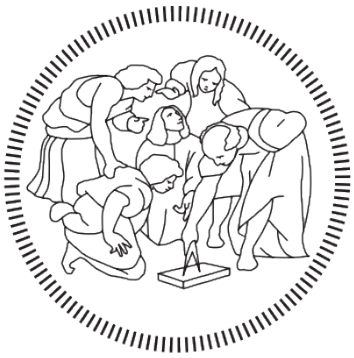


POLITECNICO
MILANO 1863

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.

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 own 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 explained, 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.

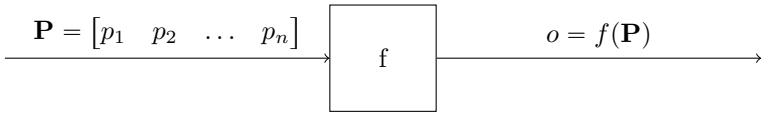
Finally, the performance of the interpolators is evaluated by comparing them on data sets. The obtained results are analyzed, and the advantages and limitations of each method in different contexts are discussed. Recommendations are made for optimal use of these methods.

Additionally, bindings 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.

1 Notations

Let's consider a system that depends directly on n parameters and which 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 $\mathbf{P}$.



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 $\mathbf{P}_{\text{new}} = \begin{pmatrix} p_{\text{new},1} & p_{\text{new},2} & \dots & p_{\text{new,n}} \end{pmatrix}$.

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 squares of the differences between the observed and predicted values. In essence, 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,

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

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

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

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

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:

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

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:

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

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.

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 desirable 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]: $T(p_{i,j}) = \dfrac{p_{i,j} - \left[\dfrac{1}{m}\sum\limits_{l=1}^{m} p_{l,j}\right]}{\max(p_{1,j}, \cdots, p_{m,j}) - \min(p_{1,j}, \cdots, p_{m,j})}$;
- Z-score normalization [9]: $T(p_{i,j}) = \dfrac{p_{i,j} - \left[\dfrac{1}{m}\sum\limits_{l=1}^{m} p_{l,j}\right]}{\sqrt{\dfrac{1}{m}\sum\limits_{l=1}^{m} p_{l,j}^2 - \left[\dfrac{1}{m}\sum\limits_{l=1}^{m} p_{l,j}\right]^2}}$.

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: $T(p_{i,j}) = \dfrac{1}{2}\left[1 + tan^{-1}(p_{i,j})\right]$;
- 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

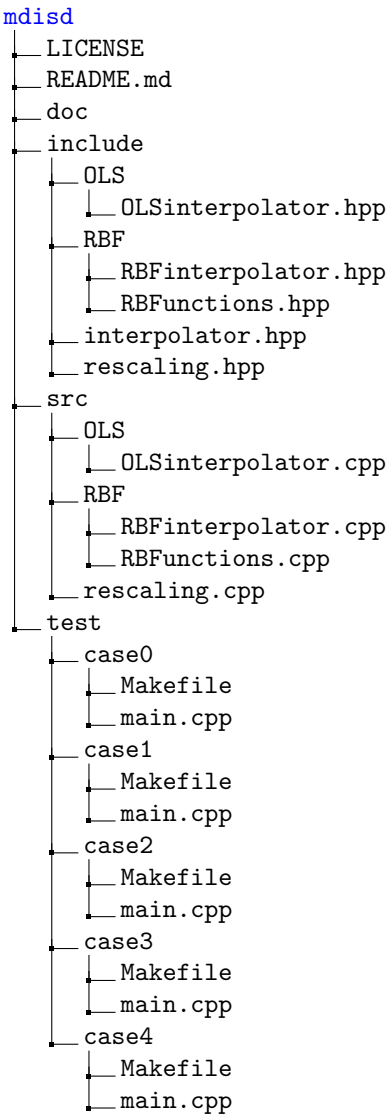


Figure 1: Tree of the mdisd library.

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.

Listing 1: *Interpolator* class.

```
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;
};
```

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.

Listing 2: *RBFInterpolator* class.

```
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;
};
```

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.

