POLITECNICO
MILANO 1863

mdisd : a C++ library for multi-dimensional interpolation of scattered data

Advanced Programming for Scientific Computing Project
A.Y. 2023/2024

Author: Nils Malmberg
Supervisor: Prof. Nicola Parolini

Abstract

This report introduces the mdisd library, a C++ tool for multi-dimensional interpolation of scattered data, featuring implementations of radial basis functions (RBF) and ordinary least squares (OLS) methods. Theoretical foundations, implementation details, and performance evaluations of these interpolators are provided. Additionally, bindings for Python integration are discussed, aiming to enhance accessibility within Python-based workflows.

mdisd's Github : https://github.com/MalmbergNilsPolimi/mdisd

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Introduction

Interpolation is a widely used technique in various scientific and technical fields to estimate values between known data points. This report presents the development of the mdisd library, written in C++, for multi-dimensional interpolation of scattered data. This project aims to provide a tool for interpolating scattered data in multiple dimensions.

The mdisd library consists of implementations of two interpolation methods: radial basis functions (RBF) and ordinary least squares (OLS). These methods offer different approaches to data interpolation, each characterized by its advantages and applications.

Firstly, the theoretical foundations of interpolation are described, highlighting the underlying principles of the RBF and OLS methods. The advantages and limitations of each method are also discussed, along with typical use cases where they excel.

Next, our implementation of RBF and OLS interpolators in the C++ language is presented, using the Eigen library for matrix computation. The code architecture, class and function design, as well as the design choices made to ensure flexibility, ease of use, and performance, are detailed.

Finally, the performance of the interpolators is evaluated by testing them on data sets. Recommendations are made for the optimal use of these methods.

Additionally, bindings (with pybind11) have been implemented to enable the use of this library in Python, allowing for seamless integration into existing Python-based workflows and enabling a wider range of users to benefit from its capabilities.

This report aims to provide a comprehensive understanding of interpolation techniques based on RBF and OLS methods, as well as a practical implementation of these methods in a C++ programming environment through the mdisd library.

The mdisd library was completely developed from scratch as part of the "Advanced Programming for Scientific Computing" course held at Politecnico di Milano by professor Luca Formaggia.

1 Notations

Let's consider a system that depends directly on n parameters and returns an output quantity based on these n parameters. For example, consider a factory whose output quantity is the final product, or more precisely the quantity of finished products. The input parameters of this factory are, for example, the flow of raw materials, the state of fatigue of the employees, the state of the machines, and so on. The plant is represented by the function f, the number of finished products by o, and the input parameters by the vector $\mathbf{P}$.

$$\begin{tikzcd} {} \arrow[r, "\mathbf{P} = \begin{bmatrix} p_1 & p_2 & \dots & p_n \end{bmatrix}"] & \boxed{f} \arrow[r, "o = f(\mathbf{P})"] & {} \end{tikzcd}$$

It can therefore be difficult to create a model that takes all these parameters into account in order to estimate the quantity of finished products for the current state of the plant. For this reason, the method proposed here aims to interpolate the quantity of finished products from a collection of data made beforehand and from the current state of each of the parameters.

To do this, let's consider that the company has taken care to collect the factory parameters and the quantities of finished products corresponding to these parameters at several different time intervals (over several days, months, or years). Consider m measurements of these parameters and product quantities.

A known quantity of finished product o_i can therefore be associated with the corresponding state of the plant P_i for all i in $[\![1; m]\!]$. In matrix format, this gives :

$$P = \begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \\ \vdots \\ \mathbf{P}_m \end{pmatrix} = \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,n} \end{pmatrix} \quad O = \begin{pmatrix} o_1 \\ o_2 \\ \vdots \\ o_m \end{pmatrix} = \begin{pmatrix} f(\mathbf{P}_1) \\ f(\mathbf{P}_2) \\ \vdots \\ f(\mathbf{P}_m) \end{pmatrix}$$

The aim now is to estimate a new output $o_{\text{new}} = f(\mathbf{P}_{\text{new}})$ from a new input $\mathbf{P}_{\text{new}} = \begin{pmatrix} p_{\text{new},1} & p_{\text{new},2} & \cdots & p_{\text{new,n}} \end{pmatrix}$.

2 Ordinary Least Squares (OLS)

2.1 OLS method

Ordinary Least Squares (OLS) is a fundamental method in statistical modeling used to estimate the relationship between a dependent variable and one or more independent variables. Its essence lies in minimizing the sum of the squared of the differences between the observed and predicted values. Thus, OLS aims to find the line (in simple linear regression) or plane/hyperplane (in multivariate regression) that best fits the observed data points. This method is widely employed in various fields, including economics, social sciences, and engineering, to analyze and understand complex relationships between variables. In multivariate regressions, OLS extends its utility by accommodating multiple independent variables, allowing for the examination of how several factors collectively influence the dependent variable [1], [2].

Using the notations defined in the dedicated section 1, the goal is to find the following function that fits the known data best:

$$\begin{aligned}\widehat{o}_{\text{new}} &= \widehat{\alpha} + \widehat{\beta}_1 \times p_{\text{new},1} + \widehat{\beta}_2 \times p_{\text{new},2} + \cdots + \widehat{\beta}_n \times p_{\text{new},n} \\ &= \widehat{\alpha} + \sum_{i=1}^{n} \widehat{\beta}_i \times p_{\text{new},i}\end{aligned}$$

The main idea behind the ordinary least squares method is to choose the coefficients $\widehat{\alpha}, \widehat{\beta}_1, \cdots, \widehat{\beta}_n$ to minimize the following sum in order to have the smallest distance between the known points and the estimates of these same points by regression:

$$\sum_{i=1}^{m} \left[o_i - \left(\widehat{\alpha} + \widehat{\beta}_1 \times p_{\text{new},1} + \widehat{\beta}_2 \times p_{\text{new},2} + \cdots + \widehat{\beta}_n \times p_{\text{new},n} \right) \right]^2$$

Let's introduce the following matrix notations:

$$X = \begin{pmatrix} 1 & p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ 1 & p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\ | & \vdots & \vdots & \vdots & \vdots \\ 1 & p_{m,1} & p_{m,2} & \cdots & p_{m,n} \end{pmatrix} \quad \widehat{\beta} = \begin{pmatrix} \widehat{\alpha} \\ \widehat{\beta}_1 \\ \vdots \\ \widehat{\beta}_n \end{pmatrix} \quad O = \begin{pmatrix} o_1 \\ o_2 \\ \vdots \\ o_m \end{pmatrix}$$

Then, the value of $\widehat{\beta}$ which minimizes the sum is

$$\boxed{\widehat{\beta} = \left(X^T X\right)^{-1} X^T O} \tag{1}$$

2.2 Matrix inversion

Now that we have the expression for the coefficients, we need to choose the method for inverting the $X^T X$ matrix. There are several inversion methods based on matrix decomposition, such as LU, QR, PLU, SVD, etc. The SVD method will be discussed here as it seems to be suitable for our use case. The SVD method, or singular value decomposition, is a method for decomposing square or rectangular matrices whose coefficients belong to the $\mathbb{K}$ field ($\mathbb{K} = \mathbb{R}$ or $\mathbb{K} = \mathbb{C}$). This method is relatively stable numerically but can have a higher computational cost than PLU methods, for example. But here, the case study is devoted to relatively small datasets that do not require very large volumes of data to be processed.

The matrix $X^T X$ is factorized into $U \Sigma V^*$ where U is a unit matrix on $\mathbb{K}$, Σ is a matrix whose diagonal coefficients are positive real numbers or zero and the others are zero and V^* is the adjoint (conjugate transpose) matrix of V which is a unit matrix on $\mathbb{K}$ [3].

This method will not be discussed further. It will be implemented using the Eigen library, which already contains an implementation of this method.

3 Radial Basis Function (RBF)

3.1 RBF method

The Radial Basis Function (RBF) method is a powerful mathematical technique used in various fields, particularly in function approximation and interpolation tasks. At its core, the RBF method employs radial basis functions, which are mathematical functions whose values depend only on the distance from a specific point, known as the center. These functions are typically symmetric and decrease as the distance from the center increases, capturing the notion of similarity between data points.

In practice, the RBF method utilizes these functions to approximate complex relationships between input and output variables by expressing the output as a weighted sum of radial basis functions evaluated at input locations. This approach offers flexibility and adaptability, making it well-suited for tasks where traditional linear models may struggle to capture nonlinear patterns in the data.

In multivariate regressions, the RBF method extends its utility by accommodating multiple input variables, allowing for the modeling of complex relationships involving multiple predictors. By leveraging the inherent flexibility of radial basis functions, multivariate RBF regression can effectively capture intricate interactions and dependencies among predictor variables, enabling accurate prediction and analysis in diverse domains such as finance, engineering, and machine learning.

The problem is formulated as follows [4]:

$$\boxed{o_{\text{new}} = \sum_{i=1}^{m} \omega_i \cdot \phi(||\mathbf{P}_{\text{new}} - \mathbf{P}_i||)} \tag{2}$$

where ϕ is the radial basis function and ω_i are the weights. Since the separate section 3.4 is dedicated to these functions, they will not be discussed in greater detail here.

The data collected and known are used to determine the weights to be applied. Indeed, it is necessary to ensure that the estimate returned by the RBF method corresponds to the known results for their corresponding parameter set.

So,

$$\boxed{\forall i \in [\![1; m]\!], \qquad o_i = \sum_{k=1}^{m} \omega_k \cdot \phi(||\mathbf{P}_i - \mathbf{P}_k||)} \tag{3}$$

This system can be written in the following matrix form [5]:

$$\boxed{\Phi\Omega = O}$$

$$\text{where, } \Phi = \begin{pmatrix} \phi\left(||\mathbf{P}_1 - \mathbf{P}_1||\right) & \phi\left(||\mathbf{P}_2 - \mathbf{P}_1||\right) & \cdots & \phi\left(||\mathbf{P}_m - \mathbf{P}_1||\right) \\ \phi\left(||\mathbf{P}_1 - \mathbf{P}_2||\right) & \phi\left(||\mathbf{P}_2 - \mathbf{P}_2||\right) & \cdots & \phi\left(||\mathbf{P}_m - \mathbf{P}_2||\right) \\ \vdots & \vdots & \vdots & \vdots \\ \phi\left(||\mathbf{P}_1 - \mathbf{P}_m||\right) & \phi\left(||\mathbf{P}_2 - \mathbf{P}_m||\right) & \cdots & \phi\left(||\mathbf{P}_m - \mathbf{P}_m||\right) \end{pmatrix} \quad \text{and} \quad \Omega = \begin{pmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_m \end{pmatrix}$$

So by using a matrix inversion method, such as the SVD decomposition discussed earlier 2.2, it is possible to determine each weight and therefore estimate o_{new}.

