Project report: Coupling ScimBa and Feel++

Helya Amiri[*] & **Rayen Tlili**[†]

Supervised by Christophe Prud'homme, Joubine Aghili

Submitted May 22, 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.

2 General 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.

3 Specific objectives

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.

4 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.**

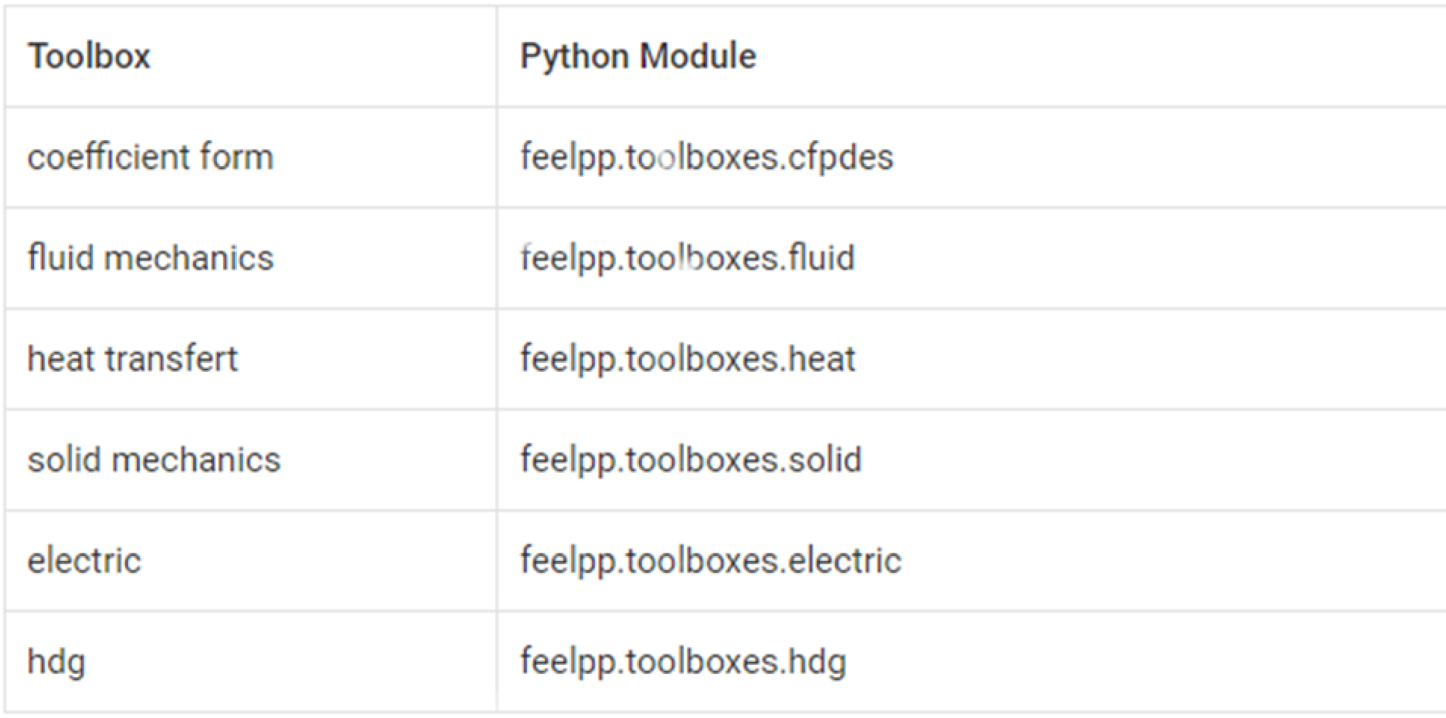
5 Exploring Feel++ documentation

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

5.1 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: .