Listing 3: *RBFunctions* class.

```
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);
};
```

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.

Listing 4: *OLSInterpolator* 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;
};
```

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.

Listing 5: *Rescaling* class.

```
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::MatrixXd combineData(const Eigen::MatrixXd& data1,
                                const Eigen::MatrixXd* data2 = nullptr);

    Eigen::VectorXd computeColumnMeans(const Eigen::MatrixXd& data1,
                                       const Eigen::MatrixXd* data2 = nullptr);

    Eigen::VectorXd computeColumnStdDevs(const Eigen::MatrixXd& data1,
                                         const Eigen::VectorXd& means,
                                         const Eigen::MatrixXd* data2 = nullptr);

    Eigen::VectorXd computeColumnMin(const Eigen::MatrixXd& data1,
                                     const Eigen::MatrixXd* data2 = nullptr);

    Eigen::VectorXd computeColumnMax(const Eigen::MatrixXd& data1,
                                     const Eigen::MatrixXd* data2 = nullptr);
};
```

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 6.

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   */
num_params ← number of columns of parameters;
num_measures ← size of measurements;
num_points ← number of rows of parametersFORinterp;

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

/* Calculate the transpose of X                                                        */
transpose_X ← X^T;

/* Decompose the matrix X^T X using SVD                                                */
U, Σ, V* ← SVD(X^T X);

/* Invert the matrix X^T X                                                             */
(X^T X)^-1 ← V*Σ^-1 U^T;

/* Calculate the β coefficients                                                        */
β ← (X^T X)^-1 X^T O;

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

/* Calculate the interpolated values                                                   */
for i from 0 to num_points − 1 do
    for j from 0 to num_params − 1 do
        results(i) ← β(0) + β(j + 1) × parametersFORinterp(i,j);
    end
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   */
num_params ← number of columns of parameters;
num_measures ← size of measurements;
num_points ← number of rows of parametersFORinterp;

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

/* Compute the weights                                                                 */
Initialize coeff as a matrix of zeros with appropriate dimensions;
for i from 0 to num_measures − 1 do
    for j from 0 to num_measures − 1 do
        coeff(i, j)← φ(||parameters(i) − parameters(j)||, r_0);
    end
    if polynomialRBF is true then
        for k from num_measures to num_measures + num_params − 1 do
            coeff(i, k) ← parameters(i, k-num_measures);
            coeff(k, i) ← parameters.tanspose(k-num_measures, i);
        end
        coeff(i, num_measures+num_params) ← 1;
        coeff(num_measures+num_params, i) ← 1;
    end
end

/* Solve the system using SVD to find the weights                                      */
if normalizeRBF then
    for i from 0 to num_measures − 1 do
        Initialize sum1 as 0;
        for j from 0 to num_measures − 1 do
            sum1 ← sum1 + φ(||parameters(i) − parameters(j)||, r_0);
        end
        NEWmeasurements(i) ← measurements(i) × sum1;
    end
    weights ← SVD(NEWmeasurements);
end
else if polynomialRBF then
    for i from 0 to num_measures − 1 do
        NEWmeasurements(i) ← measurements(i);
    end
    for i from num_measures to num_measures + num_params do
        NEWmeasurements(i) ← 0;
    end
    weights ← SVD(NEWmeasurements);
end
else
    weights ← SVD(measurements);
end
```

