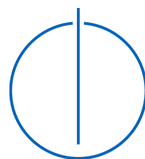
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Geometric Harmonics and Laplacian Pyramids

Darius Augsburger



DEPARTMENT OF INFORMATICS

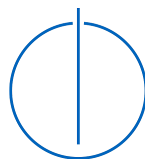
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Geometric Harmonics and Laplacian Pyramids

Geometrische Harmonische Funktionen und Laplace-Pyramiden

Author:	Darius Augsburger
Supervisor:	Univ.-Prof. Dr. Christian Mendl
Advisor:	Dr. Felix Dietrich
Submission Date:	15.04.2021



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.04.2021 Darius Augsburger

Acknowledgments

I would like to thank my advisor Dr. Felix Dietrich for his patience, advice and continuous support in understanding complex mathematical problems and in decision-making.

Abstract

This thesis is about the implementation of two out-of-sample extension algorithms, namely "Multi Scale Geometric Harmonics" and "Laplacian Pyramids", for empirical functions on data sets with the manifold assumption. The challenge is to design the software interfaces for the algorithms, improve the efficiency of the existing implementation, test and document it, and then to include everything as an open source solution in the python package datafold. The new implementations are demonstrated on suitable examples. The tests showed very good agreement with the examples from the original papers where the algorithms were first proposed.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**

2 State of the art **3**
2.1 Geometric Harmonics 3
2.1.1 Properties of Geometric Harmonics 4
2.1.2 Extension algorithm 5
2.1.3 Bandlimited extension 7
2.1.4 Gaussian extension 7
2.1.5 Multiscale extension 7
2.2 Laplacian Pyramids 9
2.2.1 Basic Laplacian Pyramids 10
2.2.2 Auto-adaptive Laplacian Pyramids 11
2.3 Datafold for data-driven model parametrization 13

3 Geometric Harmonics and Laplacian Pyramids **15**
3.1 Implementation requirements and constraints 15
3.1.1 Instantiation 15
3.1.2 Fitting the parameters 16
3.1.3 Necessary base classes 17
3.2 Implementation of Geometric Harmonics 19
3.2.1 Old Software Design 19
3.2.2 Optimized Software Design 28
3.3 Implementation of Multiscale Geometric Harmonics 35
3.3.1 The current state 35
3.3.2 The reworked model 37
3.4 Implementation of Laplacian Pyramids 41
3.4.1 The current state 43
3.4.2 Proposed enhancements 43

3.5 Testing and Demonstrations 44
3.5.1 Conducted tests 45
3.5.2 Unit circle to the plane 45
3.5.3 Image interpolation 51
3.5.4 Synthetic example 53

4 Conclusion **55**

List of Figures **56**

List of Tables **58**

Bibliography **59**

1 Introduction

Often tasks like clustering, regression and classification where large amount of data is involved cannot be computed directly. Instead, a subsample of data is produced. Thereby, the problem size is reduced drastically and after processing the subsample of data, it is scaled back to the original data. Geometric Harmonics [14, 8, 7] is one scheme to extend empirical functions this way. It is inspired by the vastly used Nyström method which is for example applied in partial differential solvers or in machine learning and spectral graph theory to subsample large data sets [2, 11, 26]. Another scheme for out-of-sample extension is the multiscale model Laplacian Pyramids that generates a smoothed version of a function with Gaussian kernels of decreasing bandwidths in each iteration [5]. Additionally, this method can be seen as an iterative version of a Nadaraya-Watson estimator [18]. Most of the several applications of the Laplacian Pyramids scheme utilize it for function approximation and out-of-sample extension [2, 16]. For both algorithms we predominantly apply the Gaussian kernel as kernel function which is itself a Radial Basis Function [3] since this type of function is in general the best choice to approximate and interpolate, especially for high-dimensional data [10].

Figure 1.1: The levels of the `MultiscaleGeometricHarmonicsInterpolator` predicting the train image beginning with the coarsest scale.