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**:

5.2 Coefficient Form Toolbox

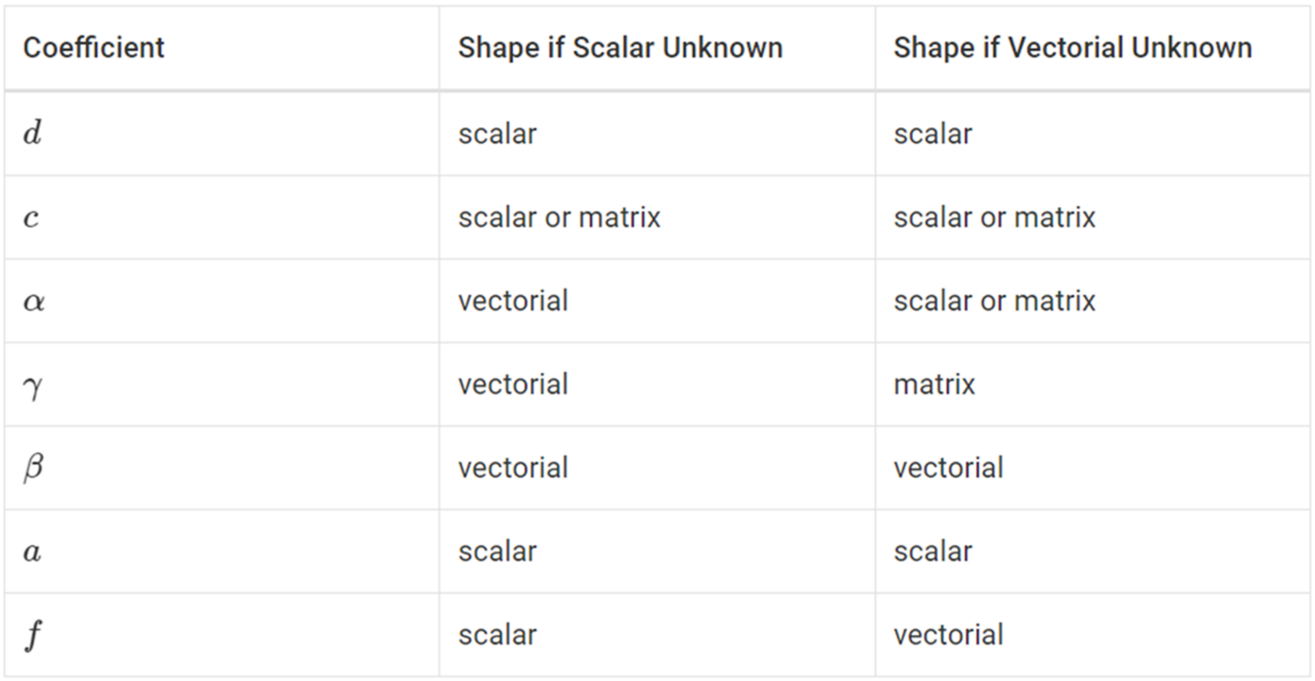
1. **What are Coefficient Form PDEs?**: The coefficient forms in PDE (Partial Differential Equation) toolboxes encapsulate crucial properties like diffusion, convection, and reaction coefficients. These coefficients are vital for characterizing diverse PDEs such as elliptic, parabolic, or hyperbolic 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