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

   /* Compute the interpolated values                                          */
44 for k from 0 to num_points − 1 do
45 |  if normalizeRBF is true then
46 |  |  normalize_part(k) ← 0;
47 |  |  for l from 0 to num_measures − 1 do
48 |  |  |  normalize_part(k) ← normalize_part(k) + φ(||parametersFORinterp(k) − parameters(l)||, r0);
49 |  |  end
50 |  end
51 |  for l from 0 to num_measures − 1 do
52 |  |  results(k) ← weights(l)×φ(||parametersFORinterp(k) − parameters(l)||, r0)/normalize_part(k);
53 |  end
54 |  if polynomialRBF is true then
55 |  |  for i from num_measures to num_measures + num_params − 1 do
56 |  |  |  results(k) ← results(k)+weights(i)×parametersFORinterp(k, i-num_measures);
57 |  |  end
58 |  |  results(k) ← results(k)+weights(num_measures+num_params);
59 |  end
60 end
```

6 How to ...

6.1 use the library

6.2 upgrade the library

7 Test of the library

7.1 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:

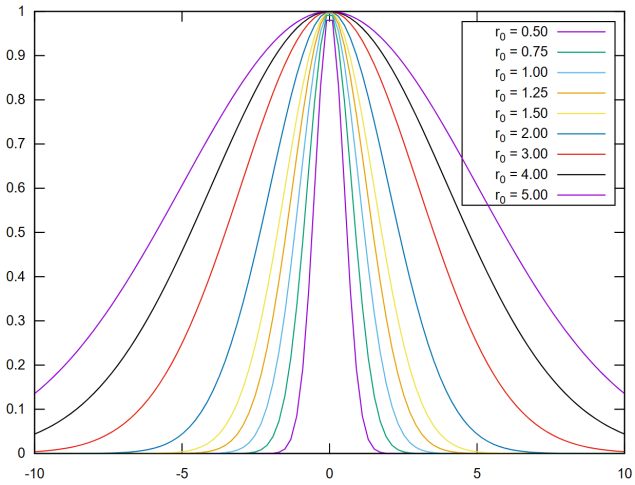


Figure 2: Gaussian basis function.

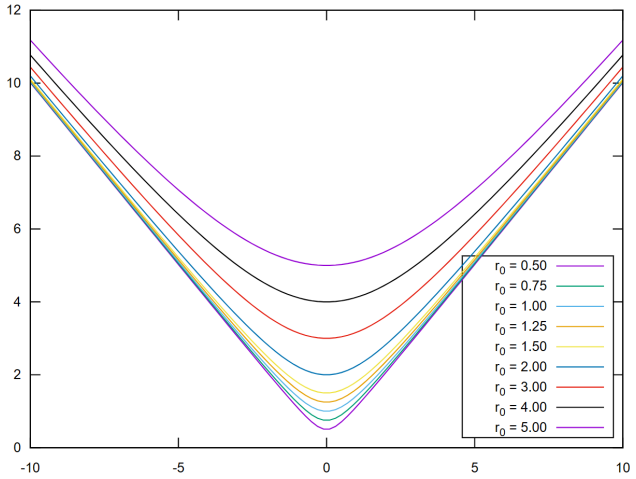


Figure 3: Multiquadratic basis function.

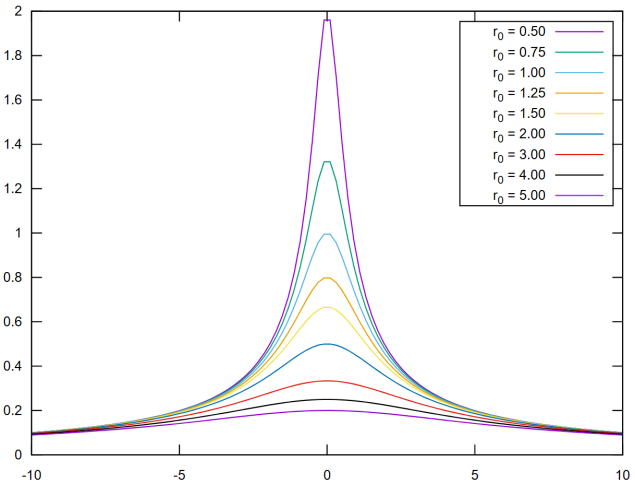


Figure 4: Inverse multiquadratic basis function.

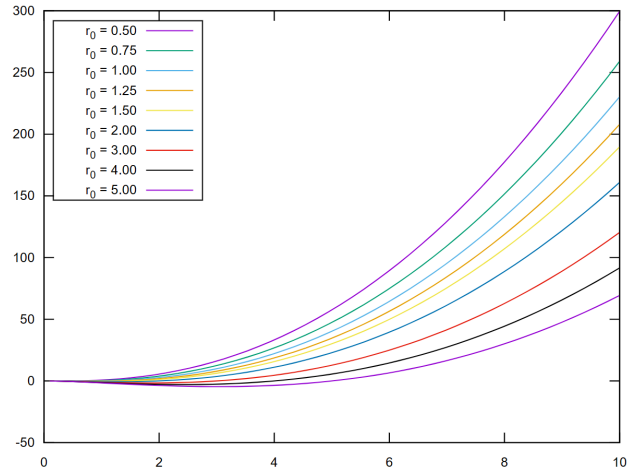


Figure 5: Thin plate spline basis function.

7.2 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, he 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	0.995308	0.267839	-0.110515
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.

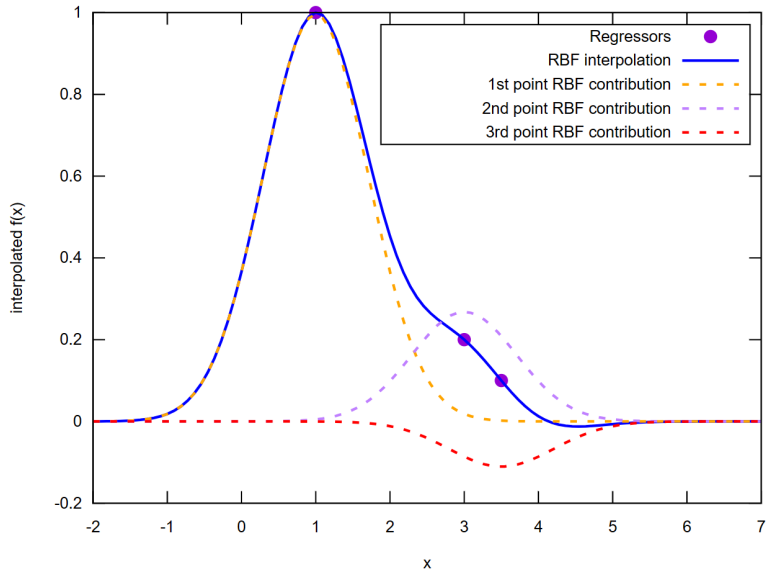