3.2 Normalized Radial Basis Function (NRBF)

The NRBF method is a variant of the RBF method, which consists of making the sum of the basis functions unity. The equations (2) and (3) thus become :

$$\boxed{o_{\text{new}} = \frac{\sum_{i=1}^{m} \omega_i \cdot \phi(||\mathbf{P}_{\text{new}} - \mathbf{P}_i||)}{\sum_{i=1}^{m} \phi(||\mathbf{P}_{\text{new}} - \mathbf{P}_i||)}} \tag{4}$$

and,

$$\boxed{\forall i \in [\![1;m]\!], \quad o_i \cdot \sum_{k=1}^{m} \phi(||\mathbf{P}_i - \mathbf{P}_k||) = \sum_{k=1}^{m} \omega_k \cdot \phi(||\mathbf{P}_i - \mathbf{P}_k||)} \tag{5}$$

There's no proof that either the NRBF or RBF method consistently outperforms the other. Both methods can be easily implemented in the same code, giving the user the freedom to choose between them [4].

3.3 Radial Basis Function augmented with Polynomials (RBFP)

Another variant of the RBF method is the RBF method with the addition of a polynomial term. Here, the notation RBFP will be used, with P denoting the polynomial term. Adding polynomial terms in the RBF method can be beneficial in certain scenarios for several reasons. Firstly, it can enhance the model's capacity to capture complex nonlinear relationships in the data. By incorporating polynomial terms, the model becomes more flexible and capable of fitting data that exhibit nonlinear behavior more accurately. Additionally, the inclusion of polynomial terms can help mitigate the issue of underfitting, especially when the dataset contains intricate patterns that cannot be adequately captured by linear or radial basis functions alone. Moreover, this augmentation can lead to improved generalization performance, allowing the model to extrapolate more effectively beyond the training data.

While adding additional polynomial terms to the RBF method can offer advantages, there are also drawbacks to consider. One significant disadvantage is the increased risk of overfitting, especially when the degree of the polynomial is too high relative to the complexity of the data. This overfitting can lead to poor generalization performance, where the model performs well on the training data but fails to accurately predict unseen data. Moreover, the inclusion of polynomial terms can result in a more complex model structure, making it computationally expensive and potentially harder to interpret. Additionally, determining the appropriate degree of the polynomial and the number of additional terms requires careful tuning, which can be time-consuming and resource-intensive. Furthermore, in some cases, the presence of polynomial terms may introduce numerical instability or sensitivity to noise in the data, leading to less reliable model predictions.

Using Vaclav Skala's formulation of the problem [6]:

let's consider an additional linear polynomial $Q(x_1, \cdots, x_n) = a_0 + \sum_{i=1}^{n} a_i x_i$ from $\mathbb{R}^n$ to $\mathbb{R}$. Then, o_{new} can be written as:

$$\boxed{o_{\text{new}} = \sum_{i=1}^{m} \omega_i \cdot \phi(||\mathbf{P}_{\text{new}} - \mathbf{P}_i||) + Q\left(\mathbf{P}_{\text{new}}\right)} \tag{6}$$

In addition to equation (3), which is used to determine the weights ω_k, the system must now consider the following new constraint:

$$\boxed{Q(\mathbf{P}_i) = a_0 + \sum_{k=1}^{n} a_k \cdot p_{i,k} = 0 \quad , \quad \forall i \in [\![1;m]\!]} \tag{7}$$

Which can lead to the following matrix system:

$$\begin{pmatrix} \phi\left(\|\mathbf{P}_1-\mathbf{P}_1\|\right) & \phi\left(\|\mathbf{P}_2-\mathbf{P}_1\|\right) & \cdots & \phi\left(\|\mathbf{P}_m-\mathbf{P}_1\|\right) & p_{1,1} & \cdots & p_{1,n} & 1 \\ \phi\left(\|\mathbf{P}_1-\mathbf{P}_2\|\right) & \phi\left(\|\mathbf{P}_2-\mathbf{P}_2\|\right) & \cdots & \phi\left(\|\mathbf{P}_m-\mathbf{P}_2\|\right) & p_{2,1} & \cdots & p_{2,n} & 1 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots & \vdots \\ \phi\left(\|\mathbf{P}_1-\mathbf{P}_m\|\right) & \phi\left(\|\mathbf{P}_2-\mathbf{P}_m\|\right) & \cdots & \phi\left(\|\mathbf{P}_m-\mathbf{P}_m\|\right) & p_{m,1} & \cdots & p_{m,n} & 1 \\ p_{1,1} & p_{2,1} & \cdots & p_{m,1} & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots & \vdots \\ p_{1,n} & p_{2,n} & \cdots & p_{m,n} & 0 & \cdots & 0 & 0 \\ 1 & 1 & \cdots & 1 & 0 & \cdots & 0 & 0 \end{pmatrix} \times \begin{pmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_m \\ a_1 \\ \vdots \\ a_n \\ a_0 \end{pmatrix} = \begin{pmatrix} o_1 \\ o_2 \\ \vdots \\ o_m \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}$$

To take the use of polynomials even further, it can be interesting to consider not just linear polynomials but all polynomials of degree d. For example, consider a function with two variables x and y. Then, for different degrees, the polynomials can be written:

- $d = 1$: $Q_{d=1}(x, y) = x + y + 1$;
- $d = 2$: $\begin{aligned} Q_{d=2}(x, y) &= x^2 + y^2 + xy + x + y + 1 \\ &= x^2 + y^2 + xy + Q_{d=1}(x, y) \end{aligned}$
- $d = 3$: $\begin{aligned} Q_{d=3}(x, y) &= x^3 + y^3 + x^2y + xy^2 + x^2 + y^2 + xy + x + y + 1 \\ &= x^3 + y^3 + x^2y + xy^2 + Q_{d=2}(x, y) \end{aligned}$

The number of variables will therefore be fixed by the number of parameters, but the degree can be adjusted by the user. For the moment, only linear polynomials are implemented in the library.

3.4 Radial basis functions

Radial Basis Functions (RBFs) are a class of mathematical functions commonly used in interpolation, approximation, and machine learning tasks. Unlike other basis functions that are often defined over a finite interval or region, RBFs extend indefinitely from a central point, spreading their influence radially. This property makes them particularly useful for problems involving scattered data or irregularly spaced inputs.

At the heart of RBFs lies their radial symmetry, meaning that their value depends only on the distance from a center point, rather than on the direction. This characteristic simplifies their computation and makes them highly adaptable to various applications.

RBFs are often employed in interpolation tasks, where they approximate a function given a set of input-output pairs. The interpolation process involves selecting appropriate centers and determining the weights associated with each center. Once these parameters are determined, the RBF interpolant can approximate the function at any point in the input space.

Despite their effectiveness, RBFs come with challenges, particularly in determining the optimal placement of centers and adjusting parameters like the scale factor (r_0) to achieve desired performance.

The radial basis functions implemented in the library are listed in Table 1.

Multiquadratic	$\phi(r) = (r^2 + r_0^2)^{1/2}$	This function is suitable for cases where a smooth interpolation or approximation is needed, and the distance from the center to the data points varies. It's important to choose an appropriate value for the parameter r_0 to control the smoothness of the function.
Inverse multiquadratic	$\phi(r) = (r^2 + r_0^2)^{-1/2}$	Similar to the multiquadratic function, the inverse multiquadratic function is suitable for smooth interpolation or approximation tasks. However, it places more emphasis on data points further away from the center due to its inverse relationship. As with the multiquadratic function, choosing an appropriate value for r_0 is crucial for achieving the desired smoothness.
Thin-plate spline	$\phi(r) = r^2 \log\left(\frac{r}{r_0}\right),$ $\phi(0) = 0$	This function is useful when both smoothness and flexibility are required in the interpolation or approximation process. It's particularly suitable for cases where the underlying function being approximated may exhibit complex behavior, such as sharp changes or curvatures. The condition $\Phi(0) = 0$ ensures that the function is well-behaved at the center.
Gaussian	$\phi(r) = \exp\left(-\frac{1}{2} \cdot \frac{r^2}{r_0^2}\right)$	The Gaussian function is commonly used when a localized influence around the center is desired, with rapid decay as the distance from the center increases. It's well-suited for cases where data points closer to the center should have a stronger influence on the interpolation or approximation. The parameter r_0 controls the width of the bell-shaped curve, with smaller values resulting in narrower curves and vice versa.

Table 1: Radial basis functions implemented in the mdisd library.

4 Data pre-processing: rescaling

Interpolation methods based on the distance between points to be interpolated and known points, such as RBF methods, are relatively sensitive to parameter anisotropy. The term anisotropy is used here to imply a greater or lesser variation in the value taken by a parameter. For example, suppose a parameter is the temperature of an oven in a factory. This parameter can take on discrete values ranging from an ambient temperature of 25 degrees Celsius, for example, to over 1000 degrees Celsius. If the measurements are taken at, say, 50, 90, 30, 1300, 400, 70, 950... then we observe a dispersion in the values taken by the parameter. If this also occurs for several or all of the other parameters, then the evaluation of the weights and therefore the interpolation may be distorted. Remember that the primary aim is to estimate the values that the function can take outside the known points.

One way of reducing the undesirable effect of the anisotropy of the parameters is to reduce the value taken by these parameters and, for example, apply a transformation T to the set of parameters so that each parameter is between 0 and 1. One difficulty with this method is to retain the effect of each parameter on the result. In addition, if a rescaling is performed on the known parameter values, then the coefficients need to be stored to perform the same rescaling on the parameter sets whose values are to be interpolated by the f function.

To achieve this, several rescaling methods are implemented in the mdisd library to give the user an additional tool for obtaining the best possible interpolation. Of course, it is important to bear in mind that the addition of calculations by this pre-processing will have an impact on the complexity and number of calculations, thus impacting on the calculation time required for an interpolation.

- Min-Max normalization [7]: $T(p_{i,j}) = \dfrac{p_{i,j} - \min(p_{1,j}, \cdots, p_{m,j})}{\max(p_{1,j}, \cdots, p_{m,j}) - \min(p_{1,j}, \cdots, p_{m,j})}$;
- Mean normalization [8]: $T(p_{i,j}) = \dfrac{p_{i,j} - \mu_j}{\max(p_{1,j}, \cdots, p_{m,j}) - \min(p_{1,j}, \cdots, p_{m,j})}$, where $\mu_j = \dfrac{1}{m} \sum_{l=1}^{m} p_{l,j}$;
- Z-score normalization [9]: $T(p_{i,j}) = \dfrac{p_{i,j} - \mu_j}{\sigma_j}$, where $\sigma_j = \sqrt{\dfrac{1}{m} \sum_{l=1}^{m} p_{l,j}^2 - \left[\dfrac{1}{m} \sum_{l=1}^{m} p_{l,j}\right]^2}$.