6 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.

```python
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.

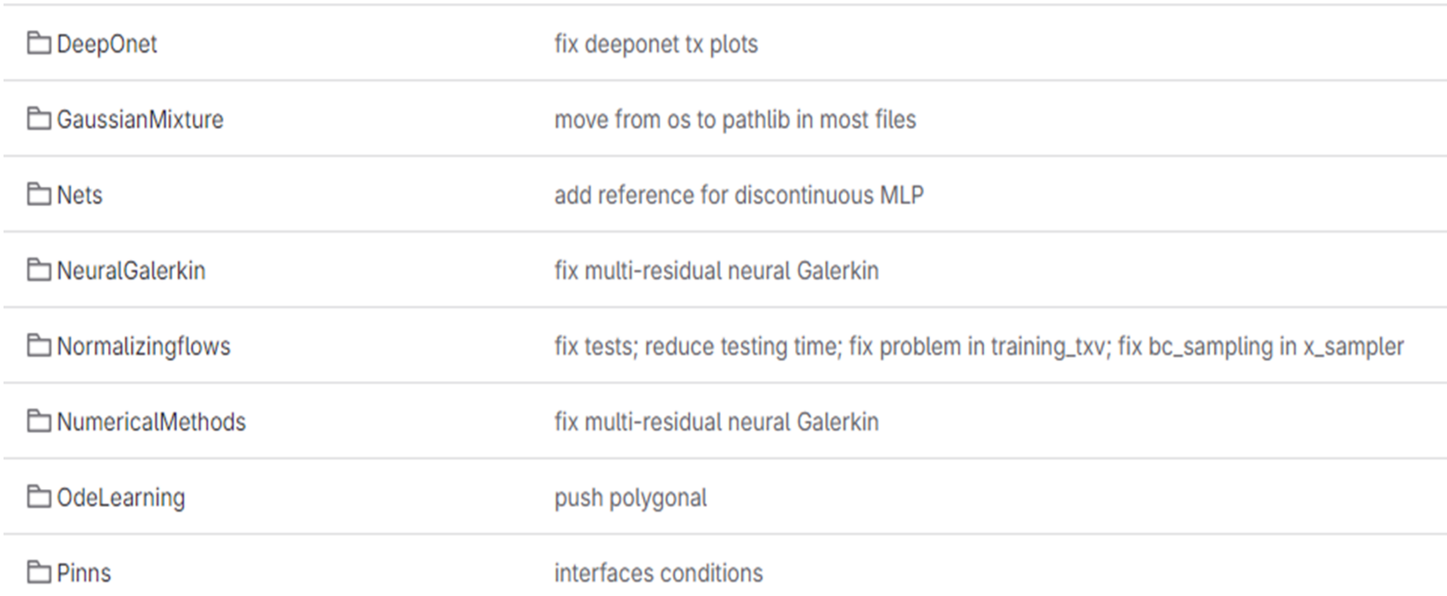
```python
class Poisson_2D(pdes.AbstractPDEx):
    def __init__(self, space_domain):
        super().__init__(
            nb_unknowns=1,
            space_domain=space_domain,
            nb_parameters=1,
            parameter_domain=[[0.50000, 0.500001]],
        )
```

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.

```python
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

7 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 Python libraries."

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


# Clone the Scimba repository
RUN git clone https://gitlab.inria.fr/scimba/scimba.git /scimba

# Install Scimba and its dependencies
WORKDIR /scimba
RUN pip3 install .

# 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", "python3"]*
```

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

8 Methodology

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

Create the right environment for using the CFPDE toolbox:

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

from tools.solvers import Poisso
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

```
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.

```
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)
```

