Project : Radial Basis and Low Rank

Rodolphe Vivant

M1 CSMI

May 30, 2025

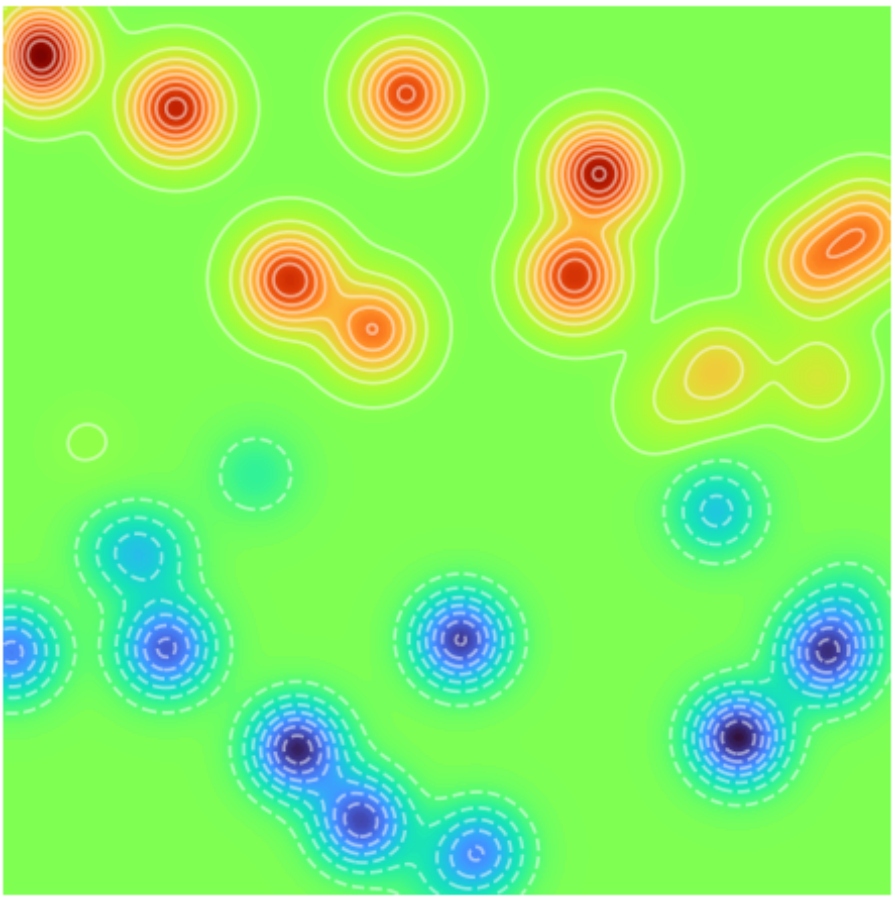

Figure 1: Approximated space with Radial Basis Functions

Abstract

Ce rapport présente l'implémentation et l'application des noyaux à base radiale (RBF) pour la résolution d'équations différentielles dans le package Python Scimba. Les RBF sont des méthodes sans maillage qui approchent les solutions à l'aide de noyaux ne dépendant que de la distance entre les points, ce qui les rend adaptées aux géométries complexes et aux problèmes de grande dimension. Plusieurs types de noyaux RBF, dont le gaussien, la puissance exponentielle, le Matern, le multiquadratique et le gaussien anisotrope, sont implémentés et testés sur des tâches d'approximation de fonctions et la résolution d'une équation de Poisson. Les résultats mettent en évidence l'importance du choix du noyau pour la régularité et la précision de l'approximation. Le projet confirme les propriétés théoriques des RBF et démontre leur efficacité pour des applications d'apprentissage automatique scientifique. Les perspectives incluent l'extension des noyaux implémentés et leur application à une classe plus large d'équations différentielles.

This report presents the implementation and application of Radial Basis Function (RBF) kernels for solving differential equations within the Scimba Python package. RBFs are mesh-free methods that approximate solutions using kernels dependent only on the distance between points, making them suitable for complex geometries and high-dimensional problems. Several types of RBF kernels, including Gaussian, powered exponential, Matern, multiquadric, and anisotropic Gaussian, are implemented and tested on function approximation tasks and the solution of a Poisson equation. The results highlight the importance of kernel choice for the regularity and accuracy of the approximant. The project confirms theoretical properties of RBFs and demonstrates their effectiveness for scientific machine learning applications. Future work includes extending kernel implementations and applying them to a broader class of differential equations.

Contents

1 Introduction

1.1 History

[2] Meshfree methods for the solving of differential equations have been developped in many areas of computational science. Originally they were applied to problems in geophysics, mapping or meteorology. They have gained popularity in recent years due to their ability to handle complex geometries and adapt to changing domains. The method we are going to present is the Radial Basis Functions (RBFs) method, which is a meshfree method that uses radial basis kernels to approximate the solution of partial differential equations (PDEs).
I will first present the theoretical background of RBFs and there application for the solving of partial differential equations, then I will describe mathematically the kernels I implemented in the Scimba package. Finally, I will present the implementation itself and the results for each kernel of the projection of 1D and 2D fucntions and the solving of a Poisson equation in 2D.

1.2 Background

1.2.1 Radial Functions

Radial basis functions (RBFs) are a class of functions that depend only on the distance from a center point[3],[2].

$$\psi(\|\mathbf{x} - \xi_j\|, \epsilon) \tag{1}$$

where ξ_j, $j = 1, .., N$ are the centers of the RBFs and ϵ is a parameter that controls the width of the RBF.

We can define the RBF approximant of a certain function f as:

$$f(x) \sim s(\mathbf{x}, \epsilon) = \sum_{j=1}^{N} \alpha_j \psi(\|\mathbf{x} - \xi_j\|, \epsilon) \tag{2}$$

where α_j are the coefficients of the basis functions centered at ξ_j for $j = 1, .., N$. In other words, $s(x, \epsilon) \in span(K(., \xi_j)_{j\in[1,N]})$.
In our case, we will need to guarantee a certain regularity of the approximant. The function tested, or the solution of the PDEs we want to approximate, must be in a Sobolev spaces $H^m(\mathbb{R}^d)$, where d is the dimension of the domain Ω we are working on. To do so, we will have to ensure that our kernel are positive (semi) definite. The choice of the kernel is crucial for the regularity of the approximant. We will test that property trough our examples.

Then, from a set of given data $(\mathbf{x}_i, u_i)$, $i = 1, .., N_c$ where $u_i = f(x_i)$, we match the RBF approximant to the data:

$$s(\mathbf{x}_i, \epsilon) = u_i \quad \forall i = 1, .., N_c \tag{3}$$

In a matrix form, we can write the system of equations 2, 3 as:

$$A\alpha = \mathbf{U} \tag{4}$$

where

$$\begin{aligned} A_{ij} &= \psi(\|\mathbf{x}_i - \xi_j\|, \epsilon), \\ U_i &= u_i. \end{aligned} \tag{5}$$

After solving this system, the approximant can be evaluated at any points in the domain Ω. This technique applied to PDE's is called collocation. During this project, we will only use the collocation method to solve PDEs.

The main advantage of RBFs is twofold. They are meshfree as told before, and the only information needed is the distance between the points. Therefore, they are very useful for complex geometries and can be easily implemented at any dimension. The drawback is that the system of equations can be ill-conditioned, especially for large number of points. But also, the RBFs can be globally supported RBFs resulting in dense matrices, which can be computationally expensive to solve.