Other functions that are sometimes used in data analysis, statistics, or machine learning can be mentioned (not available in the library):

- the tan^{-1} or $atan$ function [9]: $T(p_{i,j}) = \dfrac{1}{2}\left[1 + tan^{-1}(p_{i,j})\right]$ or $T(p_{i,j}) = \dfrac{1}{2}\left[1 + tan^{-1}\left(0.01 \cdot \dfrac{p_{i,j} - \mu_j}{\sigma_j}\right)\right]$;
- the sigmoid function [10]: $T(p_{i,j}) = \dfrac{1}{1 + e^{-p_{i,j}}}$;
- the softmax function [11]: $T(p_{i,j}) = \dfrac{e^{-p_{i,j}}}{\sum\limits_{l=1}^{m} e^{-p_{l,j}}}$.

Finally, we can say that any function of $\mathbb{R}$ in $[0, 1]$ can be used for data rescaling. But in reality, the choice of the rescaling function is very important and can also render the interpolation results non-relevant. For example, let's assume that the rescaling function is $T(p_{i,j}) = 1$, in which case all parameters have the same value all the time, so the weights will be identical and the interpolation unusable. It's easy to see that the chosen function will have a major impact on the interpolation result. But, as explained earlier, to perform interpolation, it's not enough to choose a method at random, select a radial basis function and a scale factor at random, and add a pre-processing at random. A successful interpolation requires a preliminary study to get an idea of the expected result, followed by an in-depth study of the radial base functions and the scaling coefficients to obtain the "optimum" parameters. So, for rescaling, it's also necessary to carry out tests, and compare the case of study with the existing literature... to be in the best possible conditions.

5 The library

5.1 General structure

```
mdisd
├── LICENSE
├── README.md
├── CMakeLists.txt
├── doc
├── include
│   ├── OLS
│   │   └── OLSinterpolator.hpp
│   ├── RBF
│   │   ├── RBFinterpolator.hpp
│   │   └── RBFunctions.hpp
│   ├── interpolator.hpp
│   └── rescaling.hpp
├── src
│   ├── OLS
│   │   └── OLSinterpolator.cpp
│   ├── RBF
│   │   ├── RBFinterpolator.cpp
│   │   └── RBFunctions.cpp
│   ├── rescaling.cpp
│   └── bindings.cpp
└── test
    ├── case0
    │   ├── Makefile
    │   └── main.cpp
    ├── case1
    │   ├── Makefile
    │   └── main.cpp
    ├── case2
    │   ├── Makefile
    │   └── main.cpp
    ├── case3
    │   ├── Makefile
    │   └── main.cpp
    ├── case4
    │   ├── Makefile
    │   └── main.cpp
    ├── case5
    │   ├── Makefile
    │   └── main.cpp
    └── case6
        └── test_python.py
```

Figure 1: Tree of the mdisd library.

5.2 Classes

5.2.1 *Interpolator*

Base Interpolation Method class (*interpolator.hpp*):

i. Role: This class serves as an abstract base class for all specific interpolation methods.

ii. Why: By defining a base class, we create a common interface for all interpolation methods, making it easy to use them and encapsulate them in generic containers.

```cpp
class Interpolator {

public:

    virtual Eigen::VectorXd interpolate(const Eigen::MatrixXd& parametersFORinterp,
                                        const Eigen::MatrixXd& parameters,
                                        const Eigen::VectorXd& measurements)
                                        const = 0;

    virtual Eigen::VectorXd interpolate(const Eigen::MatrixXd& parametersFORinterp,
                                        const Eigen::MatrixXd& parameters,
                                        const Eigen::VectorXd& measurements,
                                        Eigen::VectorXd* regression)
                                        const = 0;
};
```

Listing 1: ***Interpolator*** **class.**

5.2.2 *RBFInterpolator*

Specific RBF interpolation Method class (*RBFinterpolator.hpp* and *RBFinterpolator.cpp*) :

i. Role: This class implements a specific interpolation method based on Radial Basis Functions (RBF).

ii. Why: By encapsulating the RBF interpolation logic in a dedicated class, we can isolate this logic, facilitate its reuse, and allow customization specific to this method.

```cpp
class RBFInterpolator : public Interpolator {

private:

    std::function<double(double, double)> rbfunction;
    double r0;
    bool normalizeRBF;
    bool polynomialRBF;

public:

    RBFInterpolator(std::function<double(double, double)> rbfunction,
                    double r0, bool normalizeRBF=false, bool polynomialRBF=false)
                    : rbfunction(rbfunction), r0(r0), normalizeRBF(normalizeRBF),
                      polynomialRBF(polynomialRBF) {}

    Eigen::VectorXd interpolate(const Eigen::MatrixXd& parametersFORinterp,
                                const Eigen::MatrixXd& parameters,
                                const Eigen::VectorXd& measurements)
                                const override;

    Eigen::VectorXd interpolate(const Eigen::MatrixXd& parametersFORinterp,
                                const Eigen::MatrixXd& parameters,
                                const Eigen::VectorXd& measurements,
                                Eigen::VectorXd* regression)
                                const override;
};
```

Listing 2: ***RBFInterpolator*** **class.**

5.2.3 *RBFunctions*

Specific radial basis functions class (*RBFunctions.hpp* and *RBFunctions.cpp*) :

i. Role: This class implements pre-defined radial basis functions for the RBF interpolation method.

ii. Why: By encapsulating the radial basis functions in a dedicated class, we can easily choose which function to apply and add new ones.

```
class RBFunctions {

public:

    static double multiquadratic(double r, double r0);

    static double inverseMultiquadratic(double r, double r0);

    static double gaussian(double r, double r0);

    static double thinPlateSpline(double r, double r0);
};
```

Listing 3: *RBFunctions* class.

5.2.4 *OLSInterpolator*

Specific OLS interpolation Method class (*OLSinterpolator.hpp* and *OLSinterpolator.cpp*) :

i. Role: Same as RBF specific class.

ii. Why: Same as RBF specific class.

```
class OLSInterpolator : public Interpolator {

public:

    Eigen::VectorXd interpolate(const Eigen::MatrixXd& parametersFORinterp,
                                const Eigen::MatrixXd& parameters,
                                const Eigen::VectorXd& measurements)
                                const override;

    Eigen::VectorXd interpolate(const Eigen::MatrixXd& parametersFORinterp,
                                const Eigen::MatrixXd& parameters,
                                const Eigen::VectorXd& measurements,
                                Eigen::VectorXd* regression)
                                const override;
};
```

Listing 4: *OLSInterpolator* class.

5.2.5 *Rescaling*

Specific Rescaling Method class (*rescaling.hpp* and *rescaling.cpp*) :

i. Role: This class implements a data rescaling methods based.

ii. Why: By encapsulating the rescaling logic in a dedicated class, we can isolate this logic, facilitate its reuse, and allow customization specific to this method.

```cpp
class Rescaling {

public:

    std::pair<Eigen::MatrixXd, Eigen::MatrixXd> meanNormalization(
                            const Eigen::MatrixXd& data1,
                            const Eigen::MatrixXd* data2 = nullptr);

    std::pair<Eigen::MatrixXd, Eigen::MatrixXd> minMaxNormalization(
                            const Eigen::MatrixXd& data1,
                            const Eigen::MatrixXd* data2 = nullptr);

    std::pair<Eigen::MatrixXd, Eigen::MatrixXd> zScoreNormalization(
                            const Eigen::MatrixXd& data1,
                            const Eigen::MatrixXd* data2 = nullptr);

private:

    Eigen::VectorXd computeColumnMeans(const Eigen::MatrixXd& data);

    Eigen::VectorXd computeColumnStdDevs(const Eigen::MatrixXd& data,
                                          const Eigen::VectorXd& means);

    Eigen::VectorXd computeColumnMin(const Eigen::MatrixXd& data);

    Eigen::VectorXd computeColumnMax(const Eigen::MatrixXd& data);
};
```

Listing 5: *Rescaling* class.

5.3 Algorithms

This section explains how interpolation methods are implemented in the mdisd library. How the user defines the variables and how he can use the various interpolation methods are not discussed here, as they are covered in section 8.

Using the previous notations, *parameters* designates the matrix P and *measurements* designates the matrix O. *parametersFORinterp* will designate the matrix containing the coordinates of a point to be interpolated on each row. The result of the interpolation will be stored in the *results* variable.

5.4 OLS interpolation

Algorithm 1: Ordinary Least Squares (OLS) interpolation method

```
Data: parametersFORinterp, parameters, measurements
Result: results
Input  : sets of parameters for interpolation, known parameters sets, known measurements
Output: interpolated results

   /* Calculate the number of parameters, known measurements, and points to interpolate */
1  $num\_params \leftarrow$ number of columns of $parameters$;
2  $num\_measures \leftarrow$ size of $measurements$;
3  $num\_points \leftarrow$ number of rows of $parametersFORinterp$;

   /* Create a new matrix $X$ with an additional column of ones in the first position */
4  $X \leftarrow$ [1, parameters];

   /* Calculate the transpose of $X$ */
5  $transpose\_X \leftarrow X^T$;

   /* Decompose the matrix $X^TX$ using SVD */
6  $U, \Sigma, V^* \leftarrow \mathrm{SVD}(X^TX)$;

   /* Invert the matrix $X^TX$ */
7  $(X^TX)^{-1} \leftarrow V^*\Sigma^{-1}U^T$;

   /* Calculate the $\beta$ coefficients */
8  $\beta \leftarrow (X^TX)^{-1}X^TO$;

   /* Store the $\beta$ coefficients if a regression pointer is provided */
9  if $regression$ is not null then
10     $regression \leftarrow \beta$;
11 end

   /* Calculate the interpolated values */
12 for $i$ from 0 to $num\_points - 1$ do
13     for $j$ from 0 to $num\_params - 1$ do
14         $results(i) \leftarrow \beta(0) + \beta(j+1) \times parametersFORinterp(i,j)$;
15     end
16 end
```