9 Results

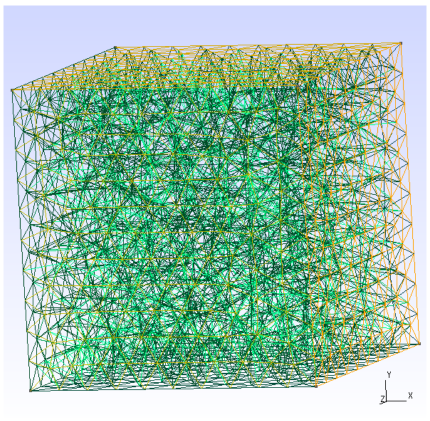
9.1 Generating visuals using Feel++

Using the Poisson class and the Laplacian2D.py program,

```
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:

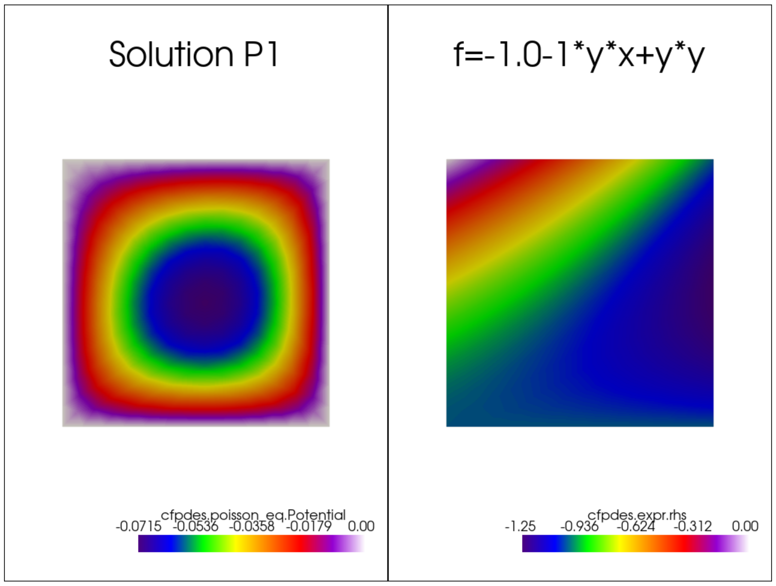
.



Calling the Poisson class specifying the parameters:

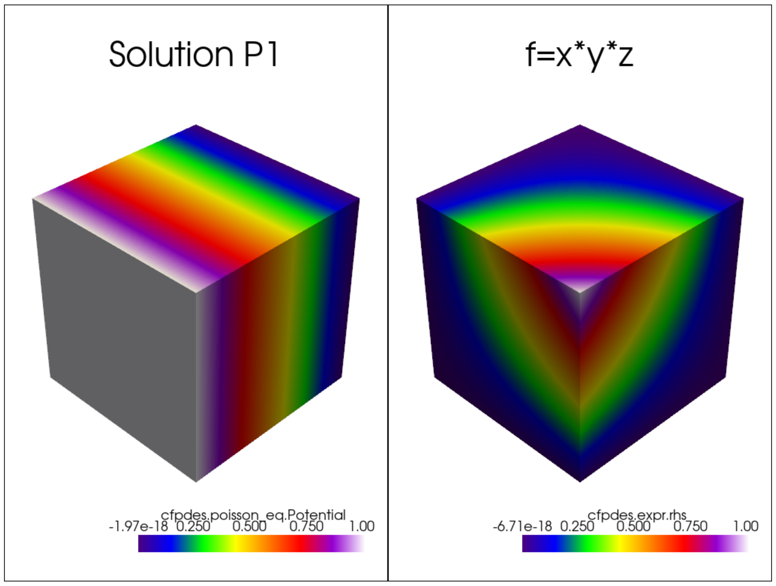
```
P = Poisson(dim = 2)
P(h=0.08,  rhs='-1.0-1*y*x+y*y', g='0', order=1, plot='f4.png')
```

.



