mdisd

A C++ library for multi-dimensional interpolation of scattered data

Nils Malmberg - 10987738
Supervised by Nicola Parolini

Advanced Programming for Scientific Computing course
Politecnico di Milano

Table of contents

Introduction

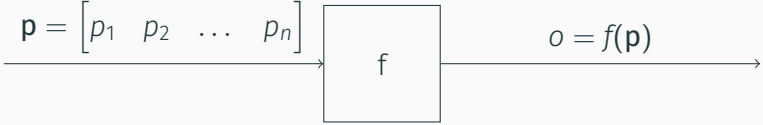
Context (p.2)

Consider a factory whose output quantity is the **quantity of finished products**. The input parameters are, for example, the **flow of raw materials**, the **state of fatigue of the employees**, the **state of the machines**, and so on.

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.

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.

We consider a system with n input parameters and one output. We define the vector $\mathbf{p} = \begin{bmatrix} p_1 & p_2 & \cdots & p_n \end{bmatrix}$ containing the input parameters and o the corresponding output. We define also f a function such that : $o = f(\mathbf{p})$.



For m sets of inputs, we can define $\mathbf{P}$ and $\mathbf{O}$ respectively the matrix containing all input parameters and the output vector such that:

$$\mathbf{P} = \begin{bmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,n} \end{bmatrix} \qquad \mathbf{O} = \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_m \end{bmatrix}$$

Goals

1. Estimate a new output o_{new} knowing:
   - the matrix $\mathbf{P}$ and the corresponding vector $\mathbf{O}$;
   - the input parameters $\mathbf{p_{new}}$.
2. Implement two interpolation methods:
   - RBF (Radial Basis Function);
   - OLS (Ordinary Least Squares).
3. Check that the library is working properly.
4. Write Python/C++ bindings.

Interpolation methods

Idea : each point defined by P and O "influences" its surroundings in the same way and in all directions according to the functionnal form $\Phi(r)$ where r is the radial distance [2]:

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

To determine the weights, we have to solve (using SVD decomposition) the m linear equations to impose that the interpolation is exact at all known data points:

$$\forall i \in [\![1; m]\!], \quad o_i = \sum_{k=1}^{m} \omega_k \cdot \phi(||\mathbf{p}_i - \mathbf{p}_k||) \tag{2}$$

- Multiquadratic : $\Phi(r) = (r^2 + r_0^2)^{1/2}$;
- Inverse multiquadratic : $\Phi(r) = (r^2 + r_0^2)^{-1/2}$;
- Thin-plate spline : $\Phi(r) = r^2 \log\left(\frac{r}{r_0}\right)$, $\Phi(0) = 0$;
- Gaussian : $\Phi(r) = \exp\left(-\frac{1}{2} \cdot \frac{r^2}{r_0^2}\right)$.

- Normalized RBF (NRBF):

$$o_{\text{new}} = \frac{\sum_{i=1}^{m} \omega_i \cdot \phi(||\mathbf{p}_{\text{new}} - \mathbf{p}_i||)}{\sum_{i=1}^{m} \phi(||\mathbf{p}_{\text{new}} - \mathbf{p}_i||)} \tag{3}$$

- RBF augmented (RBFP):

$$\begin{cases} o_{\text{new}} & = \sum_{i=1}^{m} \omega_i \cdot \phi(||\mathbf{p}_{\text{new}} - \mathbf{p}_i||) + Q(\mathbf{p}_{\text{new}}) \\ Q(\mathbf{p}_i) & = a_0 + \sum_{k=1}^{n} a_k \cdot p_{i,k} = 0 \quad , \quad \forall i \in [\![1; m]\!] \end{cases} \tag{4}$$

We consider the problem:

$$o_i = \alpha_i + \sum_{j=1}^{n} \beta_j p_{ij} \ , \ \forall i \in [\![1; m]\!] \tag{5}$$

$$\beta = \begin{bmatrix} \alpha \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{bmatrix} \qquad X = \begin{bmatrix} 1 & p_{11} & p_{12} & \cdots & p_{1n} \\ 1 & p_{21} & p_{22} & \cdots & p_{2n} \\ | & \vdots & \vdots & \vdots & \vdots \\ 1 & p_{m1} & p_{m2} & \cdots & p_{mn} \end{bmatrix} \qquad O = \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_m \end{bmatrix}$$