1.2.2 Global collocation for PDEs

Let's study the case of a time-independent PDE on a dobmain $\Omega \subset \mathbb{R}^d$:

$$\begin{aligned} \mathcal{L}u(\mathbf{x}) &= f(\mathbf{x}), \quad \mathbf{x} \in \Omega, \\ \mathcal{L}_{\partial\Omega}u(\mathbf{x}) &= g(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega \end{aligned} \tag{6}$$

??

where $\mathcal{L}$ is a linear differential operator, f is a source term and g is a boundary condition. After insterting the RBF approximant (2) into the PDE, we obtain:

$$\begin{aligned} \mathcal{L}s(\mathbf{x},\epsilon)|_{x=x_i} &= \sum_{j=1}^{N} \alpha_j \mathcal{L}\psi(\|\mathbf{x}-\xi_j\|,\epsilon)|_{x=x_i} = f(x_i), \quad x_i \in \mathcal{I} \\ \mathcal{L}_{\partial\Omega}s(\mathbf{x},\epsilon)|_{x=x_i} &= \sum_{j=1}^{N} \alpha_j \mathcal{L}_{\partial\Omega}\psi(\|\mathbf{x}-\xi_j\|,\epsilon)|_{x=x_i} = g(x_i), \quad x_i \in \mathcal{B} \end{aligned} \tag{7}$$

where the $x_i \in \mathcal{I}$ are the collocation points inside Ω and the $x_i \in \mathcal{B}$ are the collocation points on the border $\partial\Omega$.

The system of equations 7 can be written in a matrix form:

$$\begin{pmatrix} L \\ B \end{pmatrix} (\alpha) = \begin{pmatrix} f \\ g \end{pmatrix} \tag{8}$$

where

$$\begin{aligned} L_{ij} &= \mathcal{L}\psi(\|\mathbf{x}-\xi_j\|,\epsilon)|_{x=x_i}, \quad x_i \in \mathcal{I} \\ B_{ij} &= \mathcal{L}_{\partial\Omega}\psi(\|\mathbf{x}-\xi_j\|,\epsilon)|_{x=x_i}, \quad x_i \in \partial\Omega \\ f_i &= f(\mathbf{x}_i), \quad x_i \in \mathcal{I} \\ g_i &= g(\mathbf{x}_i), \quad x_i \in \partial\Omega \end{aligned} \tag{9}$$

1.2.3 Training the RBFs

To solve the system of equations, we can use a linear or a non-linear approach. To find the approximant, we train our model with the following set of parameters θ :

- Case Linear : $\theta = \{\alpha_j\}$
  There is only one step of linear regression to find the coefficients α_j of the untrained RBFs.
- Case nonLinear : $\theta = \{\alpha_j, \xi_j, \epsilon_j\}$,
  - First we train the parameters of the RBFs, which are the centers ξ_j and the widths ϵ_j of the kernels,
  - Then we optimize the coefficients α_j of the approximant through a linear regression step.

2 Project Description

The main objective of this project is to implement RBFs kernel to solve differential equations in the existing Scimba package.

2.1 Scimba

Scimba is a Python package for the implementation of different Scientific Machine Learning methods. These are the main features of the package:

- Networks implementation : Multi Layer Perceptron (MLP), Discontinuous MLP, RBF networks, activation functions and a basic trainer
- Models of differentials equations : Ordinary differetial equations (ODE), Partial (PDE), Spatial PDEs, time-space PDEs,...
- Specific networks for Physics informed neural networks (PINN) : MLP, Discontinuous MLP, nonlinear RBF networks, Fourier networks, etc.
- Trainer: Each type of PDE has its own trainer

2.2 Radial Kernels

As previously mentioned, the objective of this project is to implement RBFs kernels to solve differential equations in the Scimba package. Here are the kernels I implemented[1]:

2.2.1 Isotropic kernels

These kernels can be written in the form:

$$K(\mathbf{x}, \mathbf{y}) = \kappa(\|\mathbf{x} - \mathbf{y}\|) \tag{10}$$

with $\kappa : \mathbb{R}_0^+ \to \mathbb{R}$ and $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$.
They only depend on the distance between the points $\mathbf{x}$ and $\mathbf{y}$, thus the isotropic property.

We only focused on the implementation of the following kernels:

Powered exponential kernel

$$\kappa(r) = e^{-(\epsilon r)^\beta} \tag{11}$$

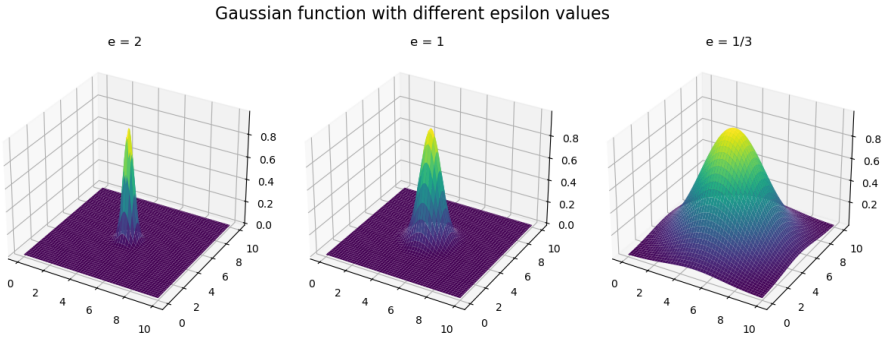

Figure 2: Gaussian RBF

- This kernel is ofen used in statistics and machine learning.
- The parameter ϵ controls the width of the kernel, and $\beta \in]0, 2]$ is a positive real number that controls the shape of the kernel.

- If $\beta = 2$, we obtain the Gaussian kernel, which is the most popular RBF kernel. This kernel generates good approximation for $\mathcal{C}^\infty(\mathbb{R}^d)$ functions.
- If $\beta = 1$, we obtain the exponential kernel, which gives a good approximation for $\mathcal{C}^0(\mathbb{R}^d)$ functions.

Matern kernel

$$\kappa(x_i, x_j) = \frac{1}{\Gamma(\nu)2^{\nu-1}} \left(\frac{\sqrt{2\nu}}{l} d(x_i, x_j) \right)^\nu K_\nu \left(\frac{\sqrt{2\nu}}{l} d(x_i, x_j) \right) \tag{12}$$

where $d(.,.)$ is the Euclidean distance,
Γ is the gamma function,
K_ν : the modified Bessel function of the second kind of order ν.

It's a generalization of the Radial Basis Function kernels.

- Used in Statistics and approximation theory.
- $\nu = 1/2$: Identical to the exponential kernel
- $\nu \to \infty$: Gaussian kernel
- $\nu = 3/2$: Good approximation for once differentiable functions
- $\nu = 5/2$: Good approximation for twice differentiable functions
- Generate classical Sobolev spaces $H^{\nu+\frac{d}{2}}(\mathbb{R}^d)$

For example, in 1D, the mattern Kernel with $\nu = 1/2$ generates the Sobolev space $H^1(\mathbb{R})$ wich is continuously injected in $\mathcal{C}^0(\mathbb{R})$. We will test that result in the next section.

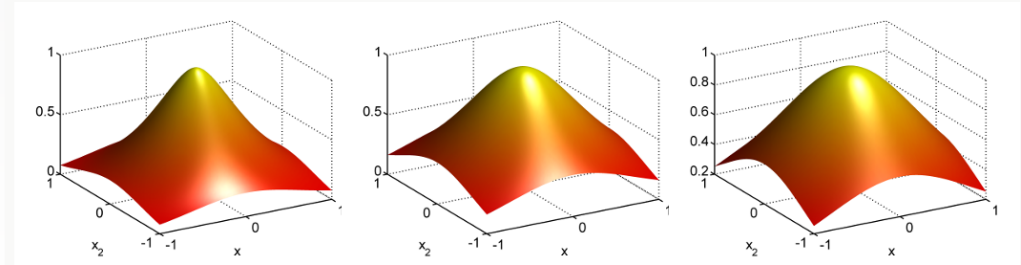

Figure 3: Matern RBF : Left $\nu = 1/2$ \ Center $\nu = 3/2$ \ Right $\nu = 5/2$

(Inverse) Multiquadric kernel

$$\kappa(\epsilon r) = (1 + \epsilon^2 r^2)^\beta \tag{13}$$

where $\beta \in \mathbb{R}\ \mathbb{N}_0$

- $\beta < 0$: IMQ
- $\beta > 0$: MQ

These kernels are used in approximation theory and engineering.