```
P(h=0.08,  rhs='-1.0-1*y*x+y*y', g='0', order=1, solver='scimba')
```

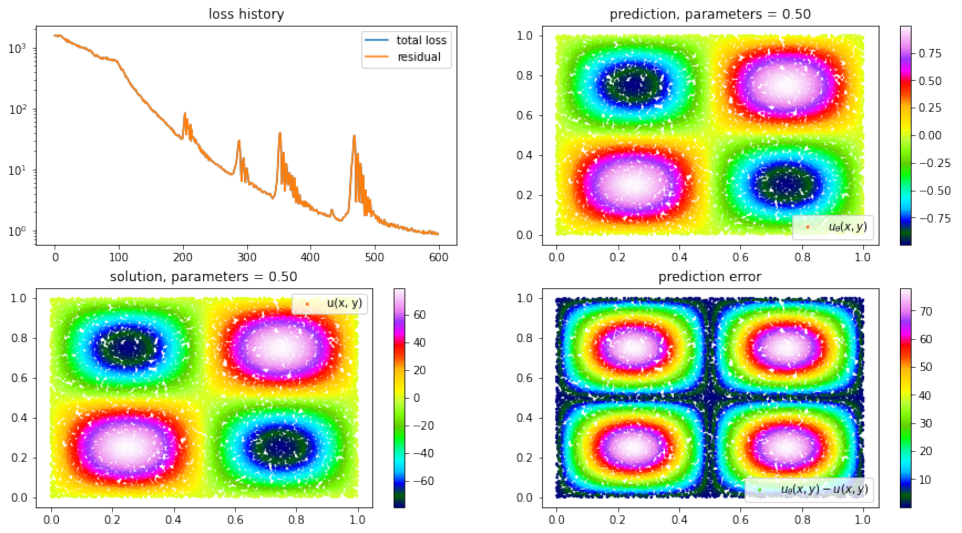
.


Solution P1
f=x*y*z
cfpdes.poisson_eq.Potential
-1.97e-18 0.250 0.500 0.750 1.00
cfpdes.expr.rhs
-6.71e-18 0.250 0.500 0.750 1.00

9.2 Generating visuals using ScimBa

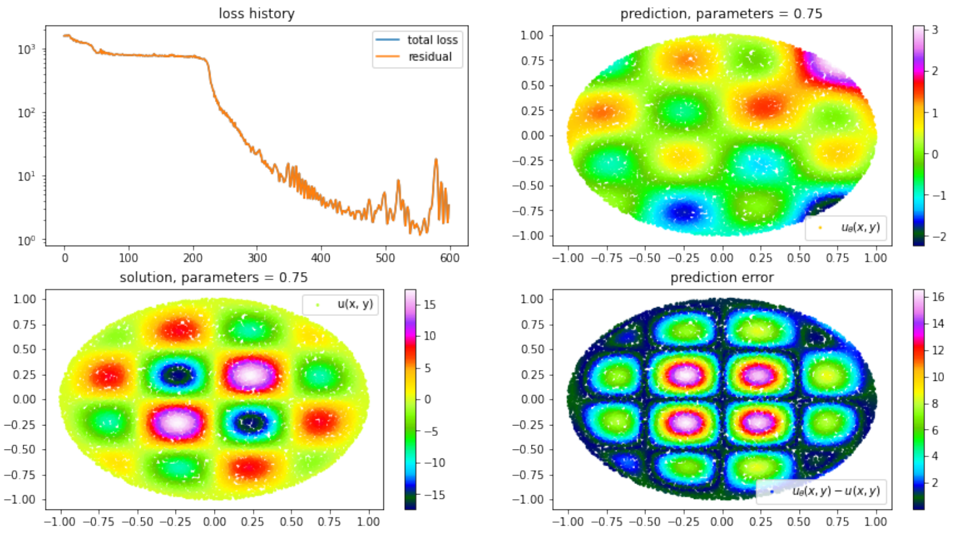
```
xdomain = domain.SpaceDomain(2, domain.SquareDomain(2, [[0.0, 1.0],
                                                        [0.0, 1.0]]))
pde = Poisson_2D(xdomain,  rhs='8*pi*pi*sin(2*pi*x)*sin(2*pi*y)', g='0')
Run_laplacian2D(pde)
```

.



