)

                line = f.readline()
    [...]
```

The `mesh2d` class is responsible for reading the mesh file and extracting the node coordinates. It initializes with the filename and reads the mesh data, including nodes and elements.

By combining the `scimba_solver` method and the `mesh2d` class, we can effectively train the Poisson function with ScimBa, extract the solution, and evaluate it on the mesh coordinates. We are now working on visualizing the solution generated by ScimBa.

5.3 Computing L2 and H1 errors

5.3.1 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.

```
import plotly.express as px
from plotly.subplots import make_subplots
import itertools
import pandas as pd
import numpy as np

def runLaplacianPk(df,model,verbose=False):
    """generate the Pk case

    Args:
        order (int, optional): order of the basis. Defaults to 1.
    """
    meas=dict()
    dim,order,json=model
    for h in df['h']:
        m=laplacian(hsize=h,json=json,dim=dim,verbose=verbose)
        for norm in ['L2','H1']:
            meas.setdefault(f'P{order}-Norm_laplace_{norm}-error', [])
            meas[f'P{order}-Norm_laplace_{norm}-error'].append(m.pop(
                                f'Norm_laplace_{norm}-error'))
    df=df.assign(**meas)
    for norm in ['L2','H1']:
        df[f'P{order}-laplace_{norm}-convergence-rate']=
            np.log2(df[f'P{order}-Norm_laplace_{norm}-error'].shift() /
                df[f'P{order}-Norm_laplace_{norm}-error']) /
                np.log2(df['h'].shift() / df['h'])
    return df
```

5.3.2 Plotting the convergence rate

Run the convergence analysis for different mesh sizes and polynomial orders. Generate and display a plot of convergence rates across mesh sizes for each polynomial order.

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

df=runConvergenceAnalysis(json=laplacian_json,dim=2,verbose=True)
def plot_convergence(df,dim,orders=[1,2]):
    fig=px.line(df, x="h", y=
        [f'P{order}-Norm_laplace_{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}-laplace_{norm}-convergence-rate"].iloc[-1]:.2f}',
                selector=dict(name=f'P{order}-Norm_laplace_{norm}-error'))
    fig.update_layout(
            title=f"Convergence rate for the {dim}D Laplacian problem",
            autosize=False,
            width=900,
            height=900,
        )
    return fig
fig=plot_convergence(df,dim=2)
fig.show()
```

6 Results

6.1 Generating visuals using Feel++

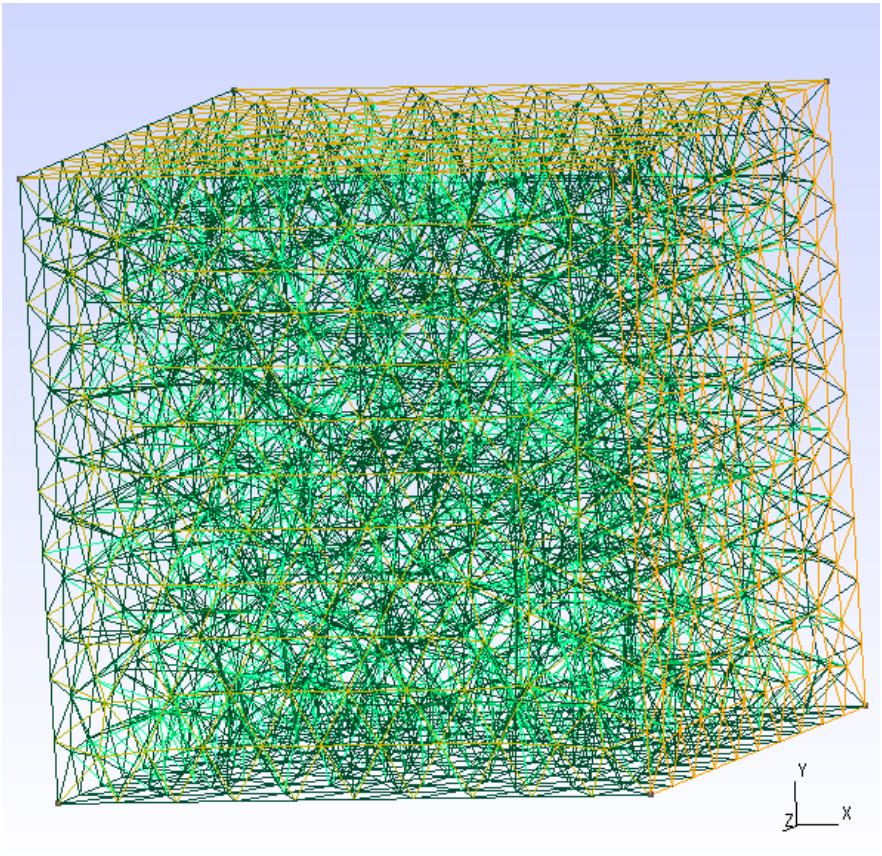
Feel++ produces geometry files for either a 2D rectangle or a 3D box. The generated file is compatible with Gmsh, facilitating subsequent mesh generation and finite element analysis. The characteristic length h controls the mesh resolution, and we define physical groups for boundaries and domains, which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