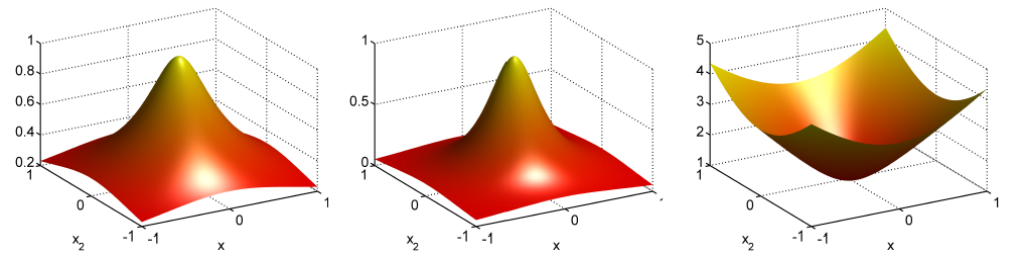

Figure 4: Multiquadric RBF : Left IMQ :$\beta = \frac{-1}{2}$ \ Center IQ : $\beta = -1$ \ Right MQ : $\beta = \frac{1}{2}$

2.2.2 Anisotropic kernels

Anisotropic Gaussian kernel

$$K(\mathbf{x}, \mathbf{z}) = e^{-(\mathbf{x}-\mathbf{z})^T E(\mathbf{x}-\mathbf{z})} \tag{14}$$

where E is a symmetric positive definite matrix.
If $E = \epsilon^2 I_d$, we obtain the isotropic Gaussian kernel.

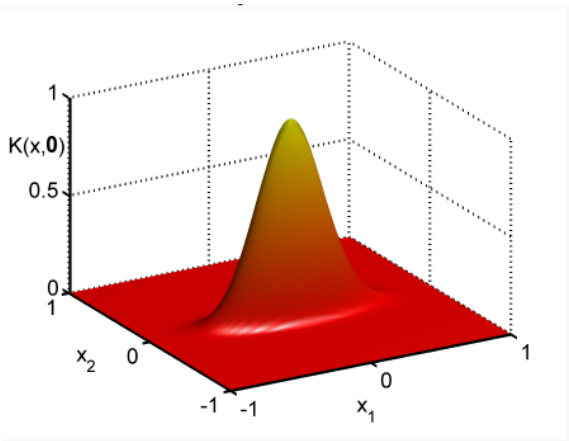

Figure 5: Anisotropic Gaussian RBF

3 Implementation and Results

The implementation of the kernels is done in the file : `src/scimba_torch/approximationspace/kernelx_space.py`.

The implementation of our kernels is contain in the KernelxSpace class that inherit from the AbstractApproxSpace and ScimbaModule classes.
This class builds a parametric approximation space, where the solution is approached by a neural network. It containes methods evaluating the network, setting and retrieving its degrees of freedom, and computing its Jacobian.
First, we create the tensor of the centers of the RBFs as an attribute of the class with a given sampler in a domain defined by the user.
Then we ask for this tensor to be a parameter of the network, so that it can be optimized during the training.
Then, we do the same ϵ parameter present in almost all the isotropic kernels.
But for the β coefficient, we do not want to make if a parameter of the network, but rather a hyperparameter that we can set before the training. We tried to implement it as a trainable parameter, but it lead to different issues during the training.

```
self.centers = next(self.integrator.sample(nb_centers))
print("centers",self.centers.x.shape)

self.centers.x = nn.Parameter(
    self.centers.x
)

self.eps = nn.Parameter(
    torch.ones(nb_centers, dtype=torch.get_default_dtype()) * 10
)  # Ecart-type initial

self.beta=beta
```

In the forward method, we start by computing the distance between the input points and the centers of the RBFs. Then we implement the kernels.

3.1 Powered Exponential Kernels

3.1.1 Gaussian kernel

Implementation

First I implemented the Gaussian kernel, which is the most popular RBF kernel.

```
if self.kernel_type == "gaussian":
      # Calcul du noyau gaussien
          gaussian_basis = torch.exp(-(self.eps*distances)**2)
          #print("gaussian_basis",gaussian_basis.shape)

          if with_last_layer:
              res = self.output_layer(gaussian_basis)
          else:
              res = gaussian_basis
```

Visualization of Gaussian RBF's kernels approximated space training

Here is an illustration of the training of our kernels to approximate the solution of a function f defined on the domain $\Omega = [-1, 1] \times [-1, 1]$.

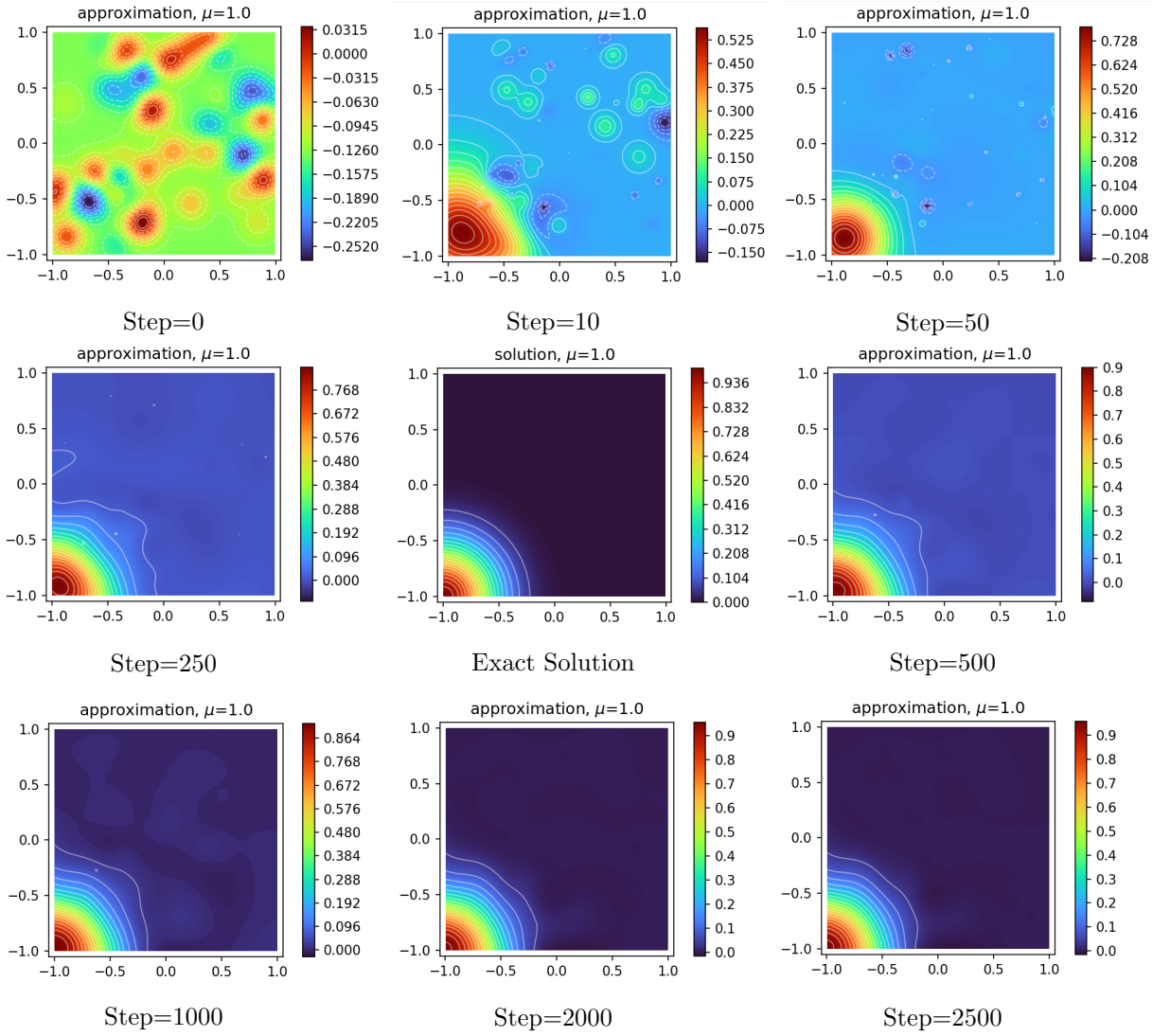

Figure 6: Gaussian RBF's kernels approximated space training at various steps

The training is working well. As we can see the RBFs are adapting to the function f we want to approximate. In this example, we only used 50 centers, but still, because of the set of trainable parmaeters being the centers of the RBFs and their ϵ parameters, the training is already efficient.