Figure 6: Reproduction of figure 2.2 from Wilna du Toit [5] showing the contribution of each radial basis function to the interpolation.

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.

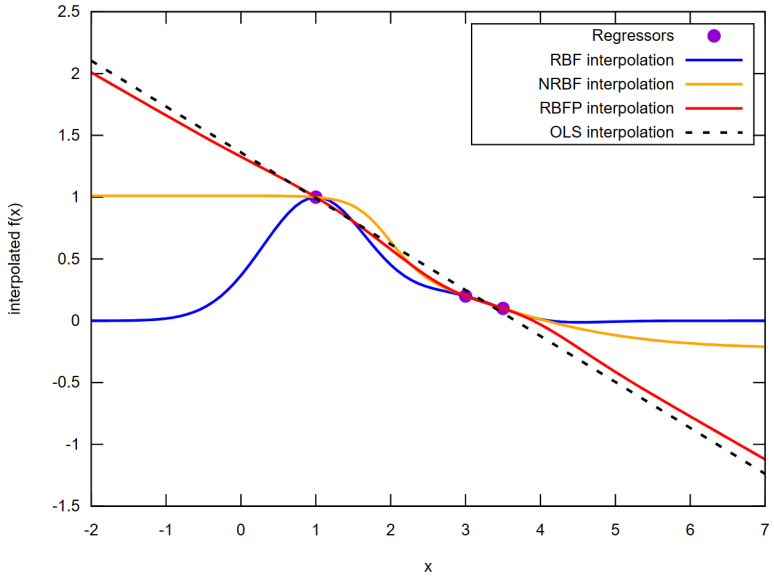


Figure 7: Comparison of RBF, NRBF, RBFP and OLS methods in a simple 1D case.

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.

7.3 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.

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	2.67147e-16	-2.83627e-16	-1.49186e-16	0.198347	-0.301653		
NRBF	-8.825	9.325	-3.875	19.95	-14.575		
RBFP	-3.38271e-17	4.82958e-16	-2.09766e-16	-3.88578e-16	-4.02456e-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.

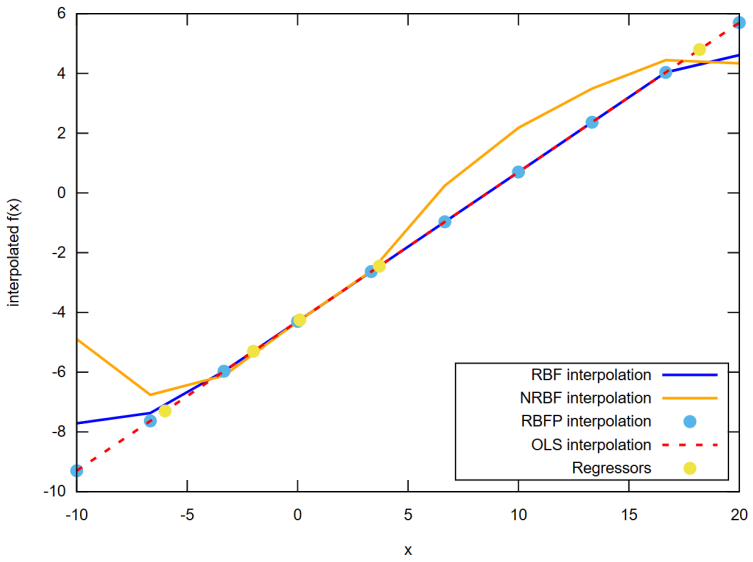


Figure 8: Interpolation of f, a linear 1D function.

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 $x = 0$.

7.4 Case 3: a 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 one hundred points are known. The interpolation will be based on the evaluation of ten points.

The process of creating known points and interpolated points is randomized. In order 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 :

$$f(\mathbf{x}) = f(x_1, x_2, x_3, x_4) = \sum_{i=1}^{4} x_i \cdot \exp\left(-\frac{x_i}{2}\right)$$

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

n°	f(**x**)	RBF(**x**)	ϵ_{RBF}	NRBF(**x**)	ϵ_{NRBF}	RBFP(**x**)	ϵ_{RBFP}	OLS(**x**)	ϵ_{OLS}
1	3.21923	3.22184	0.000812333	3.22828	0.00281112	3.21996	0.000228083	3.35424	0.0419404
2	2.85787	2.85812	8.72191e-05	2.85116	-0.0023462	2.85734	-0.000182561	2.77636	-0.0285191
3	2.10831	2.10438	-0.00186651	2.10628	-0.000965264	2.11175	0.00162928	2.11647	0.00386635