In this thesis, first we introduce two out-of-sample extension schemes, namely Geometric Harmonics and Laplacian Pyramids. For the Geometric Harmonics, we define the extension and restriction operators and show how they are in relation with the eigenpairs of a kernel k with certain properties. Additionally, we consider different extensions, among others the multiscale extension. Then we briefly outline its two interesting properties double orthogonality and variational optimality [7]. In the case of

Laplacian Pyramids, we first introduce the basic approach and then, based on this, we consider the Auto-adaptive Laplacian Pyramids [10]. Those algorithms are later implemented in the python package datafold [15]. After outlining the structure and available tools of the datafold package and considering the constraints added by the dependence to scikit-learn [19, 4] we focus the implementation documentation and optimization of the out-of-sample extension algorithms in datafold. The algorithms are either more or less implemented in datafold, namely `GeometricHarmonicsInterpolator`, `MultiscaleGeometricHarmonicsInterpolator` and `LaplacianPyramidsInterpolator`. After optimizing the Geometric Harmonics models and proposing improvements for the Laplacian Pyramids algorithm, we briefly outline conducted tests and finally, define some detailed demonstrations which generate results as Figure 1.1. One can clearly see, what multiscale actually means as the captured scale gets finer per iteration and how the Geometric Harmonics interpolates with higher frequencies per level.

In Section 2.1 we first consider the mathematical definitions and provide a single- and a multiscale Geometric Harmonics extension scheme. In the next Section 2.2 we go through the basic Laplacian Pyramids algorithm and then based on those introduce the Auto-adaptive Laplacian Pyramids. Furthermore, we introduce the python library datafold in Section 2.3 and outline its structure. The implementation of single-scale Geometric Harmonics is optimized in Section 3.2. After adjusting the structure of the single-scale Geometric Harmonics for inheritance ad reusability, we rework the multiscale model in Section 3.3. The Laplacian Pyramids implementation is outlined in Section 3.4 and we propose some optimizations on the implementation. Lastly, tests and demonstrations are outlined in Section 3.5 and the thesis is concluded in Section 4.

2 State of the art

In the first chapter, we will focus more on the mathematical background of the algorithms and definitions necessary to understand them than on implementation aspects. The algorithms are introduced and described in Section 2.1 and Section 2.2. Last but not least, in 2.3 we take a look at the structure of the datafold package [15], the available models and where our algorithms should be implemented.

2.1 Geometric Harmonics

Let $X \subset \overline{X}$ be two sets and μ be a finite measure on X. With the geometric harmonics, we can extend a function f defined on set X to $\overline{X}$ [14, 7]. Furthermore, we define a kernel $k : \overline{X} \times \overline{X} \to \mathbb{R}$. The selected kernel must satisfy the following properties regarding Coifman and S. Lafon [7]:

- k is symmetric.
- k is positive semi-definite. This property allows us to interpret the geometric harmonics as maximizing some concentration measure over X.
- k is bounded on $\overline{X} \times \overline{X}$ by a number $M > 0$.

Since k is positive semi-definite, there is a unique kernel Hilbert space $\mathcal{H}$ of functions defined on $\overline{X}$ for which k is the reproducing kernel [14]. In this thesis, we mostly use the Gaussian kernel which is defined by Rabin and Coifman [20] as

$$g_\varepsilon(x, y) = \exp\left(-\frac{\|x - y\|^2}{\varepsilon}\right). \tag{2.1}$$

Let k be a symmetric positive semi-definite kernel on $\overline{X} \times \overline{X}$. Furthermore, we define the operator $\mathbf{K} : L^2(X, d\mu) \to \mathcal{H}$ as

$$\mathbf{K}f(x) = \int_X k(x, y)\, f(y)\, d\mu(y)$$

with $x \in \overline{X}$. Then the adjoint $\mathbf{K}^* : \mathcal{H} \to L^2(X, d\mu)$ is defined as the restriction operator on the set X. That means, if $F \in \mathcal{H}$ and $x \in X$ then

$$\mathbf{K}^* F(x) = F(x).$$

Moreover, for a bounded k the operator $\mathbf{K}^*\mathbf{K} : L^2(X, d\mu) \to L^2(X, d\mu)$ is compact [14]. Since the operator $\mathbf{K}^*\mathbf{K}$ is self-adjoint, positive and compact, it grants a discrete set of eigenfunctions $\{\psi_j\}$ and non-negative eigenvalues $\{\lambda_j\}$. Thus, we can obtain the eigenfunctions and eigenvalues from the diagonalization of the kernel k on X if $x \in X$ [14]:

$$\int_X k(x,y)\,\psi_j(y)\,d\mu(y) = \lambda_j \psi_j(x)$$

By applying the Nyström method [26], we get the extension of these eigenfunctions called "geometric harmonics" to $x \in \overline{X}$ if $\lambda_j > 0$:

$$\Psi_j(x) = \frac{1}{\lambda_j}\int_X k(x,y)\,\psi_j(y)\,d\mu(y)$$

As we can see, performing interpolation with geometric harmonics builds up on the idea of the Nyström extension: instead of extending the geometric harmonics itself to a neighborhood region, the method allows interpolating arbitrary function values defined on a manifold. The inspiration of the term 'geometric harmonics' comes from ψ_j being extended as an average over its values, and thus can be thought of as verifying a certain form of mean value theorem [7].