```
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)
```

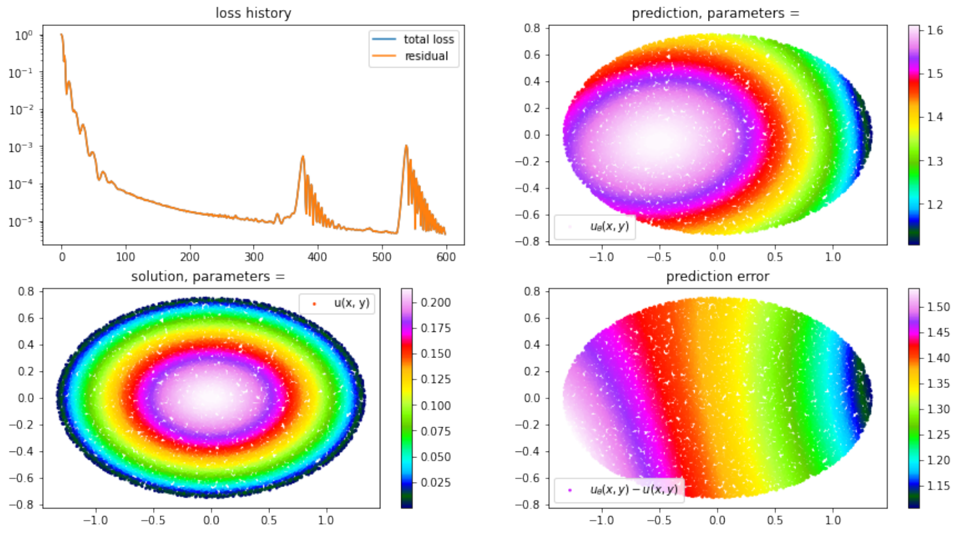
.



Laplacian on ellipse and mapping with nn

```
    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)
```

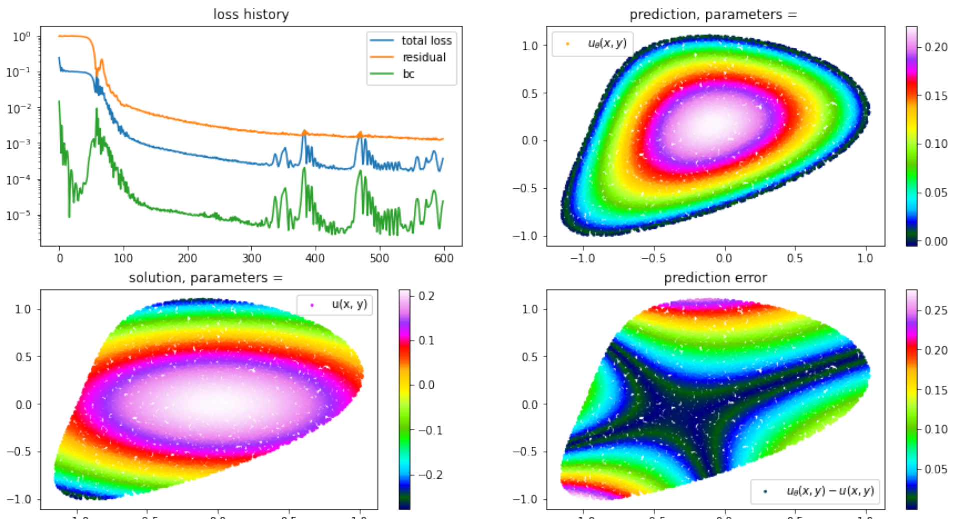
.



Laplacian on potato and mapping with nn

```
    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)
```

.


loss history
total loss
residual
bc
prediction, parameters =
$u_\theta(x,y)$
solution, parameters =
u(x, y)
prediction error
$u_\theta(x,y)-u(x,y)$

10 Conclusion

Bibliography

[1] Feel++. (n.d.). *Modeling and Quantitative Simulation*. Retrieved from `https://feelpp.github.io/mqs/cases/0.109/index.html`

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

[3] MSO4SC. (n.d.). *Poisson*. Retrieved from `https://book.mso4sc.cemosis.fr/cases/0.109/poisson/README/`

[4] Zenodo. (n.d.). *Zenodo Record*. Retrieved from `https://zenodo.org/records/1231527`

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

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

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

[8] Cemosis. (n.d.). *Course: Solving PDEs with Feel++*. Retrieved from `https://www.cemosis.fr/events/course-solving-pdes-with-feel/`

[9] MSO4SC. (n.d.). *Feel++ Book*. Retrieved from `https://book.mso4sc.cemosis.fr/feelpp/`

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

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

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

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

[14] Feel++. (n.d.). *Mixed Poisson*. Retrieved from `https://docs.feelpp.org/user/latest/using/toolboxes/hdg_poisson.html`