Result of the apprimated projection of a function

Now, I want to present an example of the approximation of a function f by the Gaussian RBFs kernel through a linear projection (Trainable parameters $\theta = \alpha_j$ the coefficients of the RBFs) and a collocation projection (Trainable parameters $\theta = \alpha_j, \epsilon_j \text{and} \xi_j$ where ϵ_j and ξ_j are the widths and the centers of the RBFs).

In this example, we will approximate the function $f(x, y) = \sin(\cos(x) \times \sin(y))/(2 \times 0.4^2)$ on the domain $\Omega = [-1, 1] \times [-1, 1]$.

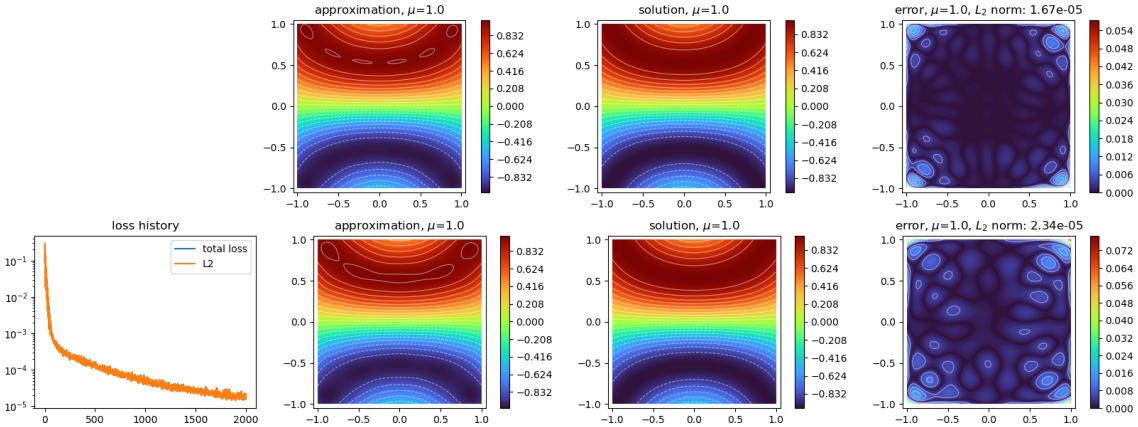

Figure 7: Gaussian RBF's approximation of the function f

Both methods give a good approximation of the function f. As we can see in 7, the loss function is still decreasing after 2000 epochs. We probably could have trained the model for more epochs to get a better approximation of the function f.

3.1.2 Powered exponential kernel

Implementation

Then we implement the powered exponential kernel, which is a generalization of the Gaussian kernel.

```
elif self.kernel_type == "exponential":
            # Calcul du noyau exponentiel

            vect_diff = torch.zeros((features_spatial.shape[0],self.centers
                .shape[0],2), dtype=torch.get_default_dtype())
            vect_diff = features_spatial.unsqueeze(1)-self.centers.
                unsqueeze(0)
            # Calcul de la norme 1 (L1) de vect_diff
            l1_norm = torch.sum(torch.abs(vect_diff), dim=-1)  # (
                batch_size, nb_centers)

            exponential_basis = torch.exp(-(self.eps*l1_norm)**self.beta)
                # (batch_size, 10)
            #print("exponential_basis",exponential_basis.shape)
            if with_last_layer:
                res = self.output_layer(exponential_basis)
            else:
                res = exponential_basis
```

Results

The $\mathcal{C}^\infty(\mathbb{R}^d)$ function f is approximated by the powered exponential RBFs kernel with different values of β and as the theory predicted, the approximation is better when $\beta = 2$ 10 (the Gaussian kernel) than when $\beta = 1$ 9 (the exponential kernel) or $\beta = 0.5$ 8 (a lesser powered exponential kernel).

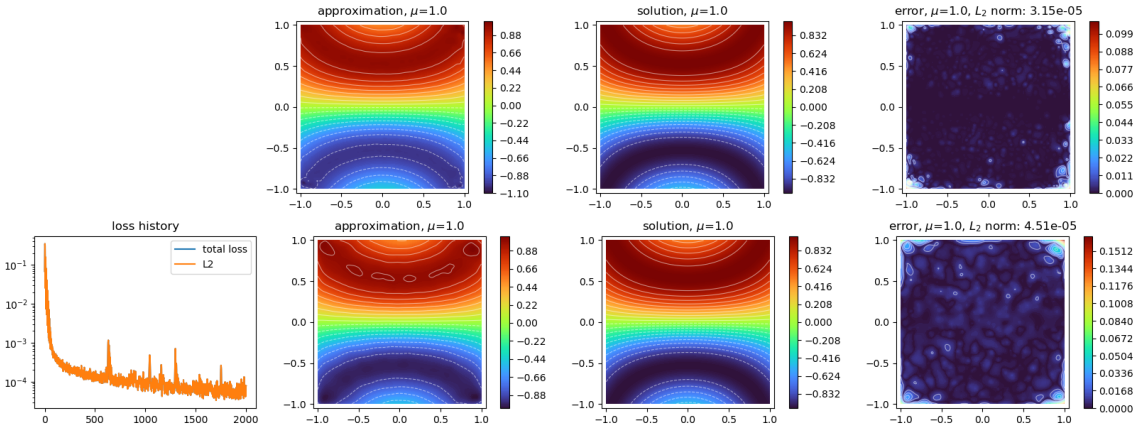

Figure 8: Powered exponential RBF's approximation of the function f with $\beta = 0.5$

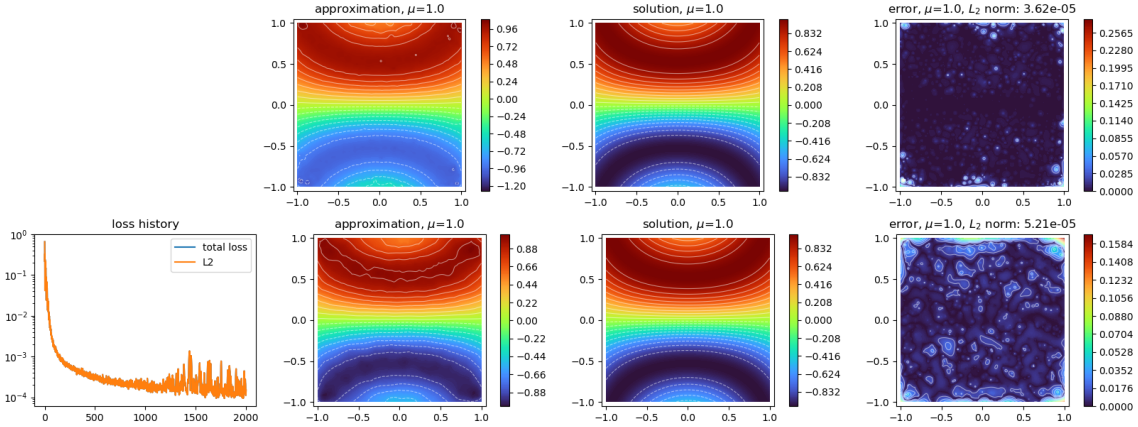

Figure 9: Powered exponential RBF's approximation of the function f with $\beta = 1$