4	2.6274	2.62862	0.000465369	2.6284	0.000381023	2.62711	-0.000109242	2.69584	0.0260481
5	1.66816	1.667	-0.000692236	1.66169	-0.00387759	1.66745	-0.000423088	1.69101	0.0136979
6	2.42807	2.42927	0.000495789	2.4284	0.000138227	2.42697	-0.000454463	2.50727	0.0326199
7	2.15976	2.1626	0.00131162	2.16867	0.00412489	2.16212	0.00108879	2.21375	0.0249963
8	3.39492	3.39061	-0.00127198	3.39201	-0.000857317	3.39693	0.000592055	3.40248	0.00222654
9	3.45539	3.44939	-0.00173612	3.45818	0.000809586	3.45846	0.000890716	3.38398	-0.0206643
10	3.73682	3.7309	-0.00158395	3.70843	-0.00759824	3.72978	-0.00188261	3.6554	-0.0217888

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

The relative error ϵ is calculated as follows:

$$\epsilon = \frac{o_{\text{interpolated}} - o_{\text{real}}}{|o_{\text{real}}|}$$

Then, the relative error is plotted for each interpolated point to compare the different interpolation methods.

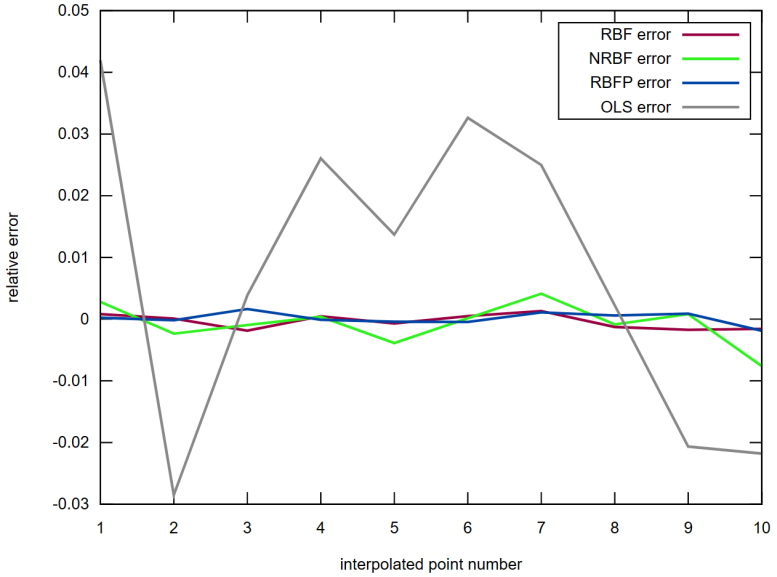


Figure 9: Error in f interpolation, a 4D function.

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 methods with the smallest errors are RBF and RBFP, which seem equivalent in this case.

7.5 Case 4: a 10D test case

In this test case, the aim is to see what effect the rescaling methods implemented in the library have. A function f from $\mathbb{R}^10$ to $\mathbb{R}$ will be considered. It will be assumed that 150 points are known and that the number of points to be interpolated is 30. The function is the same as before, i.e. :

$$f(\mathbf{x}) = f(x_1, x_2, \cdots, x_{10}) = \sum_{i=1}^{10} x_i \cdot \exp\left(-\frac{x_i}{2}\right)$$