5.5 RBF interpolation

Algorithm 2: Radial Basis Function (RBF) Interpolation Method

```
Data: parametersFORinterp, parameters, measurements
Result: results
Input : sets of parameters for interpolation, known parameter sets, known measurements
Output: interpolated results

/* Calculate the number of parameters, known measurements, and points to interpolate */
1  num_params ← number of columns of parameters;
2  num_measures ← size of measurements;
3  num_points ← number of rows of parametersFORinterp;

/* Initialize the results vector */
4  Initialize results as a vector of zeros of size num_points;

/* Compute the weights */
5  Initialize coeff as a matrix of zeros with appropriate dimensions;
6  for i from 0 to num_measures − 1 do
7      for j from 0 to num_measures − 1 do
8          coeff(i, j)← $\phi(||parameters(i) - parameters(j)||, r_0)$;
9      end
10     if polynomialRBF is true then
11         for k from num_measures to num_measures + num_params − 1 do
12             coeff(i, k) ← parameters(i, k-num_measures);
13             coeff(k, i) ← parameters.tanspose(k-num_measures, i);
14         end
15         coeff(i, num_measures+num_params) ← 1;
16         coeff(num_measures+num_params, i) ← 1;
17     end
18 end

/* Solve the system using SVD to find the weights */
19 if normalizeRBF then
20     for i from 0 to num_measures − 1 do
21         Initialize sum1 as 0;
22         for j from 0 to num_measures − 1 do
23             sum1 ← sum1 + $\phi(||parameters(i) - parameters(j)||, r_0)$;
24         end
25         NEWmeasurements(i) ← measurements(i) × sum1;
26     end
27     weights ← SVD(NEWmeasurements);
28 end
29 else if polynomialRBF then
30     for i from 0 to num_measures − 1 do
31         NEWmeasurements(i) ← measurements(i);
32     end
33     for i from num_measures to num_measures + num_params do
34         NEWmeasurements(i) ← 0;
35     end
36     weights ← SVD(NEWmeasurements);
37 end
38 else
39     weights ← SVD(measurements);
40 end
```

```
/* Store the weights if a regression pointer is provided */
if regression is not null then
    regression ← weights;
end

/* Compute the interpolated values */
for k from 0 to num_points − 1 do
    if normalizeRBF is true then
        normalize_part(k) ← 0;
        for l from 0 to num_measures − 1 do
            normalize_part(k) ← normalize_part(k) + $\phi(||parametersFORinterp(k) - parameters(l)||, r_0)$;
        end
    end
    for l from 0 to num_measures − 1 do
        results(k) ← weights(l)$\times\phi(||parametersFORinterp(k) - parameters(l)||, r_0)$/normalize_part(k);
    end
    if polynomialRBF is true then
        for i from num_measures to num_measures + num_params − 1 do
            results(k) ← results(k)+weights(i)×parametersFORinterp(k, i-num_measures);
        end
        results(k) ← results(k)+weights(num_measures+num_params);
    end
end
```

6 Test of the library

6.1 Case 0: radial basis functions

This folder allows the user to create the curves of the various radial basis functions pre-implemented in the library to see their shape and the influence of the scale factor r_0 on their shape.

After compiling and executing the contents of the file, it will be able to obtain the following graphics:

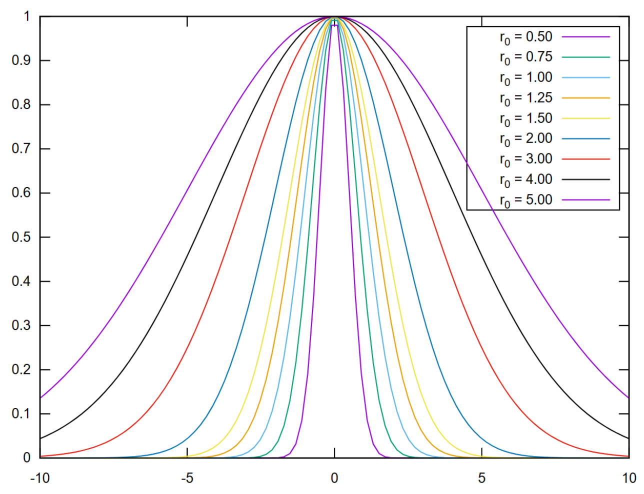

Figure 2: Gaussian basis function.

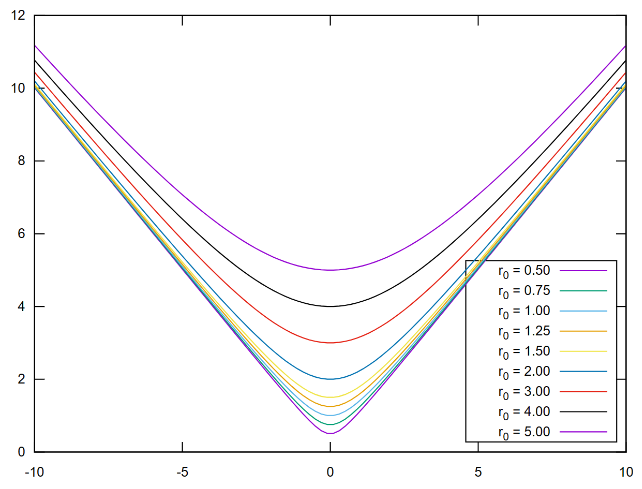

Figure 3: Multiquadratic basis function.

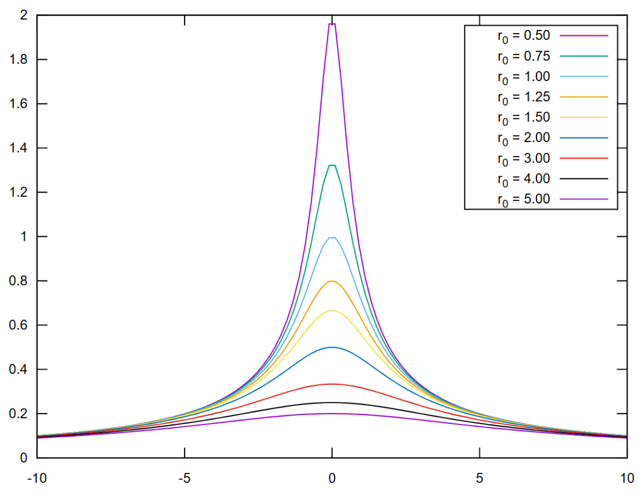

Figure 4: Inverse multiquadratic basis function.

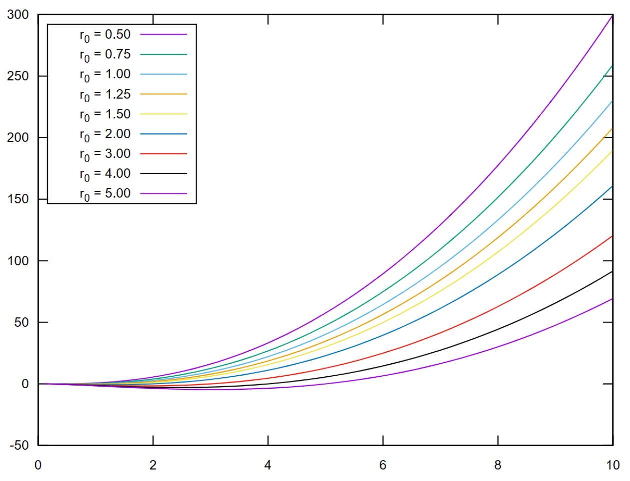

Figure 5: Thin plate spline basis function.

6.2 Case 1: a 1D test case

In this test case, the aim was to compare the results obtained by the library with one of the values found in the literature. For this purpose, to have a meaningful graphical representation, we will consider a one-dimensional case taken from Wilna du Toit's Master Thesis report [5]. In this report, Wilna du Toit considered a function f with one variable, x. The known points of f are shown in the table below.

x	1	3	3.5
f(x)	1	0.2	0.1

Table 2: Known points of the function f.

Then, du Toit chooses to use the Gaussian radial basis function with a scale factor equal to $1/\sqrt{2}$ leading to $\phi(r) = e^{-r^2}$. After computing the weights of the simple RBF method, the mdisd library returns some results near the one obtained by Wilna du Toit :

mdisd's weights	0.995	0.268	-0.111
du Toit's weights	0.995	0.268	-0.111

Table 3: Weights obtained after RBF interpolation.

Using the weights, it is possible to interpolate the function f over a given interval. The purpose of the figure below was to reproduce figure 2.2 by Wilna du Toit to show the contribution of each radial basis function when interpolating the f function.

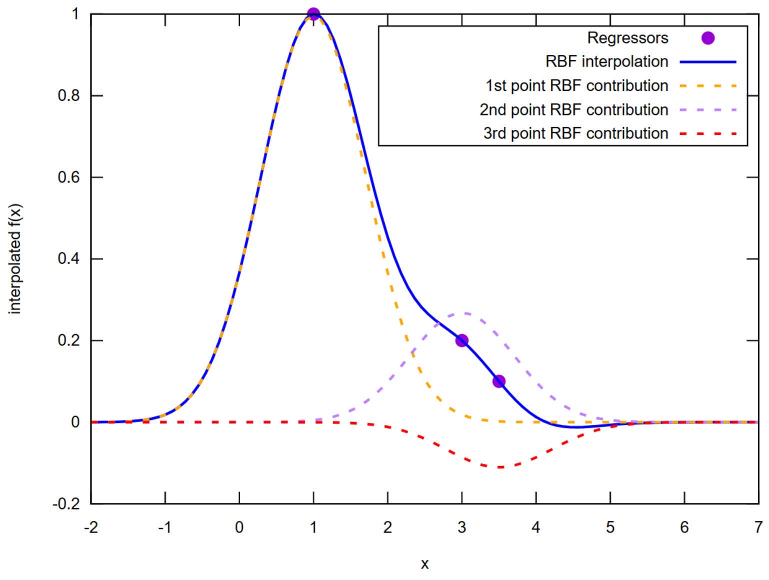

Figure 6: Reproduction of figure 2.2 from Wilna du Toit [5] showing the contribution of each radial basis function to the interpolation.

Then another graph was created to compare the interpolation methods implemented in the library (RBF, NRBF, RBFP and OLS) and see how they differ in a simple one-dimensional case.

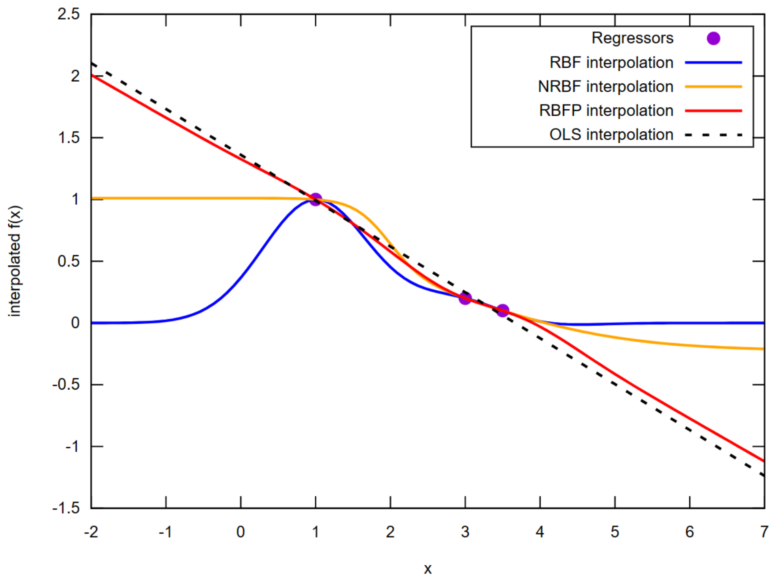

Figure 7: Comparison of RBF, NRBF, RBFP and OLS methods in a simple 1D case.

As expected, the OLS method, being a simple linear regression, simply draws a straight line minimizing the sum of the right/left deviations. For the NRBF method (normalized RBF), this method appears to pass through the three known points, as does the RBFP method (RBF augmented with linear polynomials), but it differs from the simple RBF method outside the known zones. This is why it is important to have a prior idea of the behavior of the function to be interpolated, as many factors will influence the accuracy and consistency of the interpolation: the interpolation method used, if an RBF method is chosen, the radial basis function used and the value of the scaling factor.