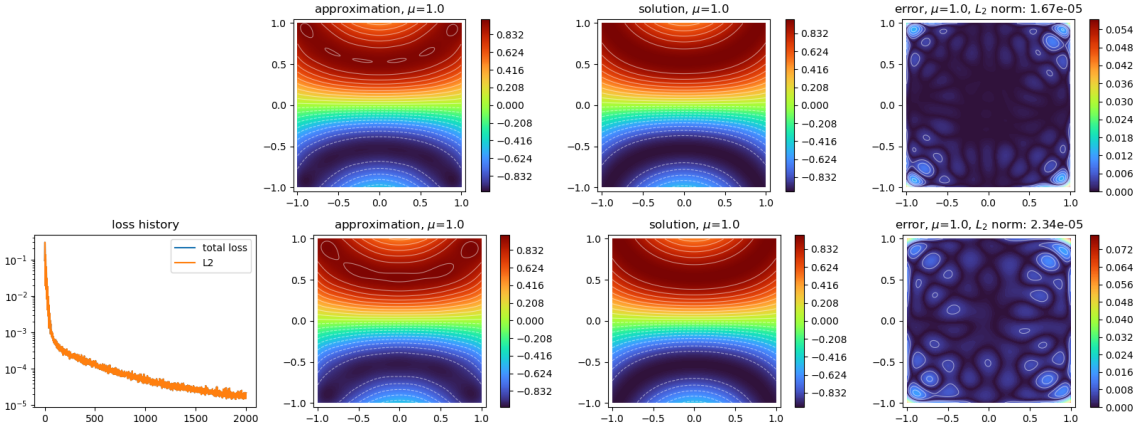

Figure 10: Powered exponential RBF's approximation of the function f with $\beta = 2$

3.2 Matern kernel

Implementation

Then we implement the Matern kernel, which is a generalization of the Radial Basis Function kernel.

```
elif self.kernel_type == "matern":
            # Calcul du noyau Mattern
            #print("features",(features.shape[1]))
            order = 0.5
            matern_kernel = gpytorch.kernels.MaternKernel(lenght_scale=self
                .eps,nu=order)

            # Evaluate the kernel on the input features and centers
            mattern_basis = matern_kernel(features_spatial, self.centers.x)
                .evaluate()

            if with_last_layer:
                res = self.output_layer(mattern_basis)
            else:
                res = mattern_basis
```

Results

Comparison between the Gaussian kernel, the Matern kernel with $\nu = 1/2$, and the exponential kernel.

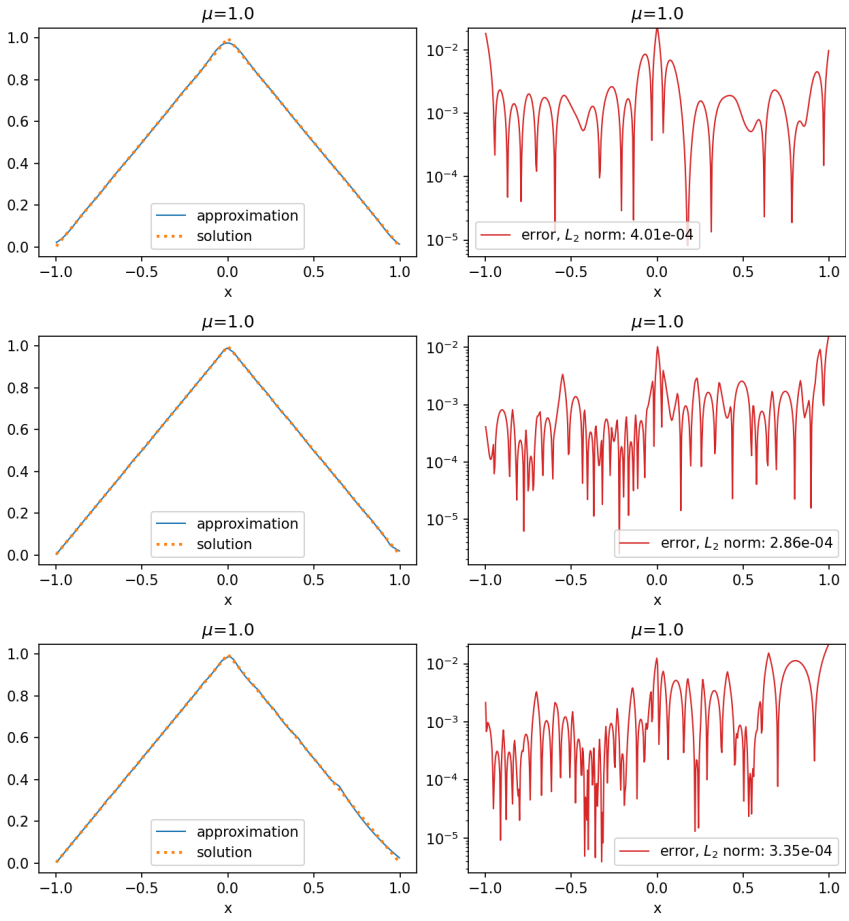

Figure 11: For a $\mathcal{C}^0(\mathbb{R})$ function

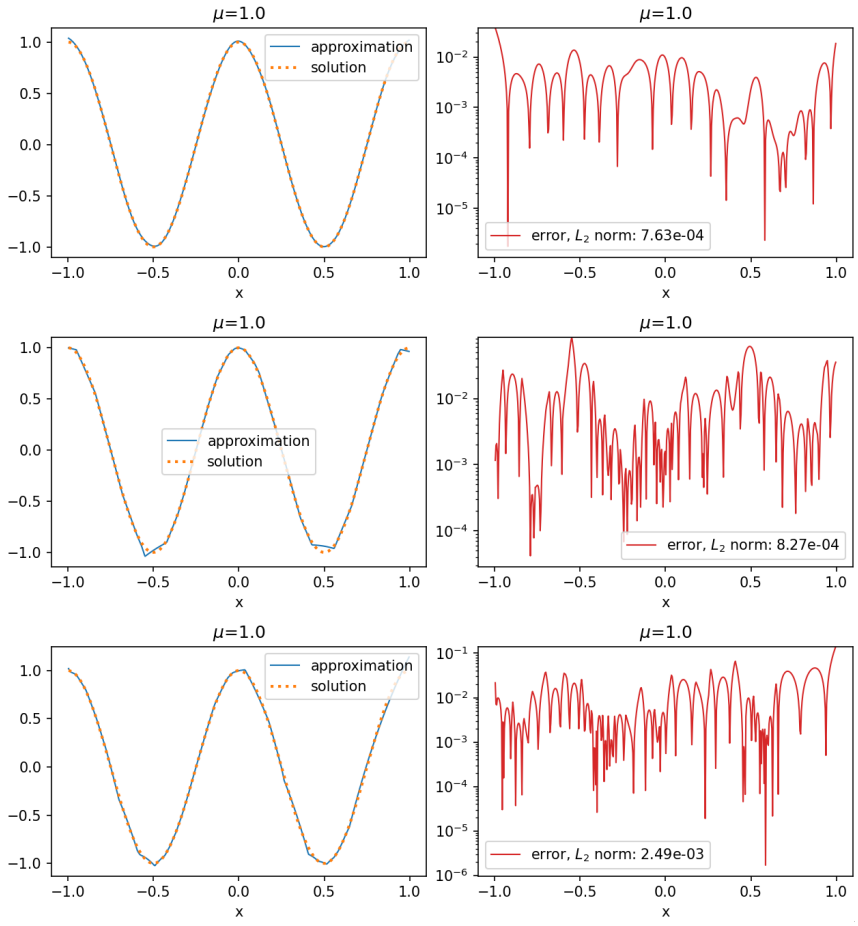

Figure 12: For a $\mathcal{C}^\infty(\mathbb{R})$ function

This two 1D examples show the behaviours of the approximations depending on the chosen kernel. The Gaussian kernel is the best approximation for $\mathcal{C}^\infty(\mathbb{R}^d)$ functions 12, while the Matern kernel with $\nu = 1/2$ and Exponential kernel are the best approximations for $\mathcal{C}^0(\mathbb{R}^d)$ functions 11.