So the coefficients which fits the best the equation are [1]:

$$\hat{\beta} = \left(X^T X\right)^{-1} X^T O \tag{6}$$

Pre-processing

- Min-Max normalization: $T(p_{i,j}) = \dfrac{p_{i,j} - \min(p_{1,j}, \cdots, p_{m,j})}{\max(p_{1,j}, \cdots, p_{m,j}) - \min(p_{1,j}, \cdots, p_{m,j})}$
- Mean normalization: $T(p_{i,j}) = \dfrac{p_{i,j} - \mu_j}{\max(p_{1,j}, \cdots, p_{m,j}) - \min(p_{1,j}, \cdots, p_{m,j})}$
- Z-score normalization: $T(p_{i,j}) = \dfrac{p_{i,j} - \mu_j}{\sigma_j}$

where $\mu_j = \dfrac{1}{m}\sum_{l=1}^{m} p_{l,j}$; $\sigma_j = \sqrt{\dfrac{1}{m}\sum_{l=1}^{m} p_{l,j}^2 - \left[\dfrac{1}{m}\sum_{l=1}^{m} p_{l,j}\right]^2}$

Can we say that any function of $\mathbb{R}$ in $[0, 1]$ can be used for data rescaling ? **No**.

The library

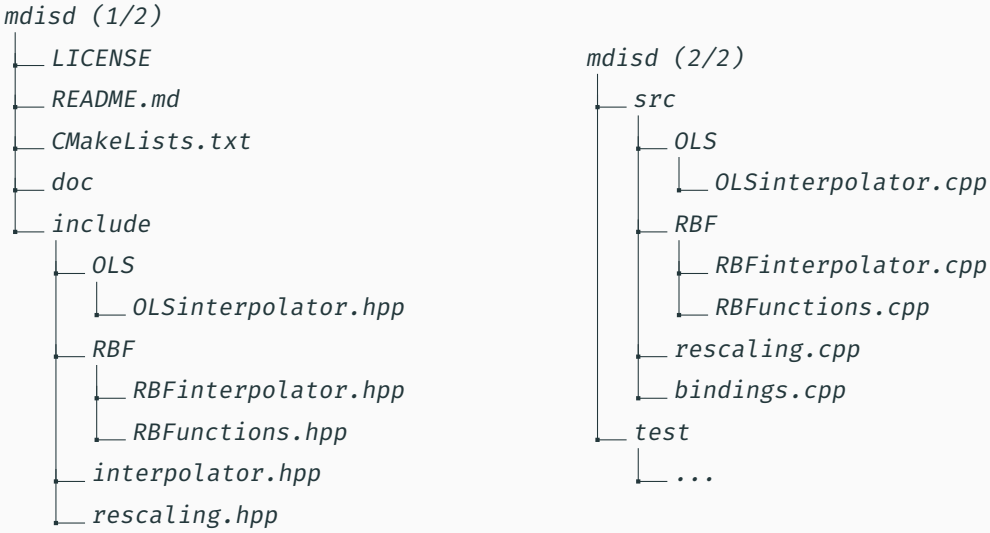


Figure 1: Tree of the mdisd library.

Interpolator class (p.9-10)

```
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;
};
```

```
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;
};
```

RBFunctions class (p.11)

```
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);
};
```

```
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;
};
```

```
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);
};
```

Test cases

Overview

- ~~Case 0: plot of radial basis functions~~;
- Case 1: 1D test case from Wilna du Toit [3];
- Case 2: 1D linear test case;
- Case 3: 4D ”isotropic” test case;
- Case 4: 4D ”anisotropic” test case with rescaling;
- Case 5: 2D/3D convergence of RBF interpolation error;
- ~~Case 6: test of Python bindings~~.

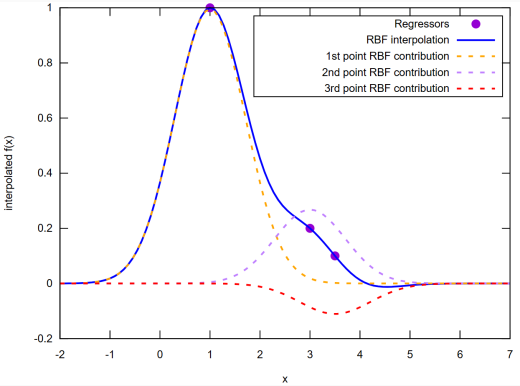


Figure 2: Reproduction of figure 2.2 from Wilna du Toit [3].