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

```
def genCube(self, filename, h=0.1):
    """
    Generate a cube geometry following the dimension  self.dim
    """

    geo="""SetFactory("OpenCASCADE");
    h={};
    dim={};
    """.format(h, self.dim)

    if self.dim==2 :
        geo+="""
        Rectangle(1) = {0, 0, 0, 1, 1, 0};
        Characteristic Length{ PointsOf{ Surface{1}; } } = h;
        Physical Curve("Gamma_D") = {1,2,3,4};
        Physical Surface("Omega") = {1};
        """

    elif self.dim==3 :
        geo+="""
        Box(1) = {0, 0, 0, 1, 1, 1};
        Characteristic Length{ PointsOf{ Volume{1}; } } = h;
        Physical Surface("Gamma_D") = {1,2,3,4,5,6};
        Physical Volume("Omega") = {1};
        """
    with open(filename, 'w') as f:
        f.write(geo
```

Generated 3D geometry and mesh viewed using gmsh:

.

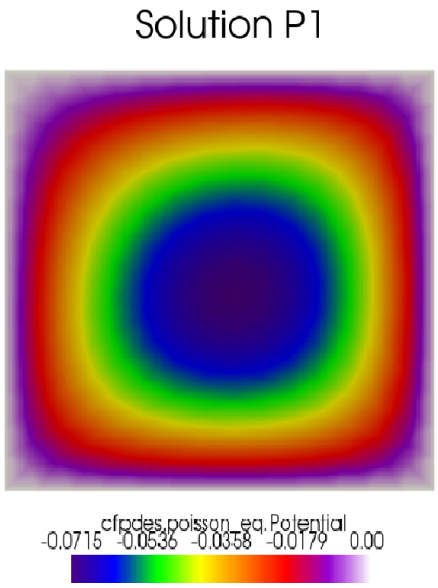


Now, we illustrate the usage of the Poisson class for conducting finite element analysis, specifically focusing on solving the Poisson equation. We initiate the Poisson class instance P by specifying the dimension as 2:

```
    P = Poisson(dim = 2)
    P(h=0.08,  rhs='-1.0-1*y*x+y*y', g='0', order=1, plot='f4.png')
```

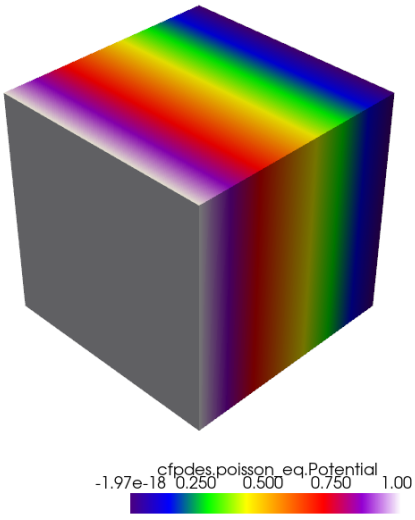
Using the Poisson class, we solve the Poisson equation for a 2D area. Customizable parameters let us adapt the solver to different problems.