Let's now text the kernel on our 2D function f :

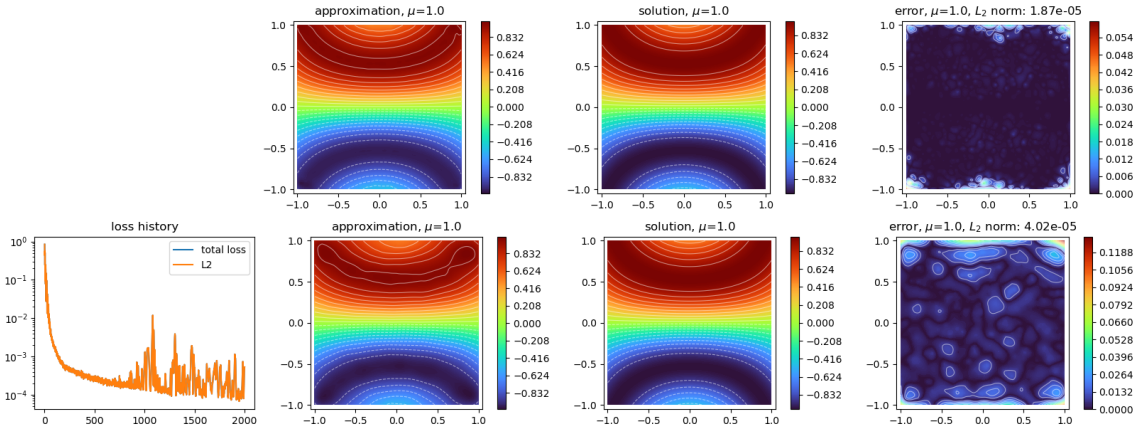

Figure 13: Matern RBF's ($\beta = 2.5$) approximation of the function f

The Matern RBF's kernel with $\beta = 2.5$ 13 gives pretty good projections for the function f we want to approximate, as we can see in the figure above.

3.3 (Inverse) Multiquadric kernel

Implementation

Then we implement the (inverse) multiquadric kernel.

```
elif self.kernel_type == "multiquadratic":
            # Calcul du noyau multiquadratique
            multiquadratic_basis = (1 + (self.eps*distances)**2)**self.beta
            #print("multiquadratic_basis",multiquadratic_basis.shape)
            if with_last_layer:
                res = self.output_layer(multiquadratic_basis)
            else:
                res = multiquadratic_basis
```

Results

(Inverse) Multiquadric RBF's approximation of the function f :

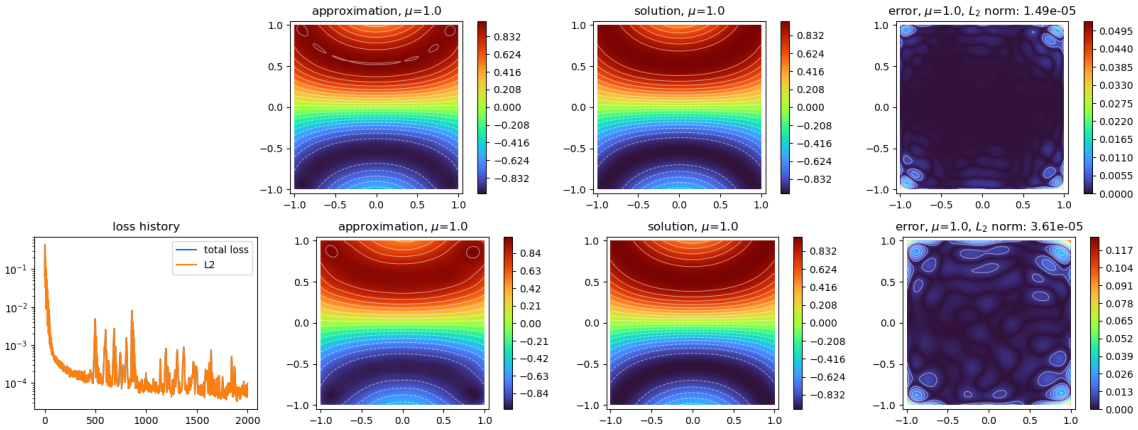

Figure 14: $\beta = -1$

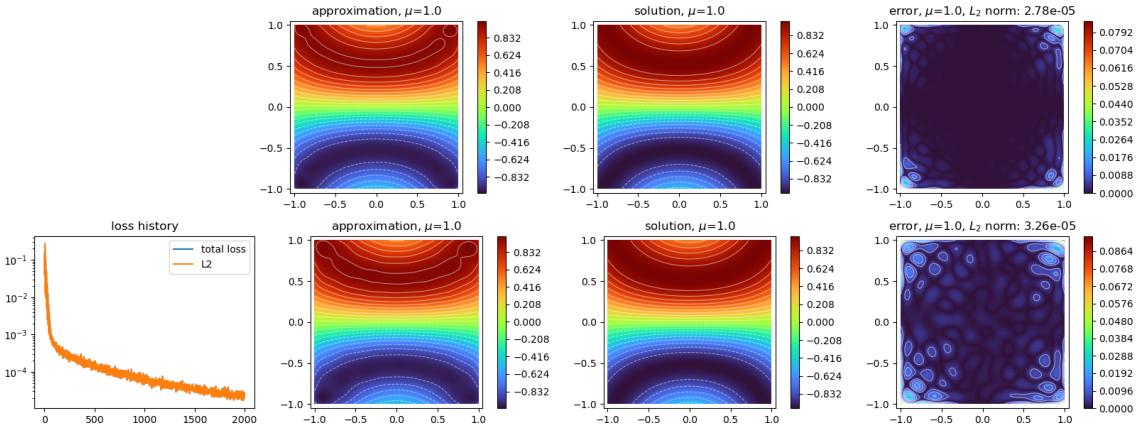

Figure 15: $\beta = -2$

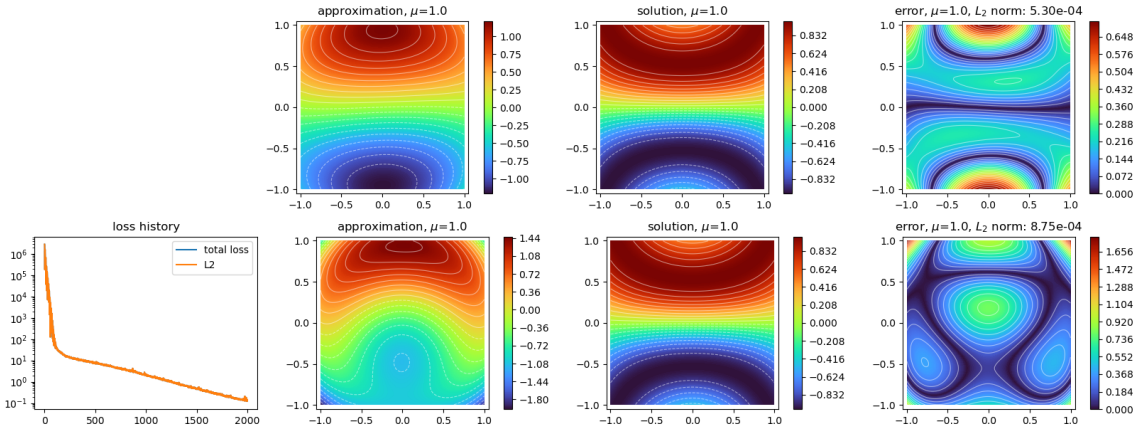

Figure 16: $\beta = 2$

The (inverse) multiquadric RBFs kernel 14, 15is a good approximation for the function f we want to approximate for $\beta < 0$. But for $\beta > 0$ 16, the kernels can't give a good approximation of the function f.