After compilation and execution of the test case, the results are shown in the table below.

n°	f(x)	no norm.	error	mean norm.	error	min-max norm.	error	z-score norm.	error
1	65.8276	27.3033	-0.585231	71.851	0.0915032	71.851	0.0915032	65.9822	0.00234873
2	24.6157	17.249	-0.299268	33.7724	0.371988	33.7724	0.371988	30.9131	0.255832
3	52.9445	40.5617	-0.233883	43.6015	-0.176469	43.6015	-0.176469	52.8839	-0.00114512
4	44.7761	33.8176	-0.24474	63.9951	0.429225	63.9951	0.429225	60.5197	0.351609
5	78.7422	19.9952	-0.746067	70.7576	-0.101402	70.7576	-0.101402	59.8883	-0.239439
6	24.4521	17.2624	-0.294035	56.1629	1.29685	56.1629	1.29685	45.1326	0.845751
7	18.939	31.3148	0.653455	35.3613	0.867116	35.3613	0.867116	38.708	1.04383
8	10.7492	11.3078	0.0519652	11.7895	0.0967723	11.7895	0.0967723	15.0754	0.402461
9	50.1393	13.3625	-0.733492	66.1534	0.319392	66.1534	0.319392	64.1738	0.27991
10	49.3736	14.7787	-0.700675	53.4264	0.0820838	53.4264	0.0820838	41.7951	-0.153492
11	26.912	12.5835	-0.532418	29.2108	0.0854199	29.2108	0.0854199	23.1537	-0.139648
12	27.2502	2.90664	-0.893335	4.44804	-0.836771	4.44804	-0.836771	11.5684	-0.575475
13	-2.06596	4.93906	3.39068	-7.5765	-2.66729	-7.5765	-2.66729	-5.12151	-1.47899
14	62.9347	19.4695	-0.69064	75.8217	0.204769	75.8217	0.204769	53.7534	-0.145886
15	25.8847	47.8672	0.849244	49.1047	0.897051	49.1047	0.897051	54.3218	1.0986
16	37.4363	37.4642	0.000746602	32.708	-0.126301	32.708	-0.126301	40.0147	0.0688744
17	43.2462	32.4265	-0.250189	52.6703	0.217916	52.6703	0.217916	54.9807	0.27134
18	98.3447	13.658	-0.861122	60.4278	-0.385552	60.4278	-0.385552	45.446	-0.53789
19	70.1674	17.7933	-0.746416	86.1167	0.227303	86.1167	0.227303	70.352	0.00263125