Since we divide by the eigenvalues λ_j of a compact operator Coifman and S. Lafon [7] point out that this procedure is extremely ill-conditioned and propose a solution to bound the condition of the extension procedure. We fix a condition number $C = \frac{1}{\delta}$ with arbitrary $\delta > 0$ and define the following sets:

$$\begin{aligned} S_\delta &= \{j | \lambda_j \geq \delta * \lambda_0\}, \qquad (2.2) \\ L^2_\delta &= Span\{\psi_j | j \in S_\delta\}, \\ \mathcal{H}_\delta &= Span\{\Psi_j | j \in S_\delta\}. \end{aligned}$$

With the new finite-dimensional vector spaces the extension procedure $L^2_\delta \to \mathcal{H}_\delta$ has a condition number bounded from above by C. We can finally summarize the algebraic relation between $\mathbf{K}, \mathbf{K}^*, \Psi_j$ and ψ_j as

$$\begin{aligned} \mathbf{K}\psi_j &= \lambda_j \Psi_j \text{ (extension)}, \\ \mathbf{K}^*\Psi_j &= \psi_j \text{ (restriction)}. \end{aligned}$$

2.1.1 Properties of Geometric Harmonics

As mentioned by S. S. Lafon [14], the geometric harmonics have two interesting properties:

Double Orthogonality

The system $\{\Psi_j\}_{j \in S_\delta}$ forms an orthogonal basis of $\mathcal{H}_\delta$, and their restrictions $\{\psi_j\}_{j \in S_\delta}$ to X forms an orthogonal basis of L^2_δ.

Variational Optimality

For $F \in \mathcal{H}$ and restriction $f \in L^2_\delta(X, d\mu)$, the concentration of F over X is the Rayleigh quotient

$$c_X(F) = \frac{\|f\|_X}{\|F\|_{\overline{X}}}.$$

Furthermore, under the constraint that $F \perp \{\Psi_0, \Psi_1, ..., \Psi_{j-1}\}$ the function Ψ_j is a solution for the problem

$$\max_{F \in \mathcal{H}} c_X(F).$$

Thus, Ψ_0 is the most concentrated element of $\mathcal{H}$ on X.

2.1.2 Extension algorithm

The natural extension algorithm associated with the geometric harmonics which are obtained from a kernel k is described in detail by Coifman and S. Lafon [7]. It only considers one scale defined by k which is essentially arbitrary since it is unrelated to f. By scale the size of k's numerical support in $\mathbb{R}^n$ is meant.

Project $f \in L^2(X, d\mu)$ onto the space L^2_δ spanned by orthonormal system $\{\psi_j\}_{j \in S_\delta}$:

$$f \to P_\delta f = \sum_{j \in S_\delta} \langle f, \psi_j \rangle_X \psi_j \tag{2.3}$$

Use the extension Ψ_j of ψ_j to extend $P_\delta f$ on $\overline{x} \in \overline{X}$ as

$$F = \mathbf{E}f(\overline{x}) = \sum_{j \in S_\delta} \langle f, \psi_j \rangle_X \Psi_j(\overline{x}) \tag{2.4}$$

A more detailed description of the calculation steps is given in Algorithm 1 which is based on the already implemented `GeometricHarmonicsInterpolator` [15].

Algorithm 1: Geometric Harmonics

Input: Function space $X \subset \overline{X}$, function values f, target space $T \subseteq \overline{X}$, condition $\frac{1}{\delta}$
Output: Extended function F

```
1  P_δ f; f̂; F;
   // Precomputation phase
2  k = predefined kernel;
3  K = kernel matrix with entries K_ij = k(x_i, x_j), x_i, x_j ∈ X;
4  (λ, ψ) = eigenpairs of K;
5  for λ_j ≥ δλ_0 do
6  |   P_δ f = P_δ f + ⟨f, ψ_j⟩ψj;
7  end
8  for λ_j ≥ δλ_0 do
9  |   Ψ_j = (1/λ_j) ψ_j;
10 |   f̂ = f̂ + ⟨P_δ f, ψ_j⟩Ψ_j;
11 end
   // Prediction phase
12 K̄ = kernel matrix with entries K_ij = k(x_i, x_j), x_i ∈ T, x_j ∈ X;
13 F = K̄ f̂;
```

