Project report: Coupling ScimBa and Feel++

Helya Amiri* & **Rayen Tlili†**

Supervised by Christophe Prud'homme, Joubine Aghili

Submitted May 27, 2024

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

Contents

Abstract

This project report details the integration efforts between ScimBa, emphasizing machine learning, and Feel++, known for its Galerkin methods in PDE solving. The aim is to establish seamless compatibility between the two libraries, fostering advanced computational techniques in scientific research. By combining machine learning with traditional PDE solvers, the project endeavors to propel computational science and engineering forward, enabling efficient data exchange and methodological synergy.

Main Content

1 Introduction

This report presents the objectives, approach, and roadmap for the coupling of ScimBa and Feel++ libraries. ScimBa is a project aimed at integrating machine learning techniques with traditional scientific computing methods, while Feel++ is a C++ implementation of Galerkin methods for solving partial differential equations (PDEs). The coupling of these two libraries is expected to enhance their capabilities and enable researchers to solve complex scientific problems more effectively.

1.1 Objectives

The primary objective of this project is the coupling of ScimBa and Feel++, which encompasses:

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 docker container with a Feel++ base.**
3. **Install dependencies and clone Scimba repository in docker image.**
4. **Enable image generation in docker.**
5. **Create a devcontainer.**
6. **Solve PDEs using Feel++.**
7. **Solve PDEs using ScimBa using PINNS.**
8. **Create a Poisson class you can call to solve using Feel++.**
9. **Adding ScimBa as a solver for the class.**
10. **Update the Poisson2d class to handle ScimBa with parametrized f and g.**
11. **Adding a diffusion tensor to the Poisson2d class.**
12. **Solve different types of PDEs with both solvers with the same parameters.**
13. **Compare the results of both solvers.**

2 Exploring Feel++ documentation

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

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 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

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.

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.

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

# Set labels for metadata
LABEL maintainer="Helya Amiri <helya.amiri@etu.unistra.fr>,
                  Rayen Tlili <rayen.tlili@etu.unistra.fr>"
LABEL description="Docker image with Feel++, Scimba, and PyTorch."

USER root

# 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.

5 Methodology

Given the Feel++ documentation and the Poisson class prototype that gives access to results from the Feel++ solver.

5.1 Solving the PDE

Create the right environment for using the CFPDE toolbox:

```python
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'))
```

Solving the Poisson equation with different parameters using the CFPDE toolbox

```python
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')
```

Adding solver flag when calling the Poisson Class.

```python
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'
    """
```

Solving using Feel++ and ScimBa

```
    if solver == 'feelpp':
      print(f"Solving the laplacian problem for hsize = {h}...")
      feelpp_mesh = feelpp.load(feelpp.mesh(dim=self.dim, realdim=self.dim),
                                              fn, h)
      self.pb.setMesh(feelpp_mesh)
      self.pb.setModelProperties(self.model)
      self.pb.init(buildModelAlgebraicFactory=True)
      self.pb.printAndSaveInfo()
      self.pb.solve()
      self.pb.exportResults()

    elif solver == 'scimba':
      print("Solving using Scimba")

      # Define a disk domain

      if geofile == 'geo/disk.geo' :
        xdomain = domain.SpaceDomain(2,
                                    domain.DiskBasedDomain(2,
                                    center=[0.0, 0.0], radius=1.0))
        pde_disk = PoissonDisk2D(xdomain,  rhs= rhs, g= g)
        Run_laplacian2D(pde_disk)

      # Define a square domain

      elif geofile == None:
        xdomain = domain.SpaceDomain(2,
                                    domain.SquareDomain(2,
                                    [[0.0, 1.0], [0.0, 1.0]]))
        pde = Poisson_2D(xdomain,  rhs= rhs, g= g)
        Run_laplacian2D(pde)
```

Solving using ScimBa:

```
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', solver='scimba')
```