6.3 Case 2: a 1D linear test case

This case study aims to demonstrate the adaptability of more complex RBF-type methods for interpolating linear functions. Again to have a visual study, here a one-dimensional case will be studied. The function from $\mathbb{R}$ to $\mathbb{R}$, $f : x \to 0.5x - 4.3$ will be defined for interpolation. For RBF methods, a multiquadratic radial basis function with $r_0 = 0$ was used.

Five known points will be considered, as defined in the table below:

x	-2	3.7	0.1	-6	18.2
$f(x)$	$f(-2)$	$f(3.7)$	$f(0.1)$	$f(-6)$	$f(18.2)$

Table 4: Known points of the function f.

After interpolation, the weights returned by each method are as follows:

Method	Weights / Coefficients						
RBF	2.67×10^{-16}	-2.84×10^{-16}	-1.49×10^{-16}	0.198	-0.302		
NRBF	-8.83	9.33	-3.88	20.0	-14.6		
RBFP	-3.38×10^{-17}	4.83×10^{-16}	-2.098×10^{-16}	-3.89×10^{-16}	-4.025×10^{-16}	0.5	-4.3
OLS	-4.3	0.5					

Table 5: Weights / Coefficients for the interpolation of f.

When reading the weights, it's easy to see that the OLS method is accurate and efficient in this type of case, since it only requires the calculation of two weights, and in the end, the weights have the same value as those expected. Concerning the RBF method, it's difficult to give an accurate opinion just by looking at the weights; a graphical study would be more appropriate. For the NRBF method, it's the same, except that it's easily noticeable that the weights have a much higher value than those of the RBF method, so a difference between the two methods is expected in terms of graphic visualization. Finally, the RBFP method seems to work well, since it gives very low weights to the weights associated with the radial basis functions, compared with the weights given to the added linear polynomial (the last two weights). Looking at the values more closely, it's easy to see that this method will make the polynomial predominate over the radial basis functions and that this polynomial has the same coefficients as those defined in the f function.

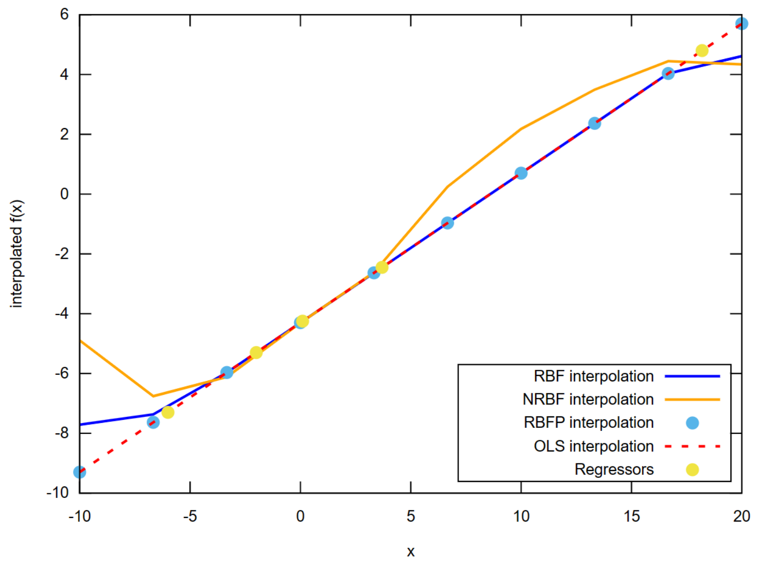

Figure 8: Interpolation of f, a linear 1D function.

Graphically, it can be seen that the OLS and RBFP methods work quite well. The simple RBF method also works, except at the two extremes, for regressors with a single neighbor. As for the NRBF method, it doesn't seem at all suited to this situation and only seems to fit the f function over a restricted interval around $x = 0$.

6.4 Case 3: an "isotropic" 4D test case

This case study consists of studying a known multivariate function and observing the discrepancies between the interpolated values and the actual values taken by the function. For this purpose, a function with four variables is defined, and 150 points are known. The interpolation will be based on the evaluation of 10 points.

The process of creating known points and interpolated points is randomized. To avoid too great a discrepancy between the values taken by the parameters, and thus to combat the anisotropy of the values taken by the parameters, it will be chosen that each parameter will have a definition domain between 0 and 1. In this case study, the function chosen will be as follows. f is a function from $\mathbb{R}^4$ to $\mathbb{R}$ written :

$$f(\mathbf{x}) = f(x_1, x_2, x_3, x_4) = \frac{3}{2} \cdot x_1^2 \cdot \cos(x_2 \cdot \pi) \cdot \sin(x_4 - x_3) - x_4 \cdot \exp\left(-\frac{x_1 + x_3}{2}\right) + \log(5 \cdot |x_3|) - 18.12 \cdot |x_2|^{x_4} \quad (8)$$

Thus, after compilation and execution of the test case, the results of the interpolations are transcribed in the following table.

n°	f($\mathbf{x}$)	RBF($\mathbf{x}$)	ϵ_{RBF}	NRBF($\mathbf{x}$)	ϵ_{NRBF}	RBFP($\mathbf{x}$)	ϵ_{RBFP}	OLS($\mathbf{x}$)	ϵ_{OLS}
1	-11.5	-11.4	0.0139	-11.3	0.0164	-11.3	0.0149	-10.0	0.132
2	-1.87	-1.81	0.0329	-1.83	0.0211	-1.82	0.0293	-3.90	-1.08
3	-11.4	-11.6	-0.0158	-11.5	-0.00907	-11.7	-0.0196	-11.9	-0.0376
4	-16.8	-17.1	-0.0185	-17.1	-0.0157	-17.1	-0.0172	-21.6	-0.282
5	-14.5	-14.6	-0.00773	-14.6	-0.00506	-14.6	-0.00683	-12.6	0.127
6	-7.21	-7.03	0.0255	-7.07	0.0198	-6.96	0.0356	-7.22	-0.000742
7	-7.88	-8.03	-0.0192	-7.91	-0.00315	-8.04	-0.0202	-8.64	-0.0967
8	-7.65	-7.78	-0.0170	-7.66	-0.00122	-7.79	-0.0177	-8.45	-0.105
9	-4.86	-4.44	0.0862	-4.55	0.0637	-4.45	0.0827	-5.96	-0.227
10	-11.5	-11.3	0.0140	-11.4	0.00879	-11.4	0.0127	-11.6	-0.0116

Table 6: Interpolation of f in 10 points, case of a 4D function.

For RBF methods, a Gaussian radial basis function with a scale factor of 0.5 was used.

The relative error ϵ is calculated as follows: $\epsilon = \dfrac{o_{\text{interpolated}} - o_{\text{real}}}{|o_{\text{real}}|}$

Then, the relative error is plotted for each interpolated point to compare the different interpolation methods. The OLS method is not shown on the graph because it is not suitable for this situation. The values are still present in the table, but it is clear that the RBF methods perform better.

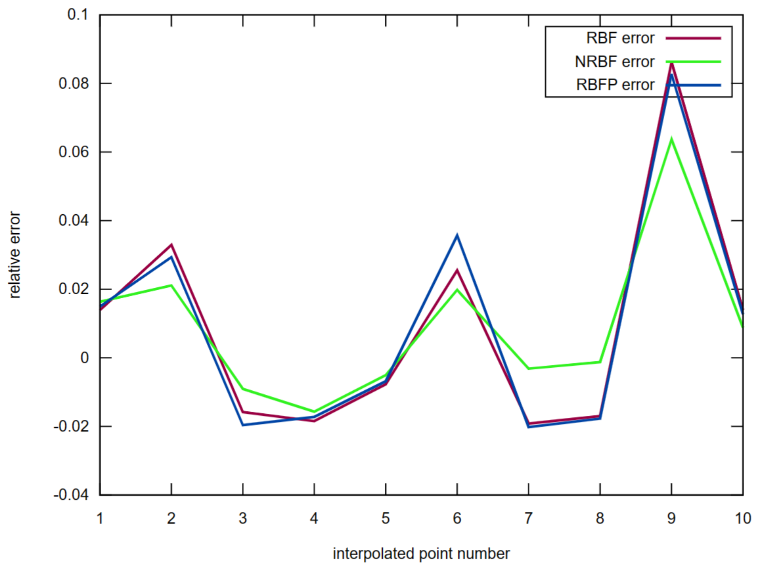

Figure 9: Error in f interpolation, a 4D function.

RBF methods seem more suitable than OLS, which is to be expected given that OLS can only be used in certain cases (linear function, which is not the case here). The method with the smallest errors in this case is the NRBF one. However, it's important to bear in mind that the matrices are assembled randomly, so the conclusions about which RBF method is the most efficient may change from one run to the next.

6.5 Case 4: study of the rescaling

In this test case, the usefulness of rescaling is examined. To do this, the same f function as above is considered (8). The aim is to compare the impact of rescaling on the relative error of interpolated points for each RBF method (RBF, NRBF and RBFP). The simulation parameters are the same: 4 variables and 150 known points. Then 10 points will be interpolated and compared with their real value using the f function.

To see the impact of rescaling, the parameters will be made slightly more anisotropic. Instead of having them only between 0 and 1, x_1 will be between -1 and 1, x_2 between 10 and 20, x_3 between 1 and 5, and x_4 between 0 and 0.5. The same radial basis function (Gaussian) and the same scale factor (0.5) will be used. Before processing the results, it should be borne in mind that since the parameters are no longer defined in the same way, it would have been appropriate to re-study which radial basis function and which scale factor would be most appropriate. In this case, the important thing is not the accuracy of the interpolation method as such, but its accuracy compared to the use or non-use of rescaling methods. The error values will naturally be higher because of the anisotropy and non-reparametrization of the radial basis function.

For each RBF method, the relative error is plotted with and without the use of the rescaling class.

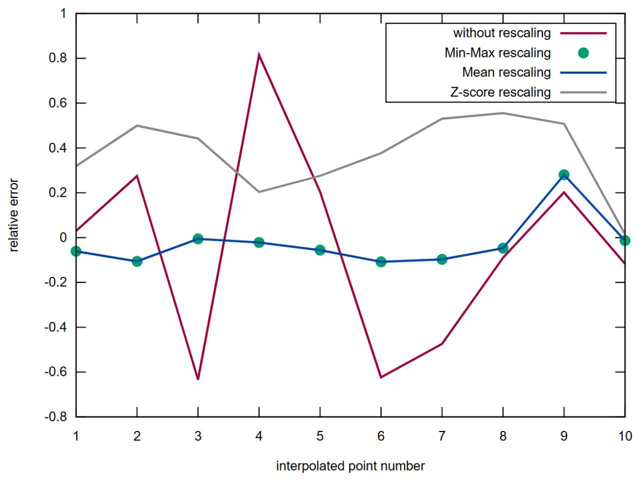

Figure 10: Error in f interpolation using RBF, a 4D function.