20	51.6771	3.81769	-0.926124	18.7667	-0.636848	18.7667	-0.636848	16.7175	-0.6765
21	37.0819	33.763	-0.0895016	38.5236	0.0388793	38.5236	0.0388793	36.5708	-0.0137817
22	65.3008	6.52653	-0.900054	64.5182	-0.0119849	64.5182	-0.0119849	44.9875	-0.311073
23	86.3513	9.69711	-0.887702	69.6407	-0.193519	69.6407	-0.193519	50.3477	-0.416943
24	7.42623	4.12289	-0.44482	-0.591254	-1.07962	-0.591254	-1.07962	5.12077	-0.310449
25	36.2375	34.0362	-0.0607445	60.3164	0.664476	60.3164	0.664476	59.1674	0.63
26	23.4006	31.4594	0.344385	29.0533	0.241562	29.0533	0.241562	29.6517	0.267137
27	15.4307	17.612	0.141366	17.1215	0.109579	17.1215	0.109579	24.0213	0.556728
28	72.5072	23.3261	-0.678293	59.4708	-0.179795	59.4708	-0.179795	54.402	-0.249702
29	9.06553	5.2813	-0.41743	27.3242	2.01407	27.3242	2.01407	23.2273	1.56216
30	50.4284	39.0246	-0.226137	45.7399	-0.0929731	45.7399	-0.0929731	52.444	0.0399705

Table 7: RBF interpolation of f in 30 points with different rescaling, case of a 10D function.

The relative error (for each interpolated point) for each method is then plotted on a graph.

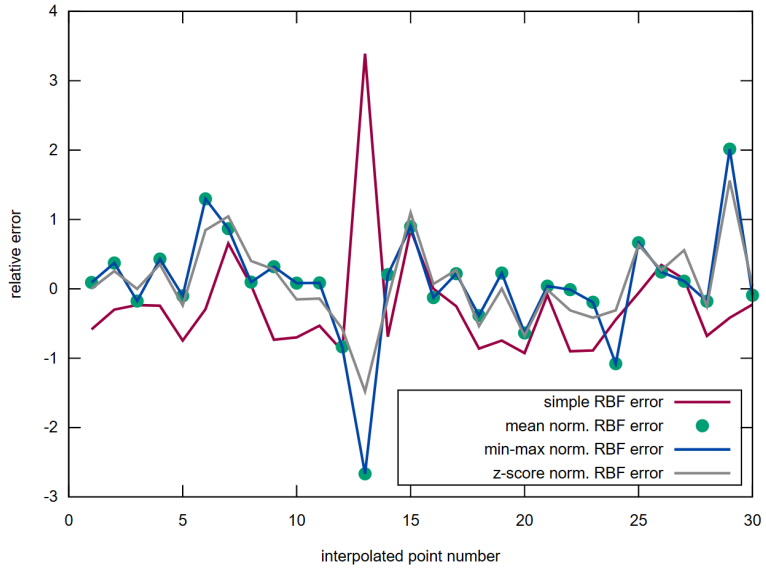


Figure 10: Error in f interpolation, a 10D function.

Here, on purpose, the drawn parameters have been made more anisotropic than before.

- Maybe reduce from 10D to 5D;
- Maybe change uniform sampling to Gaussian or another distribution;
- Independency of parameters needs to be discussed for rescaling and interpolation. Is my sampling creating real independent variables → not sure.

8 Bidings Python/C++

Conclusion

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.