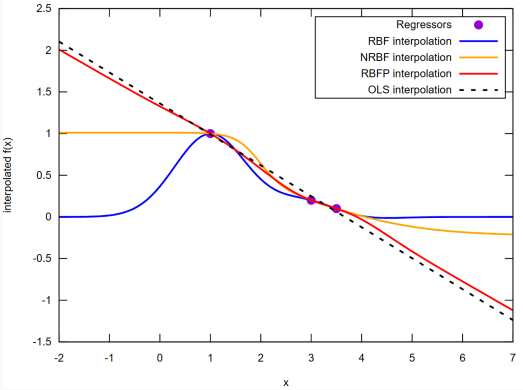


Figure 3: Comparison of RBF, NRBF, RBFP and OLS methods in a simple 1D case.

$$f(1) = 1 \;\; ; \;\; f(3) = 0.2 \;\; ; \;\; f(3.5) = 0.1$$

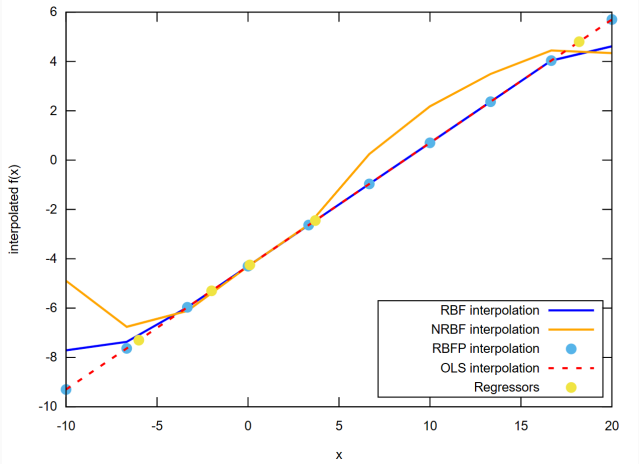


Figure 4: Interpolation of $f : x \rightarrow 0.5x - 4.3$.

$\forall (x_1, x_2, x_3, x_4) \in [0, 1]^4$,

$$f(\mathbf{x}) = \frac{3}{2} \cdot x_1^2 \cdot \cos(x_2 \cdot \pi) \cdot \sin(x_4 - x_3) - x_4 \cdot \exp\left(-\frac{x_1 + x_3}{2}\right) + \log(5 \cdot |x_3|) - 18.12 \cdot |x_2|^{x_4} \quad (7)$$

Sampling of 150 known points, interpolation of 10 point.

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

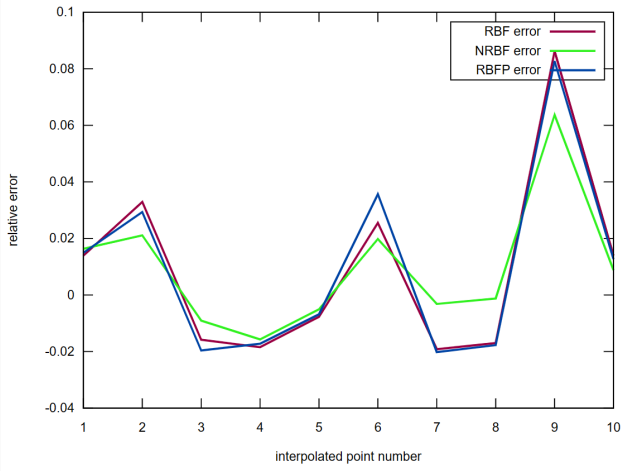


Figure 5: Interpolation relative error for each point.

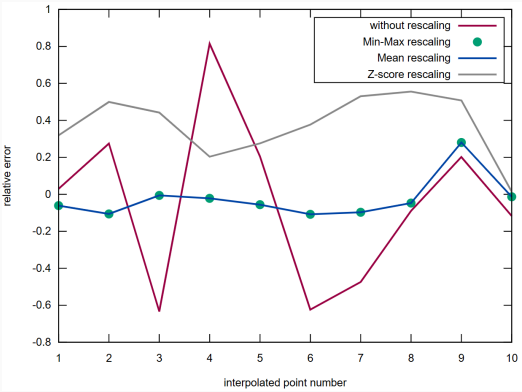


Figure 6: Error in f interpolation using RBF.

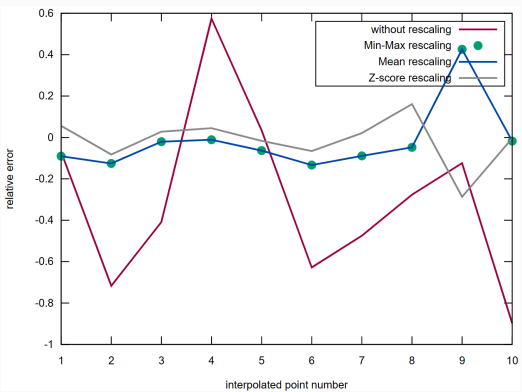


Figure 7: Error in f interpolation using NRBF.

$$x_1 \in [-1, 1] \;\; ; \;\; x_2 \in [10, 20] \;\; ; \;\; x_3 \in [1, 5] \;\; ; \;\; x_4 \in [0, 0.5]$$