If new training = False, the code suggests that you might want to continue using a previously trained and saved model without starting the training from scratch. If new training = True, it indicates that you want to start fresh, ignoring any previously saved models.

```
new_training = False
#new_training = True
if new_training:
    (
        Path.cwd()
        / Path(training_x.TrainerPINNSpace.FOLDER_FOR_SAVED_NETWORKS)
        / file_name
    ).unlink(missing_ok=True)
```

5.2 Computing L2 and H1 errors

5.2.1 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.

```
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
```

5.2.2 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++

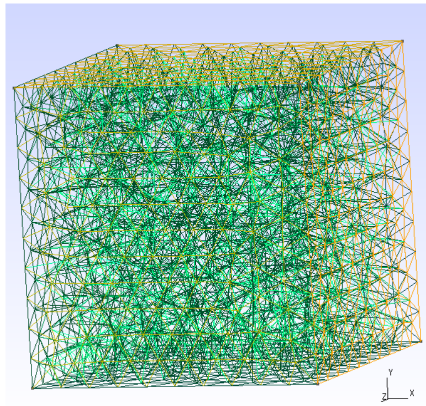
Feel++ produces geometry files for either a 2D rectangle or a 3D box. The generated file is compatible with Gmsh, facilitating subsequent mesh generation and finite element analysis. The characteristic length h controls the mesh resolution, and we define physical groups for boundaries and domains, which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

.



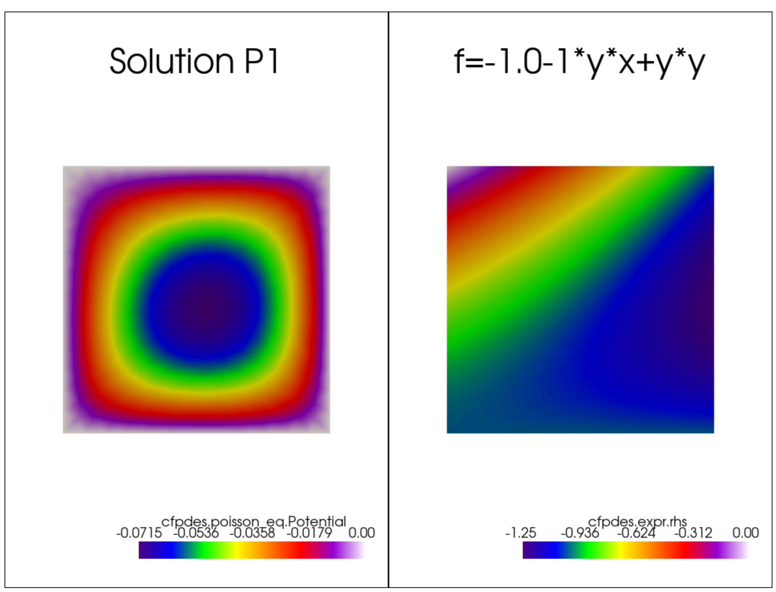
we illustrate the usage of the Poisson class for conducting finite element analysis, specifically focusing on solving the Poisson equation. The code snippet provided demonstrates the invocation of the Poisson class with specified parameters.