.




```
    P = Poisson(dim = 3)
    P(h=0.08, diff='{1,0,0,0,x+1,0,0,0,1+x*y}', g = 'x', rhs='x*y*z',
        geofile = 'geo/cube.geo', plot='3d.png')
```

.



6.2 Generating visuals using ScimBa

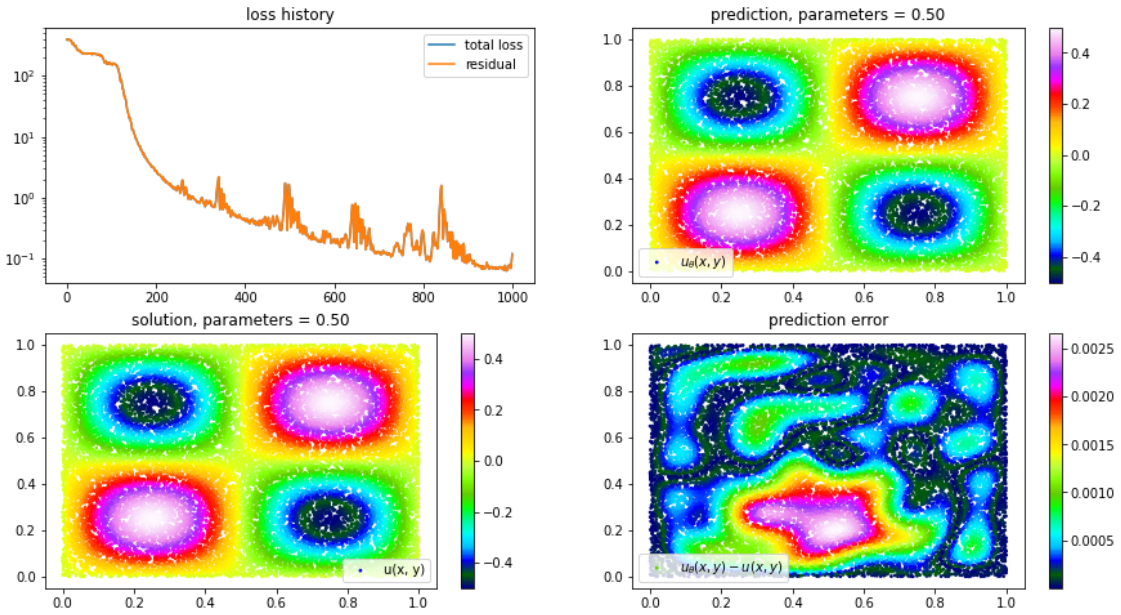
We begin by defining the spatial domain `xdomain` using ScimBa's SpaceDomain module. In this case, we specify a two-dimensional domain using a square domain configuration spanning from $(0.0, 0.0)$ to $(1.0, 1.0)$;

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:

Finally, we execute the `Run_laplacian2D` function, which solves the Poisson equation defined by `pde`:

```
xdomain = domain.SpaceDomain(2, domain.SquareDomain(2, [[0.0, 1.0],
                                                        [0.0, 1.0]]))
pde = Poisson_2D(xdomain, rhs='8*pi*pi*sin(2*pi*x)*sin(2*pi*y)', g='0')
Run_laplacian2D(pde)
```

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.