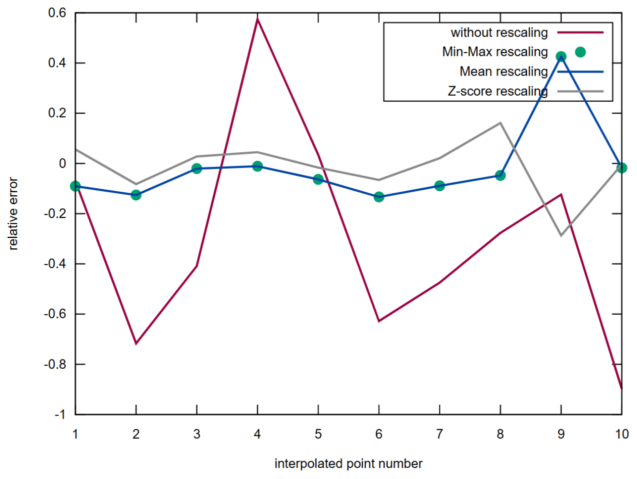

Figure 11: Error in f interpolation using NRBF, a 4D function.

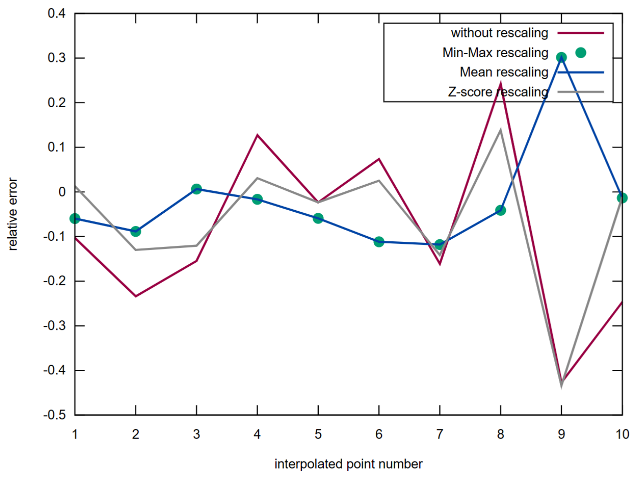

Figure 12: Error in f interpolation using RBFP, a 4D function.

The results are also shown in the tables below. The *no* index refers to interpolation without rescaling, the *min-max* index to min-max normalization, *mean* to mean normalization and *z-score* to z-score standardization. The index is applied to the $\mathbf{x}$ vector to show that rescaling takes place before the interpolation phase.

n°	$f(\mathbf{x})$	$f(\mathbf{x}_{\text{no}})$	ϵ_{no}	$f(\mathbf{x}_{\text{min-max}})$	$\epsilon_{\text{min-max}}$	$f(\mathbf{x}_{\text{mean}})$	ϵ_{mean}	$f(\mathbf{x}_{\text{z-score}})$	$\epsilon_{\text{z-score}}$
1	-40.7	-39.5	0.0297	-43.2	-0.0610	-43.2	-0.0610	-27.7	0.319
2	-25.3	-18.3	0.275	-28.0	-0.106	-28.0	-0.106	-12.6	0.500
3	-26.8	-43.8	-0.634	-27.0	-0.00589	-27.0	-0.00589	-15.0	0.442
4	-55.7	-10.3	0.815	-56.9	-0.0218	-56.9	-0.0218	-44.3	0.204
5	-26.3	-20.9	0.205	-27.8	-0.0560	-27.8	-0.0560	-19.0	0.276
6	-20.5	-33.4	-0.624	-22.8	-0.108	-22.8	-0.108	-12.8	0.377
7	-24.5	-36.1	-0.474	-26.9	-0.0971	-26.9	-0.0971	-11.5	0.530
8	-23.6	-25.7	-0.0897	-24.7	-0.0473	-24.7	-0.0473	-10.5	0.556
9	-13.5	-10.8	0.202	-9.75	0.280	-9.75	0.280	-6.67	0.508
10	-43.7	-48.8	-0.117	-44.3	-0.0132	-44.3	-0.0132	-43.1	0.0142

Table 7: RBF interpolation of f in 10 points with different rescaling, case of a 4D function.

n°	$f(\mathbf{x})$	$f(\mathbf{x}_{\text{no}})$	ϵ_{no}	$f(\mathbf{x}_{\text{min-max}})$	$\epsilon_{\text{min-max}}$	$f(\mathbf{x}_{\text{mean}})$	ϵ_{mean}	$f(\mathbf{x}_{\text{z-score}})$	$\epsilon_{\text{z-score}}$
1	-40.7	-43.5	-0.0691	-44.4	-0.0905	-44.4	-0.0905	-38.4	0.0562
2	-25.3	-43.4	-0.717	-28.4	-0.126	-28.4	-0.126	-27.3	-0.0823
3	-26.8	-37.8	-0.409	-27.4	-0.0204	-27.4	-0.0204	-26.1	0.0277
4	-55.7	-23.7	0.574	-56.3	-0.0112	-56.3	-0.0112	-53.2	0.0448
5	-26.3	-25.4	0.0344	-27.9	-0.0634	-27.9	-0.0634	-26.7	-0.0165
6	-20.5	-33.4	-0.628	-23.3	-0.133	-23.3	-0.133	-21.9	-0.0659
7	-24.5	-36.2	-0.475	-26.7	-0.0891	-26.7	-0.0891	-24.0	0.0210
8	-23.6	-30.1	-0.277	-24.7	-0.0478	-24.7	-0.0478	-19.8	0.161
9	-13.5	-15.2	-0.124	-7.78	0.426	-7.78	0.426	-17.4	-0.287
10	-43.7	-82.9	-0.898	-44.5	-0.0178	-44.5	-0.0178	-43.9	-0.00385

Table 8: NRBF interpolation of f in 10 points with different rescaling, case of a 4D function.

n°	$f(\mathbf{x})$	$f(\mathbf{x}_{\text{no}})$	ϵ_{no}	$f(\mathbf{x}_{\text{min-max}})$	$\epsilon_{\text{min-max}}$	$f(\mathbf{x}_{\text{mean}})$	ϵ_{mean}	$f(\mathbf{x}_{\text{z-score}})$	$\epsilon_{\text{z-score}}$
1	-40.7	-44.9	-0.103	-43.2	-0.0597	-43.2	-0.0597	-40.2	0.0129
2	-25.3	-31.2	-0.234	-27.5	-0.0886	-27.5	-0.0886	-28.6	-0.130
3	-26.8	-31.0	-0.155	-26.6	0.00639	-26.6	0.00639	-30.1	-0.121
4	-55.7	-48.6	0.127	-56.6	-0.0166	-56.6	-0.0166	-54.0	0.0308
5	-26.3	-26.9	-0.0234	-27.8	-0.0592	-27.8	-0.0592	-26.9	-0.0234
6	-20.5	-19.0	0.0736	-22.8	-0.112	-22.8	-0.112	-20.0	0.0252
7	-24.5	-28.5	-0.161	-27.4	-0.118	-27.4	-0.118	-28.0	-0.141
8	-23.6	-17.9	0.242	-24.6	-0.0412	-24.6	-0.0412	-20.3	0.138
9	-13.5	-19.3	-0.427	-9.46	0.301	-9.46	0.301	-19.4	-0.434
10	-43.7	-54.5	-0.247	-44.3	-0.0135	-44.3	-0.0135	-44.2	-0.0115

Table 9: RBFP interpolation of f in 10 points with different rescaling, case of a 4D function.

The main point to remember is that rescaling works more or less well depending on the method and the point to be interpolated, but that the effectiveness of rescaling is mainly due to the choice of function and its suitability for the function to be interpolated. for instance, if the user considers a large number of independent variables following a normal distribution then if the function to be interpolated is the sum of these variables, according to the central limit theorem the sum of these variables should follow a normal distribution. It can therefore be interesting to use a Gaussian radial basis function and to use a Z-score rescaling method which is based on the standardisation of a normal distribution. But in this case of study, the function is much more complex than that and, likely, rescaling methods such as z-score standardization are not the most appropriate. However, to compensate for the specificity of the rescaling method depending on the function to be interpolated, the library has been designed to create a separate class that is independent of the interpolation methods. In this way, users can rescale their data themselves using their tools, or even add this method to the Rescaling class.

6.6 Case 5: study of the convergence of the interpolation error

The objective here is to see how the interpolation error of RBF methods evolves as a function of the number of known points. To do this, it is interesting to plot the evolution of the interpolation error as a function of the number of known points for different dimensions/numbers of variables. We, therefore, need to choose a function f that can be generalized to n dimensions and have parameter values that are as isotropic as possible, in order to be in the best possible situation for using RBF methods. The f function chosen is as follows:

$$f(\mathbf{x}_{nD}) = f(x_1, \cdots, x_n) = \sum_{i=1}^{n} x_i \tag{9}$$

This function was chosen because it makes it easy to determine which radial base function to use. It also avoids introducing random numbers for the coefficients, which could have an impact on the estimation error. In addition, in order not to distort this study, the parameter values will be preserved from one dimension to another, i.e. the parameter matrix P for a dimension of n will be reused for the dimension n+1, all that remains is to add a new column and so on. The same thing will be done for known numbers of points. This saves an enormous amount of calculation time, avoiding the need to resample complete matrices, and means that the same contribution can be observed from each parameter from one dimension to another.

Then the values of dimensions n chosen for this study were: 1,2,4,6,8,10. The results are shown in the graph below:

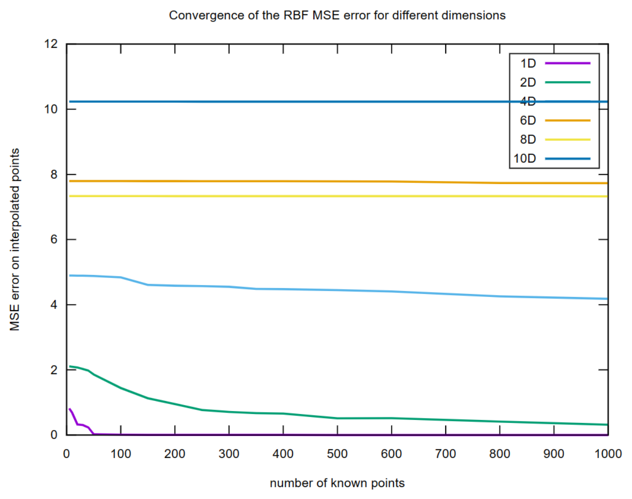

Figure 13: Mean Square Error on the RBF interpolated points.