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')
```

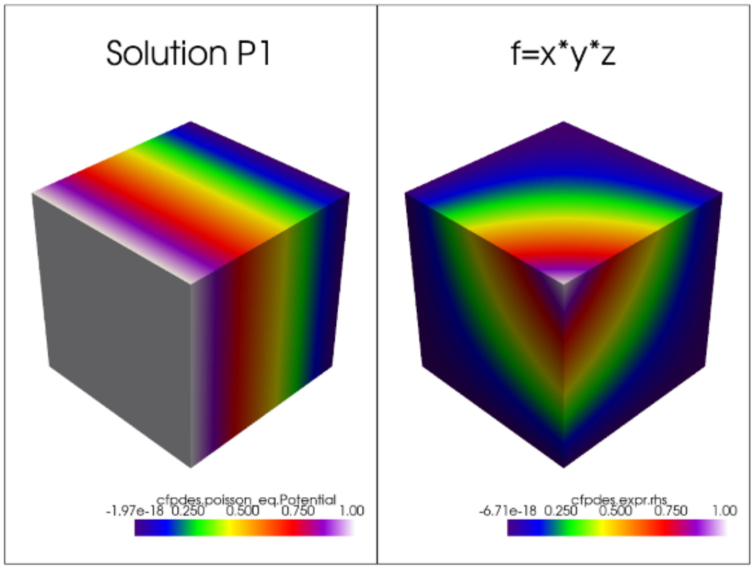
Using the Poisson class, we solve the Poisson equation for a 2D area. Customizable parameters let us adapt the solver to different problems. This method helps analyze many phenomena governed by the Poisson equation, useful in science and engineering.

.



```
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 $(1.0, 1.0)$;

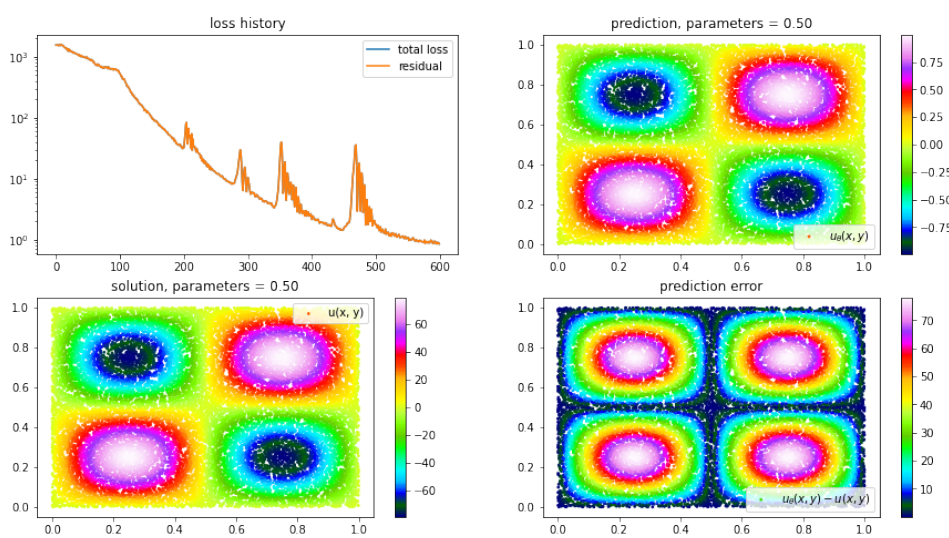
Next, we create an instance of the Poisson equation `pde` in two dimensions, specifying the right-hand side (`rhs`) as well as the boundary condition function:

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.

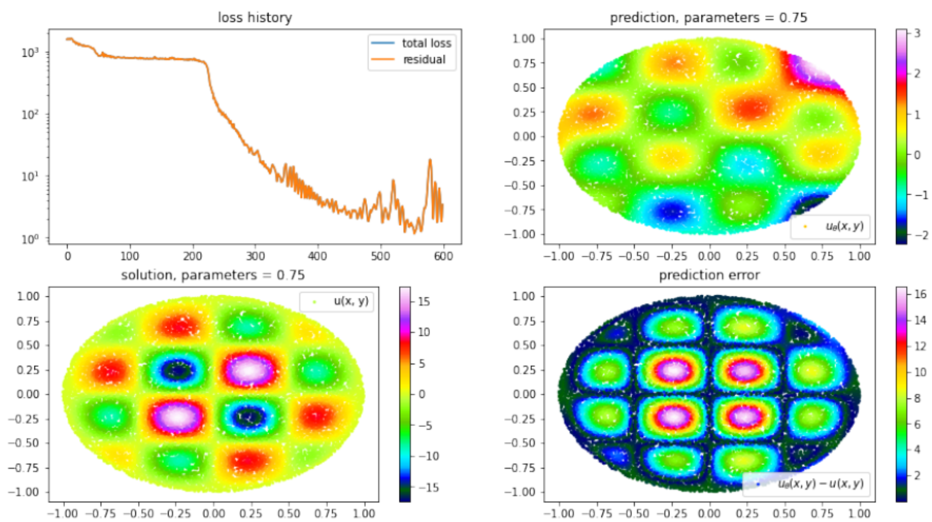
Subsequently, we create an instance of the Poisson equation `pde_disk` in two dimensions, utilizing the disk-based spatial domain defined earlier.