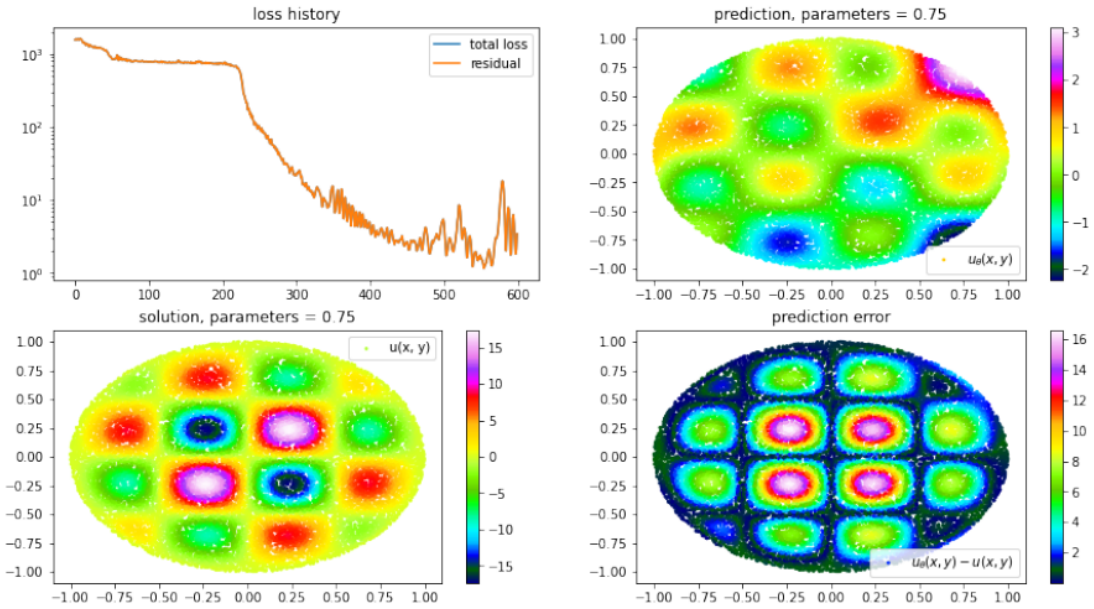
Subsequently, we create an instance of the Poisson equation `pde_disk` in two dimensions, utilizing the disk-based spatial domain defined earlier.

Finally, we execute the `Run_laplacian2D` function, which solves the Poisson equation defined by `pde_disk`:

```
    xdomain = domain.SpaceDomain(2, domain.DiskBasedDomain(2, center=[0.0, 0.0],
                                                            radius=1.0))
    pde_disk = PoissonDisk2D(space_domain=xdomain)
    Run_laplacian2D(pde_disk)
```

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

.



6.2.1 Laplacian on ellipse mapping

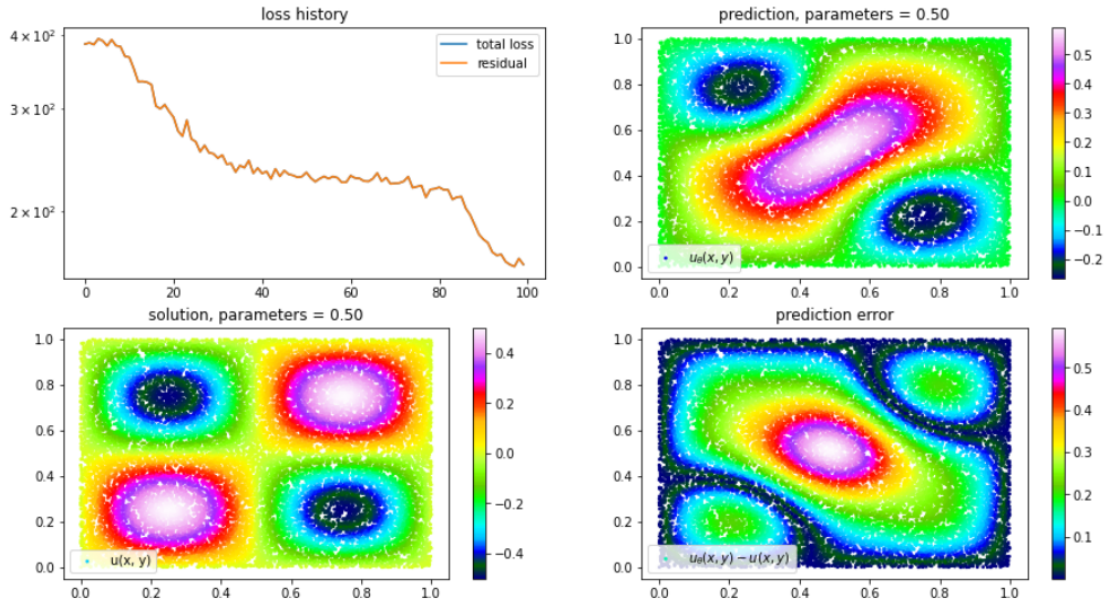
We initiate the computation and mapping process by defining the spatial domain `xdomain` using ScimBa's SpaceDomain module. In this instance, we specify a two-dimensional domain utilizing a disk-based configuration with a center at $(0.0, 0.0)$ and a radius of 1.0. Additionally, we provide a custom mapping function `disk_to_ellipse` and its corresponding Jacobian function `Jacobian_disk_to_ellipse`.