(η, δ)-extendable functions

This technique does not provide an extension for f but rather a filtered version, namely the orthogonal projection $L^2_\delta(X, d\mu)$ [14, 7]. Filtered version of functions constitute the set of (safely) $\overline{X}$-extendable functions. In the process of extension, since the condition number of the operation is $\frac{1}{\delta}$, $\log\left(\frac{1}{\delta}\right)$ digits are lost. A general empirical function f on X is extendable if the residual is smaller than a prescribed error:

$$\|f - \mathbf{P}_\delta f\| \leq \eta \tag{2.5}$$

There is in general no unique way to extend f as a function of $\mathcal{H}$, because its elements might not be determined by their restrictions to X. The second property gives us nevertheless an interpretation to the choice made by the algorithm. It returns the function with maximum concentration, which is equivalent to the function with minimal energy on $\overline{X}$. That means that the algorithm provides the best extension given the information at the disposal [14].

2.1.3 Bandlimited extension

The prolate spheroidal wave functions are introduced by Slepian and Pollak [22] as the solution of the problem of finding functions optimally concentrated in time and frequency. Those are bandlimited functions of unit energy that have maximum energy within an interval in the time domain. Their results are generalized to higher dimensions. By defining $\mathcal{H}^B$ to be the space functions of $L^2(\mathbb{R}^n)$ whose Fourier transforms are compactly supported in the ball centered at the origin of radius $\frac{B}{2}$, Slepian [21] generalizes the results to higher dimension. That means that $\mathcal{H}^B$ is the space of bandlimited functions of finite energy with bandwidth $\frac{B}{2}$. The generated space is a reproducing kernel Hilbert space [14, 7] with the so-called Bessel kernel

$$k_B^{(n)}(x,y) = \left(\frac{B}{2}\right)^{\frac{n}{2}} \frac{J_{\frac{n}{2}}\left(\pi B \|x-y\|\right)}{\|x-y\|^{\frac{n}{2}}}$$

and J_v being the Bessel function of the first kind and of order v. We refer to the operator $\mathbf{E}_B$ as the Bessel kernel with bandwidth $B > 0$ which computes the bandlimited extension of band B that is maximally concentrated on X [7].

2.1.4 Gaussian extension

The Gaussian kernel (2.1) which is mainly used in this paper is also in general widely used in the Machine Learning community. Furthermore, Coifman and S. Lafon [7] state that it is a limiting case of the Bessel kernels:

$$\lim_{n} \to \infty \frac{k_B^{(n)}\left(\sqrt{\frac{n}{\pi}}\|x-y\|\right)}{V_{B,n}} = \exp\left(-B^2\|x-y\|^2\right)$$

In this equation, $V_{B,n}$ represents the volume of the unit ball of radius B in $\mathbb{R}^n$. While this seems to mean that Bessel and Gaussian kernels are equivalent in higher dimensions, in lower dimensions Bessel kernels show oscillations, unlike Gaussian kernels [7].

2.1.5 Multiscale extension

For the Multiscale extension elaborated by Coifman and S. Lafon [7] we define $\overline{X} = \mathbb{R}^n$, X as smooth, compact submanifold C^∞ of dimension d and a function f defined on X. Furthermore, we assume $d\mu$ to be the Riemannian measure dx on X. The key point of this procedure is to consider several kernels, i.e. several scales for the extension. First, we consider the analysis of the two different Fourier analyses on the function f. A purely intrinsic analysis is obtained by the use of the eigenfunctions of the Laplace Beltrami operator (LBO) [1] Δ on X which are the analogue of the Fourier

basis to arbitrary submanifolds. As X is compact the spectrum of Δ is discrete and corresponds to pure frequency modes. The second analysis that can be performed is the classical Fourier transform in $\mathbb{R}^n$ which can be applied to various extensions of f. Both analyses and their relation are investigated thoroughly by Coifman and S. Lafon [7] by considering the restriction and extension operators. The relation of the intrinsic and extrinsic Fourier analysis of f then results in the proposed Multiscale extension scheme. It consists of two phases: As first phase, we have the precomputation phase in which one computes the minimal frequency band B_j for each eigenfunction ϕ_j of Δ to which it can be extended to using $\mathbf{E}_{B_j}$. The second phase is the extension phase which first computes the decomposition of f over ϕ_j. Subsequently, we require enough coefficients such that the relative error is of order η:

$$f = \sum_{j \in S} \langle f, \phi_j \rangle_X \phi_j + \mathcal{O}\left(\eta \|f\|_X\right)$$

To extend f one uses the precomputed extensions of ϕ_j:

$$F = \sum_{j \in S} \langle f, \phi_j \rangle_X \mathbf{E}_{B_j} \phi_j$$

F is a sum of functions that oscillate at intrinsic frequency ν^2 on X that vanish at distance $\frac{1}{\nu_j^2}$ from this set [7].