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

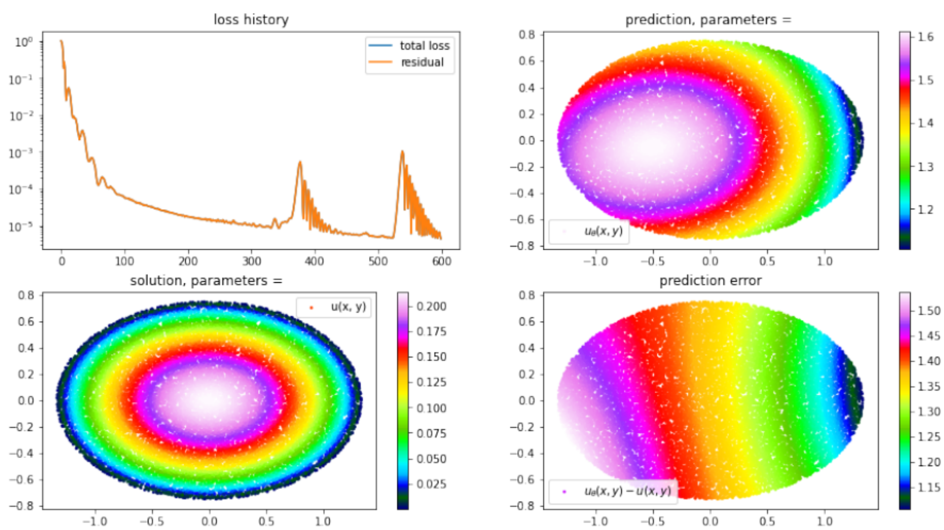
We initiate the computation and mapping process by defining the spatial domain `xdomain` using ScimBa's SpaceDomain module. In this instance, we specify a two-dimensional domain utilizing a disk-based configuration with a center at $(0.0, 0.0)$ and a radius of 1.0. Additionally, we provide a custom mapping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

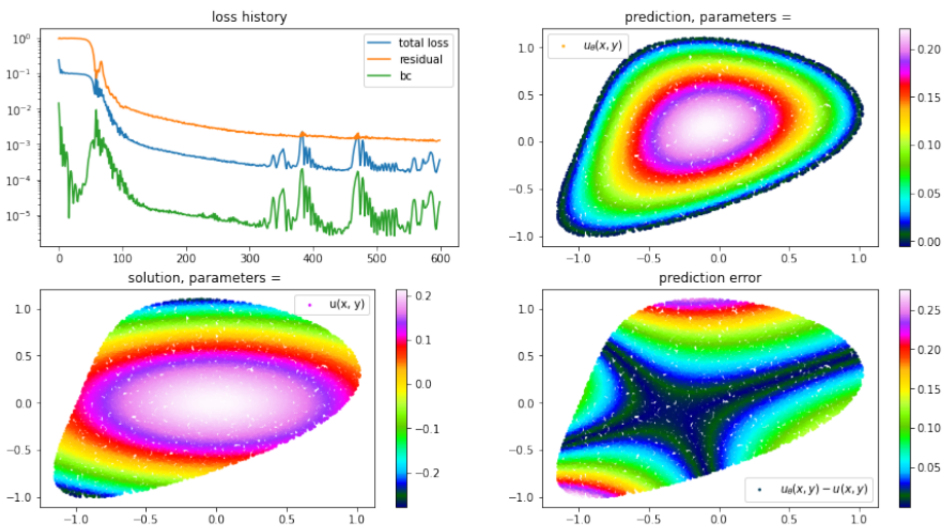
We also provide a custom mapping function `disk_to_potato` and its corresponding Jacobian function `Jacobian_disk_to_potato`.