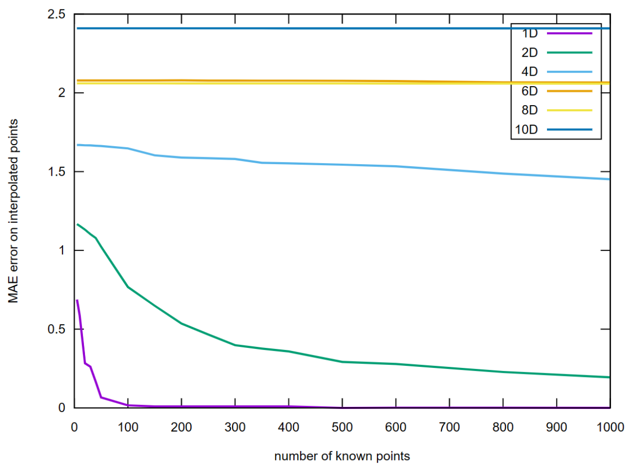

Figure 14: Mean Absolute Error on the RBF interpolated points.

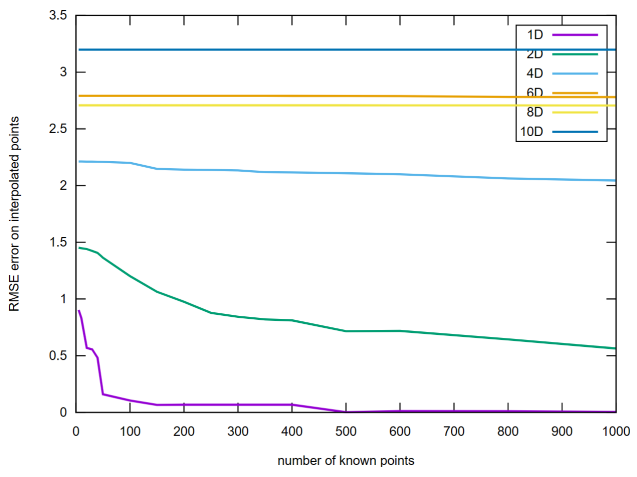

Figure 15: Root Mean Square Error on the RBF interpolated points.

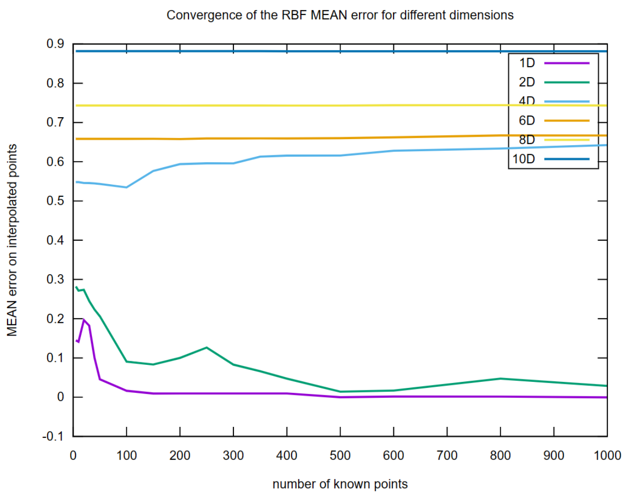

Figure 16: Mean Error on the RBF interpolated points.

The various error estimators plotted for dimensions 1 and 2 therefore converge towards 0, but for higher dimensions 1000 known points do not seem sufficient. We therefore need to push the simulations a little further if we want to observe similar results for dimensions 1 and 2. On the other hand, it was noted that very quickly, for dimensions greater than 4, the volume of data required seems very large.

6.7 Case 6: some test in Python

The purpose of this test case is to test each of the functions offered by the library in Python in order to check the functionality of the bindings. To do this, each class was tested and the graph 8 reproduced.

7 Bidings Python/C++

To create the bindings linking the C++ library to its use in Python, the pybind11 library was used for its simplicity, robustness and compatibility with the Eigen library.

A *bindings.cpp* file was added to the project and contains all the bindings needed to run mdisd in Python.

```cpp
#include <pybind11/pybind11.h>
#include <pybind11/eigen.h>
#include <pybind11/stl.h>

#include "interpolator.hpp"
#include "OLSinterpolator.hpp"
#include "RBFinterpolator.hpp"
#include "RBFunctions.hpp"
#include "rescaling.hpp"

namespace py = pybind11;

template <typename T>
py::tuple interpolate_with_regression(T& self, const Eigen::MatrixXd&
    points_to_interpolate, const Eigen::MatrixXd& known_parameters, const
    Eigen::VectorXd& known_measurements) {
    Eigen::VectorXd regression;
    Eigen::VectorXd results = self.interpolate(points_to_interpolate, known_parameters,
        known_measurements, &regression);
    return py::make_tuple(results, regression);
}

PYBIND11_MODULE(mdisd_py, m) {

    py::class_<OLSInterpolator>(m, "OLSInterpolator")
        .def(py::init<>())
        .def("interpolate", static_cast<Eigen::VectorXd (OLSInterpolator::*)(const
            Eigen::MatrixXd&, const Eigen::MatrixXd&, const Eigen::VectorXd&)
            const>(&OLSInterpolator::interpolate), py::arg("points_to_interpolate"),
            py::arg("known_parameters"), py::arg("known_measurements"))
        .def("interpolate_with_regression",
            &interpolate_with_regression<OLSInterpolator>);

    py::class_<RBFInterpolator>(m, "RBFInterpolator")
        .def(py::init<std::function<double(double, double)>, double, bool, bool>(),
            py::arg("rbf_function"), py::arg("scale_factor"), py::arg("flag1"),
            py::arg("flag2"))
        .def("interpolate", static_cast<Eigen::VectorXd (RBFInterpolator::*)(const
            Eigen::MatrixXd&, const Eigen::MatrixXd&, const Eigen::VectorXd&)
            const>(&RBFInterpolator::interpolate), py::arg("points_to_interpolate"),
            py::arg("known_parameters"), py::arg("known_measurements"))
        .def("interpolate_with_regression",
            &interpolate_with_regression<RBFInterpolator>);


    py::class_<std::function<double(double, double)>>(m, "FunctionDD");

    py::module m_rbfunction = m.def_submodule("rbfunction", "RBFunctions submodule");

    auto multiquadratic_wrapper = []() -> std::function<double(double, double)> {
        return [](double r, double r0) -> double {
            return RBFunctions::multiquadratic(r, r0);
        };
    };
    m_rbfunction.def("Multiquadratic", multiquadratic_wrapper, "Multiquadratic radial
        basis function");

    m_rbfunction.def("call_Multiquadratic", [](double r, double r0) {
        return RBFunctions::multiquadratic(r, r0);
    }, "Call Multiquadratic radial basis function");


    auto invmultiquadratic_wrapper = []() -> std::function<double(double, double)> {
        return [](double r, double r0) -> double {
            return RBFunctions::inverseMultiquadratic(r, r0);
        };
    };
    m_rbfunction.def("invMultiquadratic", invmultiquadratic_wrapper, "Inverse
        multiquadratic radial basis function");

    m_rbfunction.def("call_invMultiquadratic", [](double r, double r0) {
        return RBFunctions::inverseMultiquadratic(r, r0);
    }, "Call Inverse multiquadratic radial basis function");


    auto gaussian_wrapper = []() -> std::function<double(double, double)> {
        return [](double r, double r0) -> double {
            return RBFunctions::gaussian(r, r0);
        };
    };
    m_rbfunction.def("Gaussian", gaussian_wrapper, "Gaussian radial basis function");

    m_rbfunction.def("call_Gaussian", [](double r, double r0) {
        return RBFunctions::gaussian(r, r0);
    }, "Call Gaussian radial basis function");


    auto thinPlateSpline_wrapper = []() -> std::function<double(double, double)> {
        return [](double r, double r0) -> double {
            return RBFunctions::thinPlateSpline(r, r0);
        };
    };
    m_rbfunction.def("ThinPlateSpline", thinPlateSpline_wrapper, "Thin Plate Spline
        radial basis function");

    m_rbfunction.def("call_ThinPlateSpline", [](double r, double r0) {
        return RBFunctions::thinPlateSpline(r, r0);
    }, "Call Thin Plate Spline radial basis function");


    py::class_<Rescaling>(m, "Rescaling")
        .def(py::init<>())
        .def("Mean", &Rescaling::meanNormalization, py::arg("data1"), py::arg("data2")
            = nullptr)
        .def("MinMax", &Rescaling::minMaxNormalization, py::arg("data1"),
            py::arg("data2") = nullptr)
        .def("Zscore", &Rescaling::zScoreNormalization, py::arg("data1"),
            py::arg("data2") = nullptr);
}
```

Listing 6: Bindings.

This was one of the trickiest stages of the project because it's quite complex to link (especially for the first time) C++ logic with objects to Python logic. The simplest parts were the *OLSInterpolator* and *Rescaling* classes, which were fairly straightforward to transcribe, although the *interpolator* override caused a few problems. For this, the adaptation chosen was to create a second function *interpolate_with_coefficients* for which returns the result and the coefficients. Indeed, it was not possible (with my knowledge at this stage) to keep a single function usable in Python that may or may not take an additional vector to store the coefficients.

The same thing was used for the RBF method, but another problem arose with the *RBFunctions* when they were called in Python. A type problem arose and the solution found was to define the *RBFunctions* as function wrappers. But this prevents the user from using the *RBFunctions* on their own, without calling an interpolation method with them. So, after a lot of research, the solution found was simply to double the functions and create functions that can be called by the user and functions that can be called by *RBFInterpolator*.

Of course, the solutions found are not the most optimal, and no doubt to facilitate bindings we would have to completely rethink the library. Nevertheless, for the moment it works like this, but the process can seem rather heavy.

8 How to ...

8.1 download and use the library for C++ applications

8.1.1 clone the library

To use the library, first, you will need to clone the GitHub repository of the library.

```
# To clone a specific version of the library.
git clone --branch vX.Y.Z https://github.com/MalmbergNilsPolimi/mdisd

# To clone the last version of the library.
git clone https://github.com/MalmbergNilsPolimi/mdisd
```

If you just want to run the test cases that are already implemented in the library, you can just the following commands :

```
cd mdisd

# Replace X by the number of the test case you want to run. X is from 0 to 5.
cd test/caseX

make
make run
```

But before using *make* and *make run* commands you need to check if you have on your computer the Eigen library and modify the path to the library. So in each *Makefile* you will find this to indicate where to change the path to the Eigen library:

```
####################################################################################
############### THE USER NEED TO CHANGE HERE THE PATH TO EIGEN LIBRARY ###############
####################################################################################
INCLUDE_DIR_EIGEN := /u/sw/toolchains/gcc-glibc/11.2.0/pkgs/eigen/3.3.9/include/eigen3
####################################################################################
```

After using the test case, you can clean all files and figures using the command :

```
make clean
```

Now, if you want to use the library in one of your projects you can follow these instructions:

```
cd mdisd
mkdir build && cd build
cmake ..
make
```

Now if you use the command *ls* you will see that *.so* files were created. They are the files of the mdisd shared library. So to use it in a project you can use the command *pwd*, copy the path, and then use the command *-Lpath -lmdisd* during the compilation. You will also need to indicate where to find the header files. They are in the directory *mdisd/include* and its subdirectories.