3.4 Anisotropic Gaussian kernel

Case of the Gaussian Kernel

Let's start with the particular case of the Gaussian Kernel (where the anisotropic Matrix $E = \epsilon^2 Id$).

In this section, i will use the following anisotropic gaussian function to test the anisotropic Gaussian kernel:

$$f(x, y) = \exp\left(-50 * (x - 0.5)^2 - 5 * (y + 0.5)^2\right) \tag{15}$$

in the domain $\Omega = [-1, 1] \times [-1, 1]$.

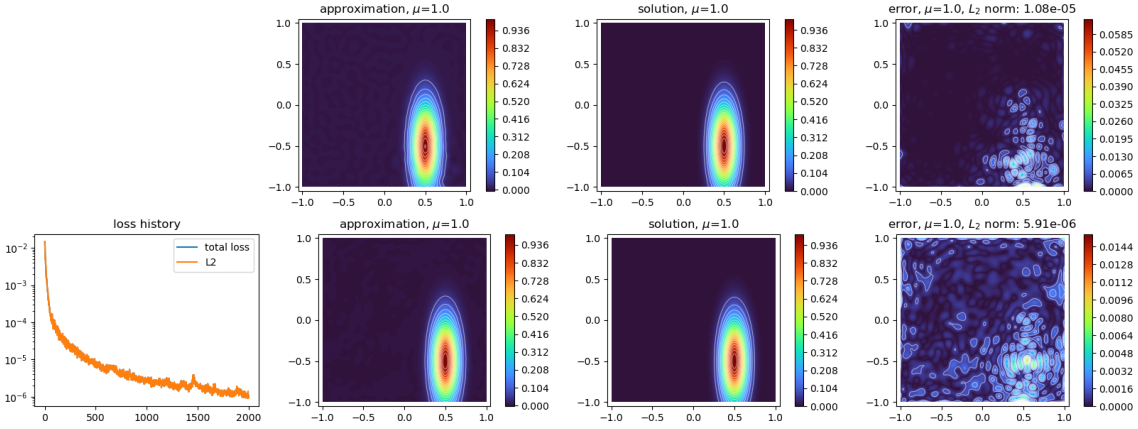

Figure 17: Approximation of an anisotropic function with Gaussian Kernels

In this particularly simple example 17, the general Gaussian Kernels give a good approximation of our anisotropic function.

Implementation

Finally, we implement the anisotropic Gaussian kernel.

First, we need to define a list of anisotropic matrix E as trainable parameters of the class.

```
M= torch.randn((self.centers.shape[0],self.centers.shape[1], self.centers.shape[1]), dtype=torch.get_default_dtype())
E= M @ M.transpose(-1,-2) + torch.eye(self.centers.shape[1], dtype=torch.get_default_dtype()) * 1e-5
self.M_aniso = nn.Parameter(E)
```

This ensure that our matrix E is symmetric positive definite.

And then in the forward method, we compute the anisotropic Gaussian kernel as follows:

```
elif self.kernel_type == "anisotropic_gaussian":
            # Calcul du noyau anisotrope
            vect_diff = torch.zeros((features_spatial.shape[0],self.centers.shape[0],2), dtype=torch.get_default_dtype())
            vect_diff = features_spatial.unsqueeze(1)-self.centers.unsqueeze(0)  # (batch_size, nb_centers, 2)

            transformed_distances = torch.einsum(
                'nkd,kdd,nkd->nk', vect_diff, self.M_aniso, vect_diff
            )
            #print("transformed_distances",transformed_distances.shape)

            anisotropic_basis = torch.exp(-transformed_distances)  # (batch_size, nb_centers)

            if with_last_layer:
                res = self.output_layer(anisotropic_basis)
            else:
                res = anisotropic_basis
```

Results

To test this kernel, I chose an anisotropic function f defined on the domain $\Omega = [-1, 1] \times [-1, 1]$:

$$f(x, y) = \exp^{\frac{1}{4} * \left(\left(\frac{x_{Rot}}{\sigma_x}\right)^2 + \left(\frac{y_{Rot}}{\sigma_y}\right)^2 \right)} \tag{16}$$

where $x_{Rot} = x\cos(\theta) + y\sin(\theta)$ and $y_{Rot} = -x\sin(\theta) + y\cos(\theta)$ are the rotated coordinates,
$\theta = \frac{\pi}{4}$,
$\sigma_x = 1$ and $\sigma_y = 0.3$

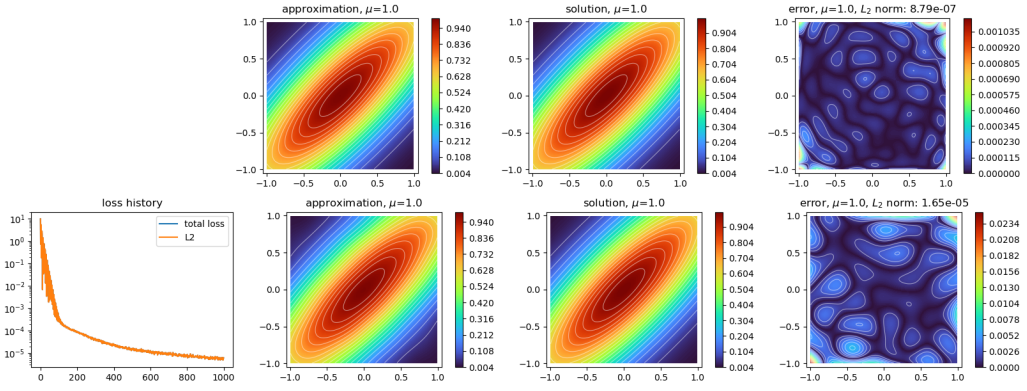

Figure 18: Anisotropic Gaussian RBF's approximation of the function f

In this simple test 18 of aproximating a gaussion anisotropic function with the anisotropic Gaussian kernel, the said kernel gives a good approximation of the function f.

I also tested it on our usual function $f(x, y) = \sin(\cos(x) \times \sin(y))/(2 \times 0.4^2)$:

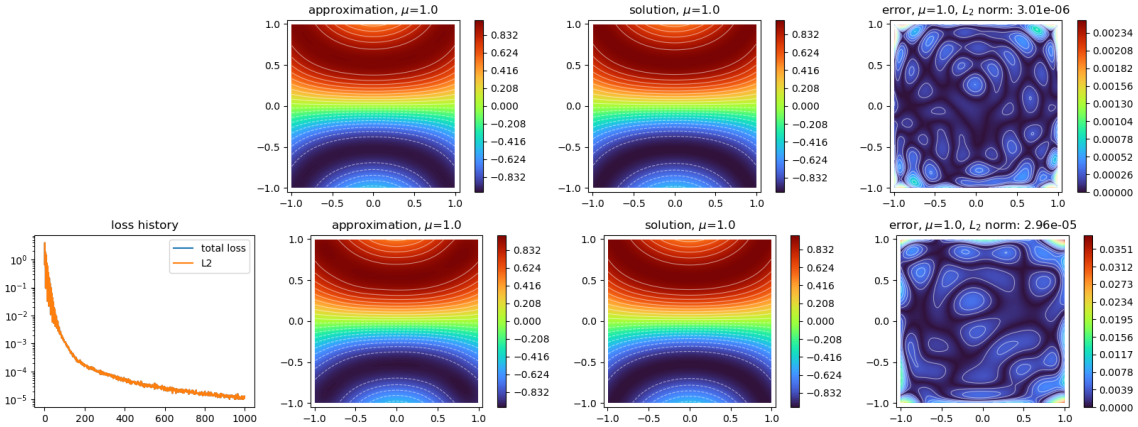

Figure 19: Anisotropic Gaussian RBF's approximation of the function f

The anisotropic Gaussian kernel also gives a very good approximation of the function f 19.