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)
```

.



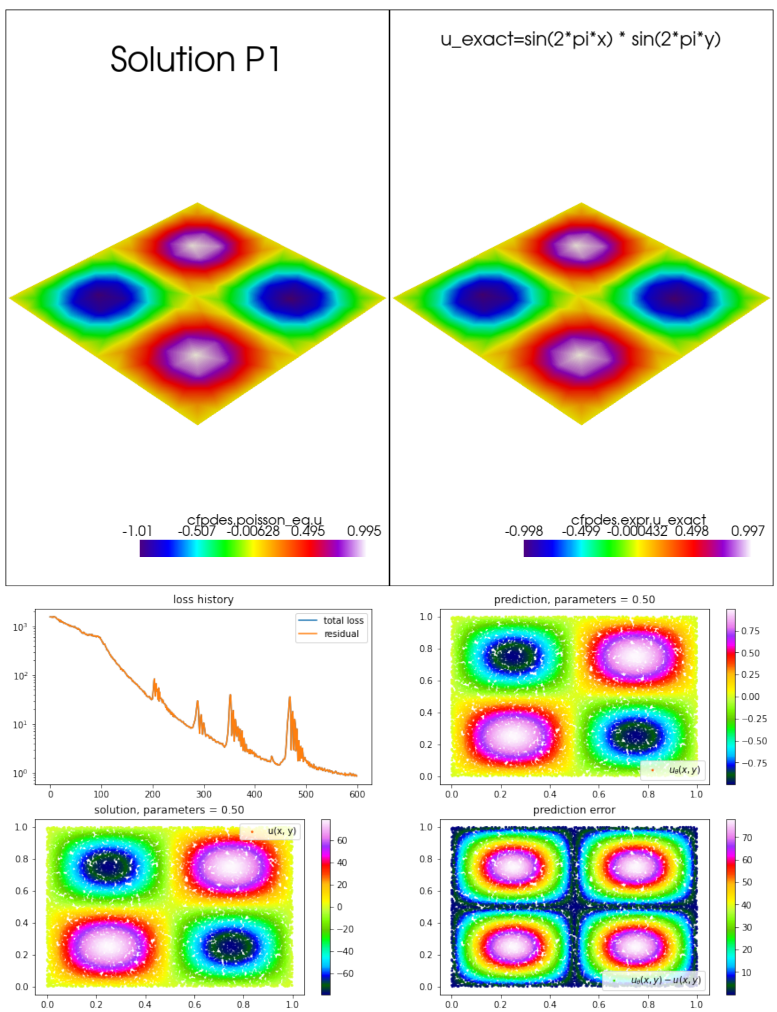
This process allows us to analyze and visualize the behavior of the Laplacian on complex geometries, such as potato shapes, with the aid of neural networks. By including boundary condition loss and adjusting weights, we enhance the accuracy and control of the mapping process. This highlights the potential 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)
```