8.1.2 use the library

To see, the different uses of the library functions you can refer to the different test cases. But here is a brief recap on how to use it.

```cpp
#include <Eigen/Dense>
#include "RBFunctions.hpp"
#include "RBFinterpolator.hpp"
#include "OLSinterpolator.hpp"
#include "rescaling.hpp"


// Before using interpolation methods, you need to define different matrices.

// Contain sets of known points.
Eigen::MatrixXd parameters(number of known points, number of variables);

// Contain values associated to the known points.
Eigen::VectorXd measurements(number of known points);

// Contain the points where we want an interpolated value.
Eigen::MatrixXd parametersFORinterp(number of points to interpolate, number of
    variables);


// If you want to rescale your data.
Rescaling rescaling = minMaxNormalization(parameters, &parametersFORinterp);


// To use OLS interpolation.
OLSInterpolator interpolatorOLS;
// If you use the rescaling replace parametersFORinterp by rescaling.second and
    parameters by rescaling.first
Eigen::VectorXd OLS_points_interpolated =
    interpolatorOLS.interpolate(parametersFORinterp, parameters, measurements);

// To use RBF interpolation.
double scale_factor{0.5};
RBFInterpolator interpolatorRBF(&RBFunctions::gaussian, scale_factor);
Eigen::VectorXd RBF_points_interpolated =
    interpolatorRBF.interpolate(parametersFORinterp, parameters, measurements);

// To use NRBF interpolation.
bool normalize{true};
RBFInterpolator interpolatorNRBF(&RBFunctions::gaussian, scale_factor, normalize);
Eigen::VectorXd NRBF_points_interpolated =
    interpolatorNRBF.interpolate(parametersFORinterp, parameters, measurements);

// To use RBFP interpolation.
bool polynomial{true};
normalize=false;
RBFInterpolator interpolatorNRBF(&RBFunctions::gaussian, scale_factor, normalize,
    polynomial);
Eigen::VectorXd NRBF_points_interpolated =
    interpolatorNRBF.interpolate(parametersFORinterp, parameters, measurements);
```

Listing 7: How to use mdisd functions.

8.2 download and use the library for Python applications

8.2.1 clone the library

To use the library in a Python project, first, you need to clone the project as in section 8.1.1. You will also need the pybind11 library that you can clone using directly in the mdisd directory:

```
git clone https://github.com/pybind/pybind11.git
```

Then before using the *cmake* command as in 8.1.1, you will need to update different paths in *CMakeLists.txt* (at the root of the library, in mdsid) paths to Eigen and pybind11.

Then you need to update the path to Python. For that, first, you will to find which version of Python you have and where is it located. In the shell write:

```
python3

# Now you are in the python interpreter if python3 is installed
import sys
sys.prefix
```

Copy the path printed in the python interpreter. Then go back to a shell and write:

```
cd path_from_python_interpreter
ls

# X is the version of python that you have, for me it was python3.9
cd python3.X.
pwd
```

Then copy this new path. Go back in the *CMakeLists.txt* file and update the path in:

```
include_directories(/u/sw/toolchains/gcc-glibc/11.2.0/base/include/python3.9)
```

Then by using *cmake .. && make* in the build folder (as previously), you will create the file *mdisd_py.cpython-39-x86_64-linux-gnu.so* that you can use in your Python projects to link the library.

8.2.2 use the library

To use the library, you just need to add the path to the *.so* files. An example is given in the test case number 6. But how to use the different functions of the library. For that, you can refer directly to the test case 6 which contains examples of uses.

```
# Add the path to the .so file to your project.
import sys
sys.path.append('../../build')

import mdisd_py.rbfunction as rbfunction
import mdisd_py

import numpy as np
import matplotlib.pyplot as plt

#Test of the different radial basis functions.
```

```python
print("Multiquadratique(2.5 , 1.0) = ", rbfunction.call_Multiquadratic(2.5, 1.0))
print("invMultiquadratique(2.5 , 1.0) = ", rbfunction.call_invMultiquadratic(2.5, 1.0))
print("Gaussian(2.5, 1.0) = ", rbfunction.call_Gaussian(2.5, 1.0))
print("ThinPlateSpline(2.5, 1.0) = ", rbfunction.call_ThinPlateSpline(2.5, 1.0))

# Test of the different rescaling methods.
data1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
data2 = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])

mean_normalized_data1, mean_normalized_data2 = mdisd_py.Rescaling().Mean(data1, data2)
print("Mean normalized data1:\n", mean_normalized_data1)
print("Mean normalized data2:\n", mean_normalized_data2)

minmax_normalized_data1, minmax_normalized_data2 = mdisd_py.Rescaling().MinMax(data1,
    data2)
print("minmax normalized data1:\n", minmax_normalized_data1)
print("minmax normalized data2:\n", minmax_normalized_data2)

zscore_normalized_data1, zscore_normalized_data2 = mdisd_py.Rescaling().Zscore(data1,
    data2)
print("zscore normalized data1:\n", zscore_normalized_data1)
print("zscore normalized data2:\n", zscore_normalized_data2)


# Test of interpolation.

def f(x):
    return 0.5 * x - 4.3

known_parameters = np.array([[-2], [3.7], [0.1], [-6], [18.2]])
known_measurements = f(known_parameters)

points_to_interpolate = []
inf , sup = -10 , 20
num_points = 10

for i in range(num_points):
    points_to_interpolate.append(inf + i * (sup - inf) / (num_points - 1))

# OLS interpolation.
ols_interpolator = mdisd_py.OLSInterpolator()

ols_results = ols_interpolator.interpolate(points_to_interpolate, known_parameters,
    known_measurements)
ols_results2, ols_regression =
    ols_interpolator.interpolate_with_coefficients(points_to_interpolate,
    known_parameters, known_measurements)

print("OLS weights:\n", ols_regression)

# RBF interpolation.
rbf_interpolator = mdisd_py.RBFInterpolator(rbfunction.Multiquadratic(), 0, False,
    False)

rbf_results = rbf_interpolator.interpolate(points_to_interpolate, known_parameters,
    known_measurements)
rbf_results2, rbf_regression =
    rbf_interpolator.interpolate_with_coefficients(points_to_interpolate,
    known_parameters, known_measurements)
```

```python


print("RBF weights:\n", rbf_regression)

# NRBF interpolation.
nrbf_interpolator = mdisd_py.RBFInterpolator(rbfunction.Multiquadratic(), 0, True,
    False)

nrbf_results = nrbf_interpolator.interpolate(points_to_interpolate, known_parameters,
    known_measurements)
nrbf_results2, nrbf_regression =
    nrbf_interpolator.interpolate_with_coefficients(points_to_interpolate,
    known_parameters, known_measurements)

print("NRBF weights:\n", nrbf_regression)

# RBFP interpolation.
rbfp_interpolator = mdisd_py.RBFInterpolator(rbfunction.Multiquadratic(), 0, False,
    True)

rbfp_results = rbfp_interpolator.interpolate(points_to_interpolate, known_parameters,
    known_measurements)
rbfp_results2, rbfp_regression =
    rbfp_interpolator.interpolate_with_coefficients(points_to_interpolate,
    known_parameters, known_measurements)

print("RBFP weights:\n", rbfp_regression)

plt.scatter(known_parameters, known_measurements, label="regressors", color='green')
plt.plot(points_to_interpolate, ols_results, label="OLS interpolation",
    linestyle='dashed', color='red')
plt.plot(points_to_interpolate, rbf_results, label="RBF interpolation", color='blue')
plt.plot(points_to_interpolate, nrbf_results, label="NRBF interpolation",
    color='green')
plt.scatter(points_to_interpolate, rbfp_results, label="RBFP interpolation",
    color='orange')

plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend()
plt.savefig('plot_test_python.png')
```

Listing 8: Test case 6.

Conclusion

The mdisd library is a versatile tool, leveraging on the Eigen library, for the multi-dimensional interpolation of scattered data, implementing both radial basis functions (RBF) and ordinary least squares (OLS) methods.

This project has explored the theoretical foundations of interpolation and the principles behind RBF and OLS methods, as well as their advantages and limitations. The mdisd library's code architecture has been designed with a focus on flexibility, ease of use, and performance.

The performance of the RBF and OLS interpolators has been evaluated, and recommendations have been provided for their optimal use in specific scenarios. The integration of pybind11 bindings allows the mdisd library to be used in Python, broadening its accessibility and applicability.

In summary, the mdisd library offers a comprehensive and practical implementation of RBF and OLS interpolation methods in a C++ programming environment.

References

[1] J. Faraway. "Practical Regression and Anova using R". In: (2002). URL: https://cran.r-project.org/doc/contrib/Faraway-PRA.pdf.

[2] C. De Chaisemartin. "Ordinary Least Squares: the multivariate case". Lecture, Paris School of Economics. 2011. URL: https://www.parisschoolofeconomics.eu/docs/de-chaisemartin-clement/ols_multivariate_2011.pdf.

[3] G. Strang. "Chapter 7 - The Singular Value Decomposition (SVD)". Lecture - MIT Mathematics. 2016. URL: https://math.mit.edu/classes/18.095/2016IAP/lec2/SVD_Notes.pdf.

[4] William H. Press et al. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd ed. USA: Cambridge University Press, 2007. ISBN: 0521880688.

[5] Wilna Du Toit. "Radial basis function interpolation". In: 2008. URL: https://api.semanticscholar.org/CorpusID:123907500.

[6] Vaclav Skala. "Radial basis functions interpolation and applications: an incremental approach". In: *Proceedings of the 4th International Conference on Applied Mathematics, Simulation, Modelling*. ASM'10. Corfu Island, Greece: World Scientific, Engineering Academy, and Society (WSEAS), 2010, pp. 209–213. ISBN: 9789604742103.

[7] S.Gopal, Krishna Patro, and Kishore Kumar Sahu. "Normalization: A Preprocessing Stage". In: *ArXiv* abs/1503.06462 (2015). URL: https://api.semanticscholar.org/CorpusID:16159835.

[8] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[9] Dimas Anggoro and Wiwit Supriyanti. "Improving Accuracy by applying Z-Score Normalization in Linear Regression and Polynomial Regression Model for Real Estate Data". In: *International Journal of Emerging Trends & Technology in Computer Science* (Nov. 2019), pp. 549–555. DOI: 10.30534/ijeter/2019/247112019.

[10] Maria De Marsico and Daniel Riccio. "A New Data Normalization Function for Multibiometric Contexts: A Case Study". In: June 2008, pp. 1033–1040. ISBN: 978-3-540-69811-1. DOI: 10.1007/978-3-540-69812-8_103.

[11] Qiuyu Zhu et al. "Improving Classification Performance of Softmax Loss Function Based on Scalable Batch-Normalization". In: *Applied Sciences* 10 (Apr. 2020), p. 2950. DOI: 10.3390/app10082950.