3.5 Results for Partial Differential Equations

In this section, I will present results for an elliptic Poisson PDE solved with the RBFs kernels implemented in Scimba :

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega = [-1, 1]^2, \\ u &= g \quad \text{on } \partial\Omega \end{aligned} \tag{17}$$

where $f = -4$ is a source term and $g(x, y) = x^2 + y^2$ is a boundary condition. To do so, I use already implemented classes in Scimba, which are the following:

- `Laplacian2DDirichlet_StrongForm` : for the definition of the PDE (The Poisson equation in its strong form)
- `PinnsElliptic` : for the solving of the PDE

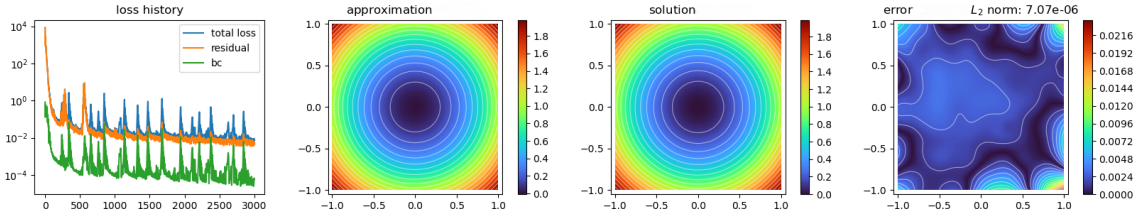

Figure 20: Gaussian RBF's approximation of the solution of the Poisson equation

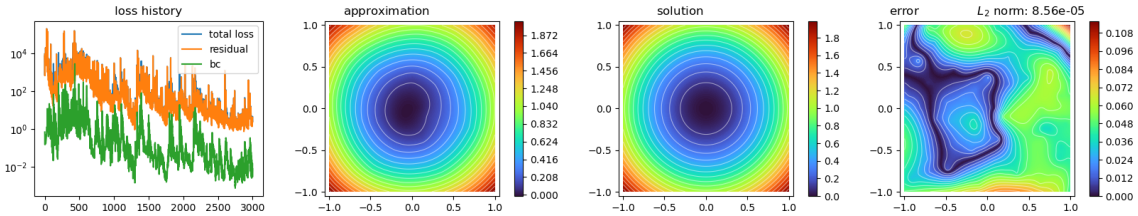

Figure 21: Exponential RBF's approximation of the solution of the Poisson equation

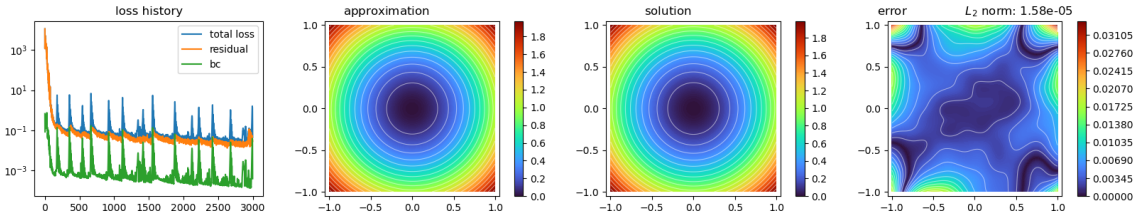

Figure 22: Multiquadratic RBF's with $\beta = -2$ approximation of the solution of the Poisson equation

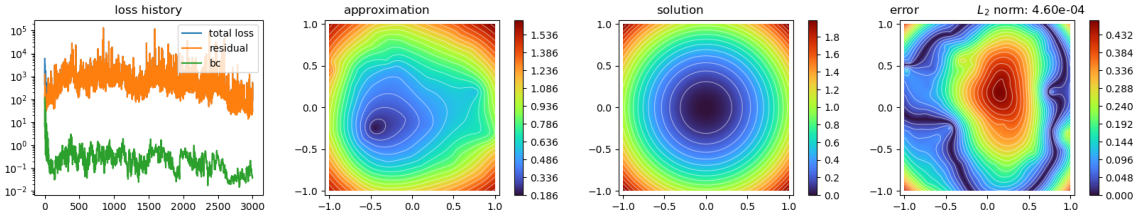

Figure 23: Matern RBF's with $\nu = 0.5$ approximation of the solution of the Poisson equation

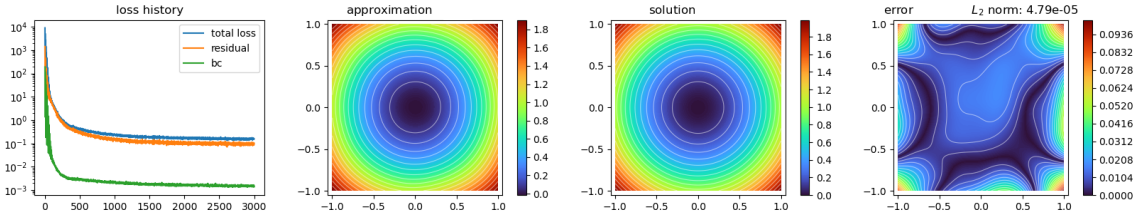

Figure 24: Matern RBF's with $\nu = 2.5$ approximation of the solution of the Poisson equation

Results

As predicted and shown with simple projections, the Gaussian kernels 20 give the best approximation of the solution of this Poisson equation. In deed, the exact solution of the Poisson equation being $u(x, y) = x^2 + y^2$ and the Gaussian RBFs kernel gives good approximations for $\mathcal{C}^\infty(\mathbb{R}^d)$ functions, it is the best choice for this problem.
The Matern RBFs kernel with $\nu = 2.5$ 24 and the Multiquadratic Kernel with $\beta = -2$ 22 give also a good approximation of the solution.
And last but not least, the exponential RBFs kernel 21 as well as the Matern Kernel with $\nu = 0.5$ 23 don't give as good approximation of the solution of the Poisson equation as the others.

4 Conclusion

This project introduced me to the concept of Kernel methods in machine learning, and more specifically to the Radial Basis Functions (RBFs) method. These state of the art methods are particularly studied in the context of meshfree methods for solving differential equations without the constraint of the dimension of the problem nore the geometry of the domain.

I was able to implement several RBFs kernels in the Scimba package and test them on simple 1D and 2D examples, as well as on a Poisson equation.
The results confirm some theoritical properties of RBFs kernels such as the importance of the choice of the kernel for the regularity of the approximant and thus the quality of the approximation of the solution of a PDE.
Now, it remains to test the implemented kernels on other examples of stationary and non-stationary differential equations.
Also, there are other kernels left to implement such as compactly supported kernels and oscillatory kernels wich I didn't have the time to implement.
In terms of RBFs kernels, the current implementation of the Matern Kernel doesn't take into account other values of ν than $0.5, 1.5$ `and` 2.5 because I use the GPyTorch library which only implements these values. It could be interesting to implement a more general Matern kernel that takes into account any theoriticaly valid values of ν. This particular kernel being a generalization of the RBFs kernel that generates the Sobolev spaces $H^{\nu+\frac{d}{2}}(\mathbb{R}^d)$, it could be useful for the approximation of functions in these spaces.

Finally, I would like to thank my supervisor, Dr. Emmanuel Franck and the PHD student Nicolas Paillez, for their help and support during this project.

References

[1] Stefano De Marchi. *A short tour of radial basis functions and meshless approximation*. 2024.

[2] Kernel Techniques: From Machine Learning to Meshless Methods. "Robert Schalback and Holger Wendland". In: *Acta Numerica* (2006). DOI: 10.1017/S0962492904000077.

[3] Ulrika Sundin. "Global radial basis function collocation methods for PDEs". PhD thesis. UPPSALA UNIVERSITY, 2019.