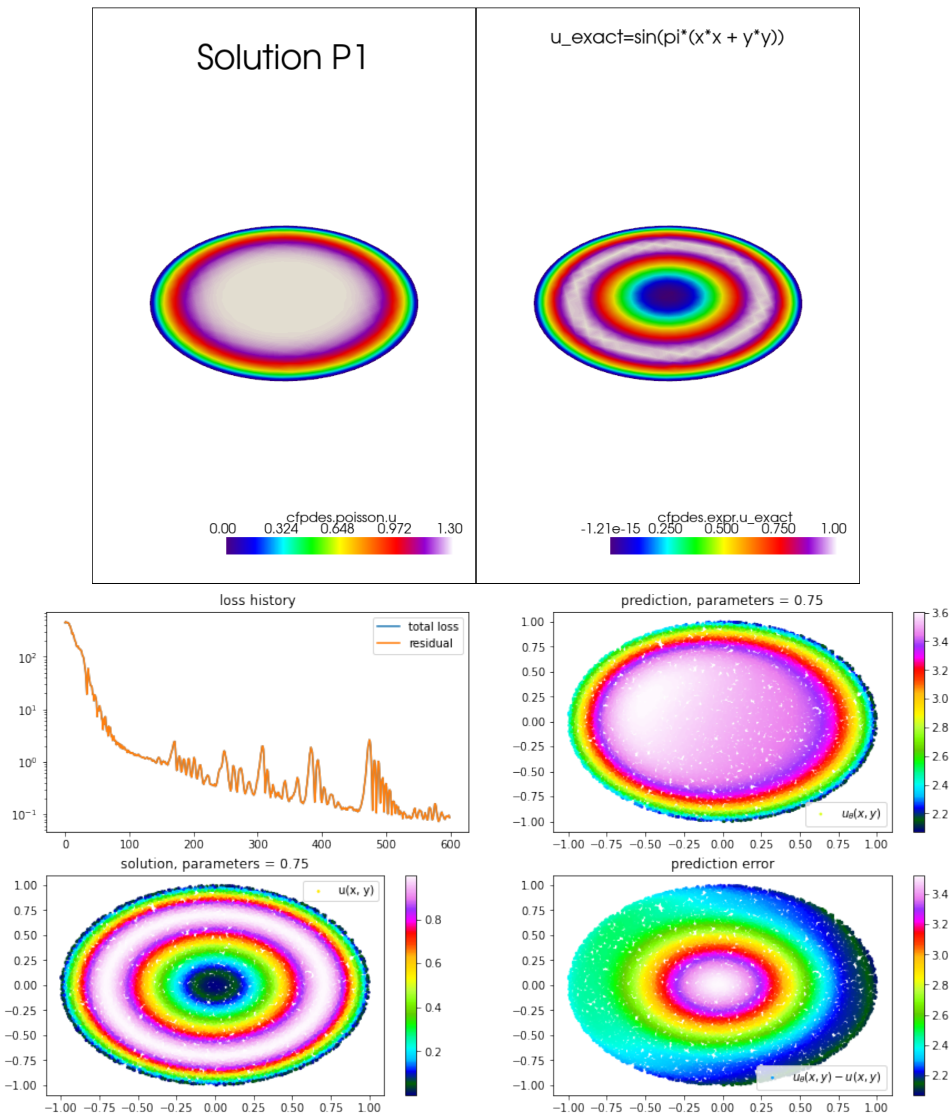
.



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.