Next, we create an instance of the Poisson equation `pde` in two dimensions, utilizing the ellipse-based spatial domain defined earlier.

Finally, we execute the `Run_laplacian2D` function, which computes the Laplacian on the ellipse and maps it using a neural network. Additionally, we set parameters `bc_loss_bool=True` to include boundary condition loss and `w_bc=10` and `w_res=0.1` to control the weight of the boundary condition and residual loss:

```
    xdomain = domain.SpaceDomain(
        2,
        domain.DiskBasedDomain(
            2,
            [0.0, 0.0],
            1.0,
            mapping=disk_to_ellipse,
            Jacobian=Jacobian_disk_to_ellipse,
        ),
    )
    pde = Poisson_2D_ellipse(xdomain)
    Run_laplacian2D(pde, bc_loss_bool=True, w_bc=10, w_res=0.1)
```

.



6.2.2 Laplacian on potato mapping

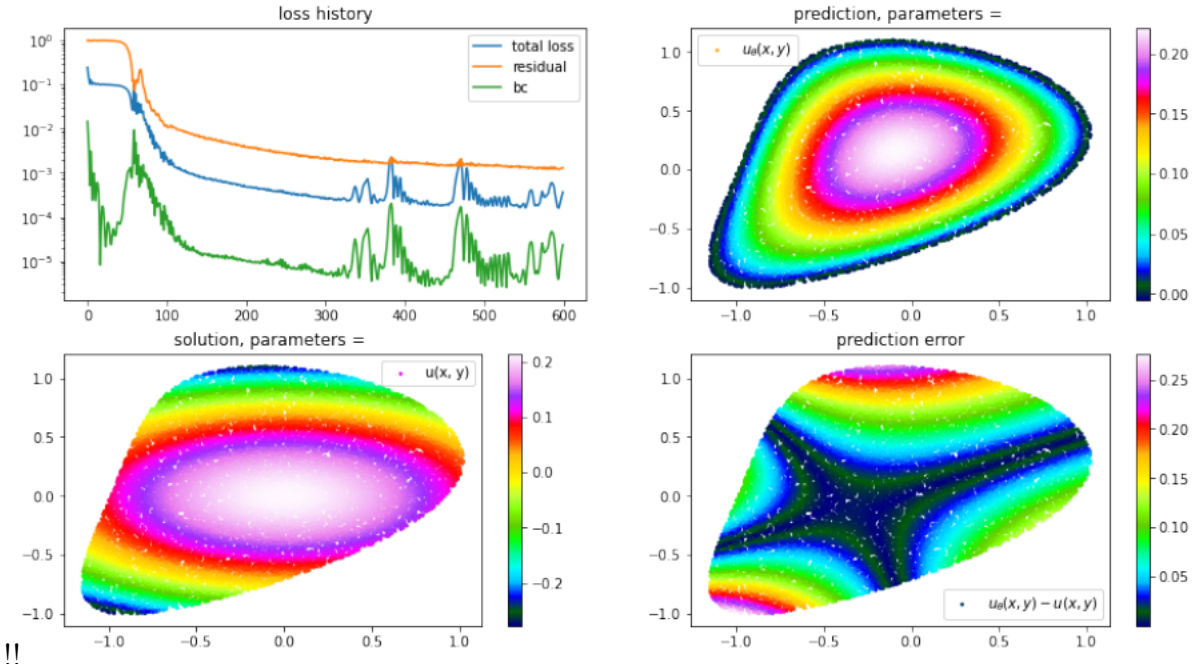
We also provide a custom mapping function `disk_to_potato` and its corresponding Jacobian function `Jacobian_disk_to_potato`.