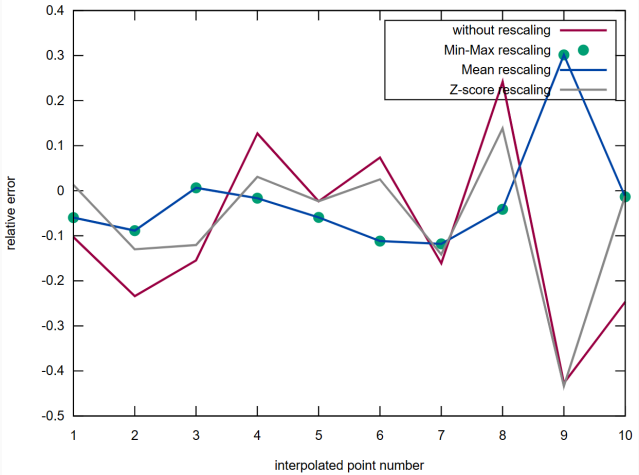


Figure 8: Error in f interpolation using RBFP.

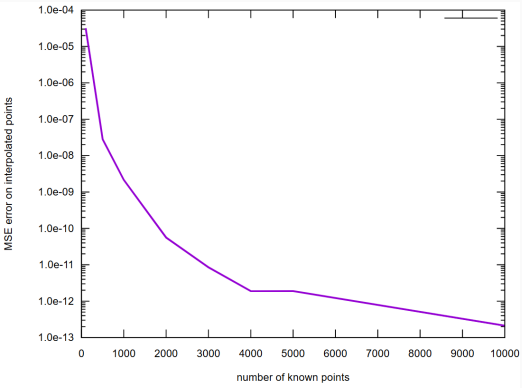


Figure 9: Interpolation Mean Square Error in function of the number of known points for F_{2D} using 2500 interpolated points.

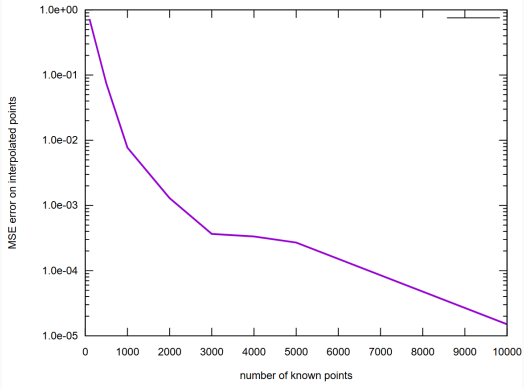


Figure 10: Interpolation Mean Square Error in function of the number of known points for F_{3D} using 2500 interpolated points.

Bindings

- Use of pybind11 for its simplicity and robustness;
- But some difficulties:
  - *interpolator* override $\Rightarrow$ add of *interpolator_with_coefficients*;
  - impossibility to use *RBFunctions* without *interpolator* $\Rightarrow$ double the functions.

How to use the library...

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

To run the cases, you need to change the path to the Eigen Library in the different Makefiles, then you can use the commands:

- *make*
- *make run*
- *make clean*

To generate and use the *.so* file. You need to change the paths to Python, pybind11 and Eigen library in the *CMakeLists.txt* file and then:

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

This will create library files for C++ and Python and you will just need during the compilation of your C++ project to specify the path of the headers (*include/* and subdirectories) and of the *.so* file. For a Python project, you will just need to add to the environment path the *build/* folder and then *mdisd_py*.

Conclusion

Conclusion (p.34)

At the end of this project, we have a **functional library in C++ and Python** for interpolating multi-dimensional scattered datasets. Several methods are proposed, including pre-processing methods.

Several test cases have been proposed to show how to use the library and to demonstrate how it works.

The library is **not perfect** and **can/need to be improved**.
But it let the possibility to the users **to easily upgrade** it by implement new radial basis functions, use their own pre-processing methods...

Possible improvements

- Add features:
  - implement other radial basis functions;
  - implement in RBFP different polynomials;
  - import a local RBF method permitting parallelization;
  - add choice for matrix inversion: SVD, PLU, LU, QR;
  - add other interpolation methods: Shepard's interpolation;
  - rescaling: use of hypercubes?
- Changes:
  - to have better bindings, change the library structure/classes;
  - is the Eigen library being used optimally?
  - does readme files contain enough information?

References

References

C. De Chaisemartin.
Ordinary least squares: the multivariate case.
Lecture, Paris School of Economics, 2011.

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery.
Numerical Recipes 3rd Edition: The Art of Scientific Computing.
Cambridge University Press, USA, 3 edition, 2007.

W. D. Toit.
Radial basis function interpolation.
2008.