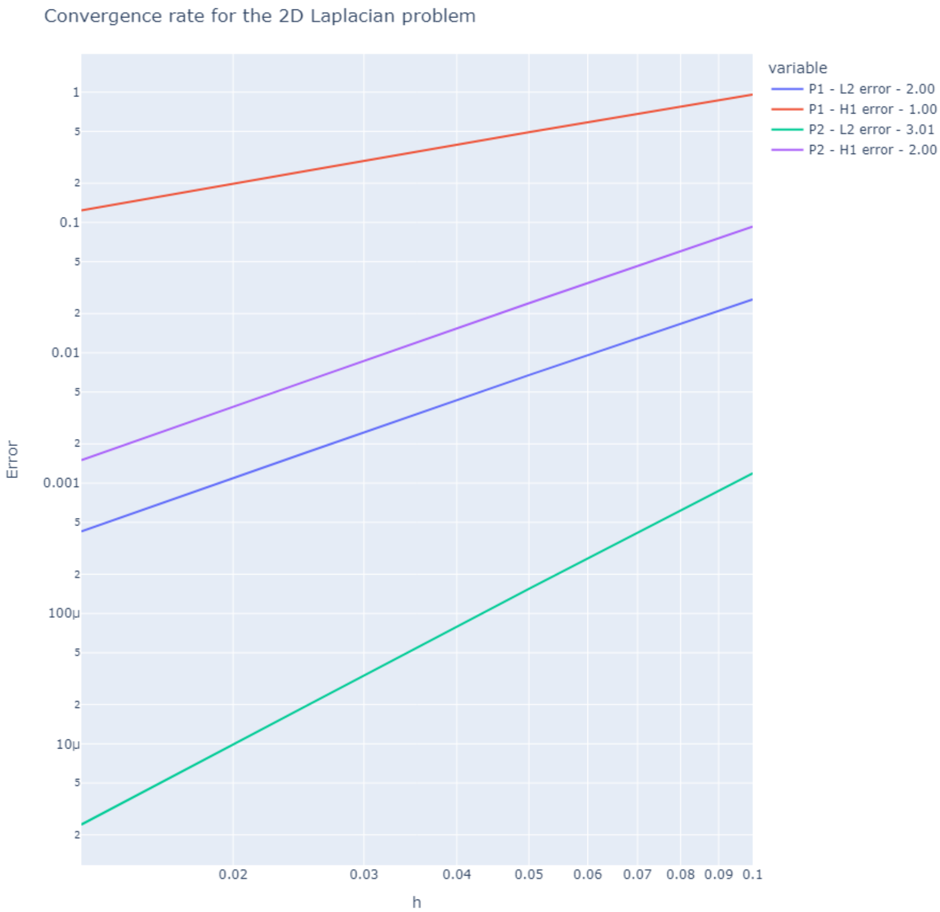
```
# for disk domain
u_exact = 'sin(pi*(x*x + y*y))'
rhs = '4*pi*sin(pi*(x*x + y*y)) - 4*pi*pi*(x*x + y*y)*cos(pi*(x*x + y*y))'

P(rhs=rhs, g='0', order=1, geofile='geo/disk.geo', plot='2d.png',
    u_exact = u_exact)
P(rhs=rhs, g='0', order=1, geofile='geo/disk.geo', solver='scimba',
    u_exact = u_exact)
```

.



6.4 Error convergence rate

.



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 enhance efficiency, accuracy, and versatility in tackling a wide range of scientific and engineering challenges.

SimBa offers advanced capabilities for visualization, computation, and mapping of mathematical models, providing valuable insights into the behavior of complex systems. Its flexibility and scalability make it a valuable tool for analyzing various phenomena and generating visual representations that aid in understanding and interpretation.

On the other hand, Feel++ provides a comprehensive framework for finite element analysis, offering robust solvers and customizable parameters for solving differential equations and simulating physical processes. Its versatility and adaptability make it well-suited for a wide range of applications in scientific computing.

By combining SimBa's training capabilities with Feel++'s powerful solver framework, we can leverage the strengths of both tools to solve problems more easily and efficiently. This integration allows us to perform comprehensive analyses, visualize results in meaningful ways, and gain deeper insights into both solvers.

The project successfully demonstrates the use of the CFPDE toolbox, leveraging both Feel++ and ScimBa frameworks, for solving Poisson equations across various domains and dimensions. 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.

4. Solver Performance: (remaining)

   (a) comparing Feel++ and ScimBa visuals.

   (b) comparing error convergence.

Overall, the project illustrates a comprehensive approach to solving and visualizing Poisson equations using advanced computational tools and frameworks. The successful implementation and visualization of solutions across different domains and dimensions underscore the robustness of the methodology and the effectiveness of the CFPDE toolbox.

Bibliography

[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