Next, we create an instance of the Poisson equation `pde` in two dimensions, utilizing the potato-shaped spatial domain defined earlier.

Finally, we execute the `Run_laplacian2D` function, which computes the Laplacian on the potato-shaped domain and maps it using a neural network. We set parameters `bc_loss_bool=True` to include boundary condition loss and `w_bc=10` and `w_res=0.1` to control the weight of the boundary condition and residual loss.

```
    xdomain = domain.SpaceDomain(
        2,
        domain.DiskBasedDomain(
            2,
            [0.0, 0.0],
            1.0,
            mapping=disk_to_potato,
            Jacobian=Jacobian_disk_to_potato,
        ),
    )
    pde = Poisson_2D_ellipse(xdomain)
    Run_laplacian2D(pde, bc_loss_bool=True, w_bc=10, w_res=0.1)
```

.



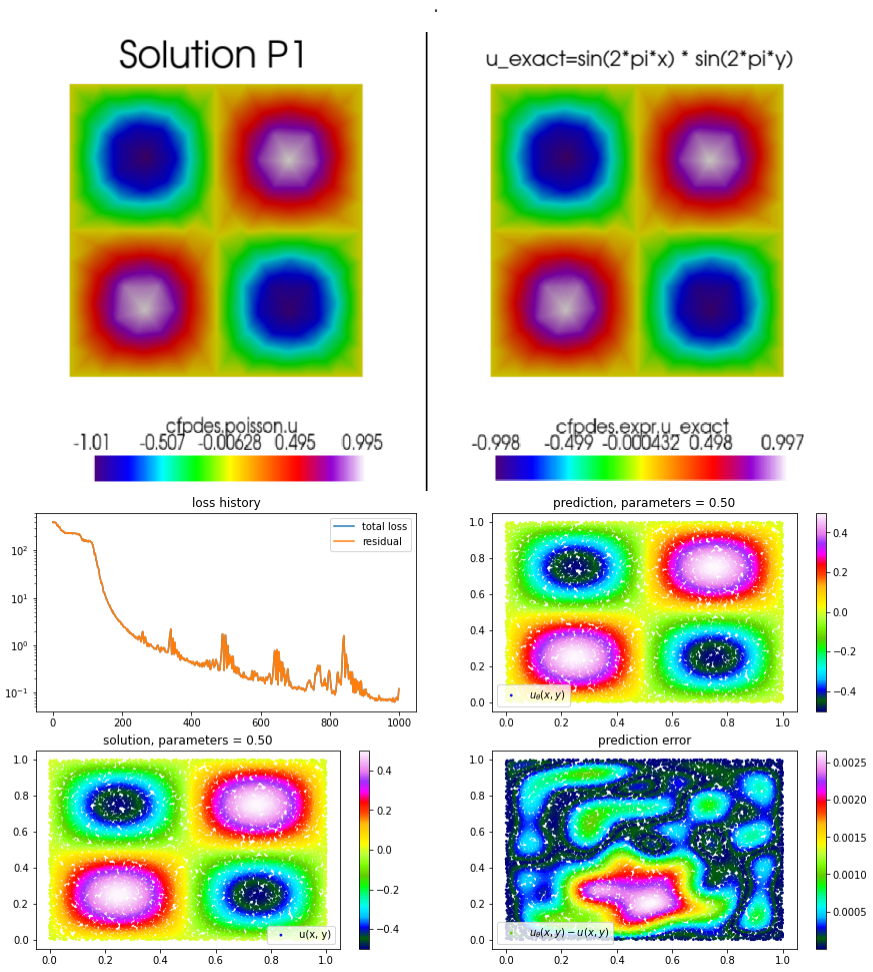
!!

This process allows us to analyze and visualize the behavior of the Laplacian on complex geometries, such as potato shapes, with the aid of neural networks. By including boundary condition loss and adjusting weights, we enhance the accuracy and control of the mapping process. This highlights the potential of combining numerical methods with machine learning techniques for solving and analyzing mathematical problems in scientific computing.

6.3 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.

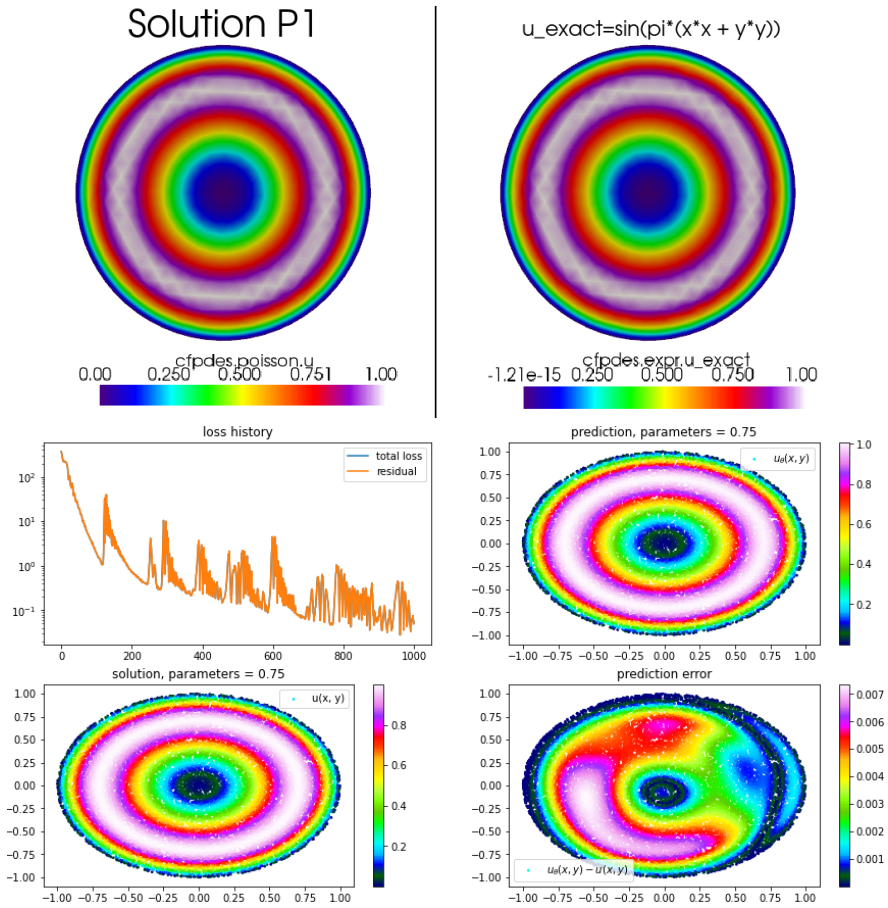
```
# 2D on different domains
P = Poisson(dim = 2)

# for square domain
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', order=1, plot='f2.png', u_exact = u_exact)
P(rhs=rhs, g='0', order=1, 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 # for disk domain
3 u_exact =  'sin(pi*(x*x + y*y))'
4 rhs = '4*pi*sin(pi*(x*x + y*y)) - 4*pi*pi*(x*x + y*y)*cos(pi*(x*x + y*y))'
5
6 P(rhs=rhs, g='0', order=1, geofile='geo/disk.geo', plot='2d.png',
7     u_exact = u_exact)
8 P(rhs=rhs, g='0', order=1, geofile='geo/disk.geo', solver='scimba',
9     u_exact = u_exact)
```

.



6.4 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]
...
```

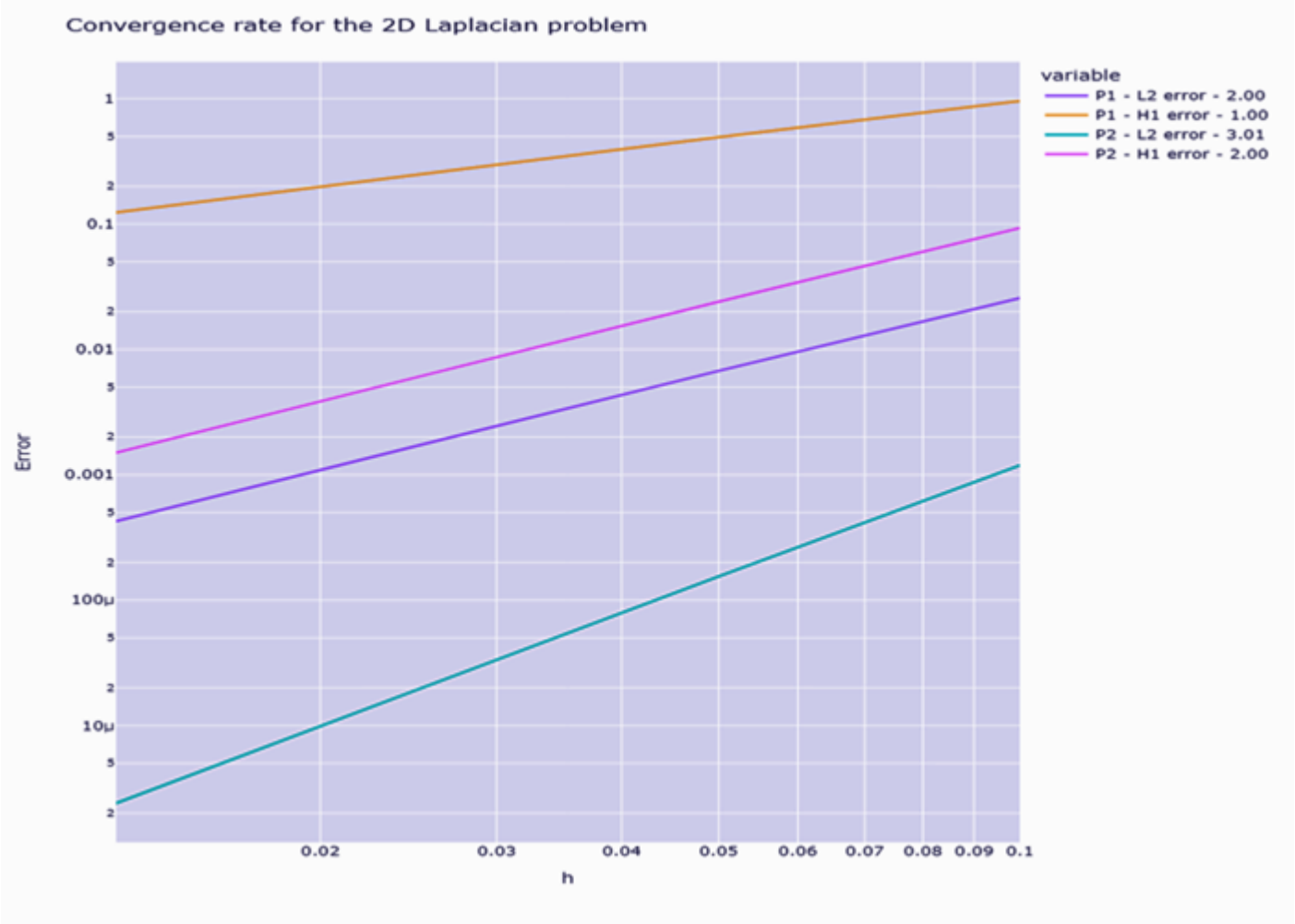
Figure 2: Solution from ScimBa visualized on the mesh coordinates.

The screenshot in Figure 2 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.

6.5 Error convergence rate

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



!!

The graph in Figure 6.5 displays the convergence rates of L2 and H1 errors for the 2D Laplacian problem using different polynomial orders (P1 and P2). 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.

```
def plot_convergence(df,dim,orders=[1,2]):
    [...]
    for order,norm in list(itertools.product(orders,['L2','H1'])):
        fig.update_traces(name=
        f'P{order} -
        {norm} error -
        {df[f"P{order}-laplace_{norm}-convergence-rate"].iloc[-1]:.2f}',
        selector=dict(name=f'P{order}-Norm_laplace_{norm}-error'))
        [...]
```

Specifically, the convergence rates for P1 and P2 are shown for both L2 and H1 norms, indicating the rate at which the numerical solution approaches the exact solution as the mesh is refined.

7 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

[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`