An easier description of the algorithm is proposed by Chiavazzo et al. [6] and Sonday [23]: We initially project the function f as described in Equation (2.3) at a coarse scale which corresponds to a large ε_0 for the Gaussian kernel (2.1). Then the residual $f - P_\delta f$ is projected at a finer scale ε_1. These steps are iterated for even finer scales $\varepsilon_l = \frac{\varepsilon_0}{\mu^l}$ with a fixed scale divisor $\mu > 1$, until the norm of the residual remains larger than a fixed admissible error. This description leads to Algorithm 2 which is the reference for the Multiscale Geometric Harmonics implementation in 3.3.2.

Algorithm 2: Multiscale Geometric Harmonics

Input: Function space $X \subset \overline{X}$, function values f, target space $T \subseteq \overline{X}$, condition $\frac{1}{\delta}$, scale ε, scale divisor μ, admissible error e

Output: Extended function F

```
// Precomputation phase
```

1 $\overline{f} = f$; $\varepsilon_0 = \varepsilon$; $\ell = 0$;

2 **while** $e \leq \|\overline{f}\|$ **do**

3 $K =$ kernel matrix with entries $K_{ij} = g_{\varepsilon_\ell}\left(x_i, x_j\right)$ and $x_i, x_j \in X$;

4 $\hat{f}_\ell = \hat{f}$ of Algorithm 1 Geometric Harmonics(X, $\emptyset$, $\overline{f}$, $\frac{1}{\delta}$) precomputation phase performed with kernel matrix K;

5 $\overline{f} = \overline{f} - P_\delta f$ also from Algorithm 1 precomputation phase;

6 $\varepsilon_{\ell+1} = \varepsilon_\ell \div \mu$;

7 ℓ++;

8 **end**

```
// Prediction phase
```

9 **for** *ℓ in range(len($\hat{f}$))* **do**

10 $\overline{K} =$ kernel matrix with entries $K_{ij} = g_{\varepsilon_\ell}\left(x_i, x_j\right)$ and $x_i \in T, x_j \in X$;

11 $F = F + \overline{K}\hat{f}_\ell$;

12 **end**

2.2 Laplacian Pyramids

The Laplacian Pyramids algorithm originated from image processing applications and was first introduced by Burt and Adelson [5]. This procedure decomposes the input image into a series of images, each capturing a different frequency band of the original image. Laplacian Pyramids was later proved to be a tight frame [9]. Furthermore, Rabin and Coifman [20] introduced a multiscale approach based on Laplacian Pyramids for high-dimensional data analysis which has been applied by Mishne and Cohen [17]. In Section 2.2.1 we first consider basic Laplacian Pyramids. Additionally, we introduce Auto-adaptive Laplacian Pyramids proposed by Fernández et al. [10] in Section 2.2.2. Another variant proposed by Fernández et al. [10] is Local Auto-adaptive Laplacian Pyramids. It serves better for data that is not equally distributed and that has different density characteristics. Even though this is an interesting topic, we will not go further into details in this thesis.

2.2.1 Basic Laplacian Pyramids

First, we consider the multiscale Laplacian Pyramids described by Rabin and Coifman [20] and Fernández et al. [10]. They both define the up-sampling, but not the reduction step. Our goal is to extend an empirical function f defined on a dataset X to N new points $x_i \in \mathbb{R}^m$. Again, the Gaussian kernel is defined as in Equation (2.1) and in the proceeding referred as $K^{(\ell)}\left(x_i, x_j\right) = g_{\varepsilon_\ell}\left(x_i, x_j\right)$, where ε_0 is the initial scale and ε_ℓ is the scale of level ℓ. Then from the kernel construct $P^{(0)}$, the normalized kernel which represents the smoothing operator:

$$P^{(0)}\left(x_i, x_j\right) = \frac{K^{(0)}\left(x_i, x_j\right)}{\sum_k K^{(0)}\left(x_j, x_k\right)}.$$

The first representation of f is then the convolution $\overline{f}^{(0)} = f * P^{(0)}$. At level ℓ we then construct the kernel matrix $P^{(\ell)}$ with a smaller scale $\varepsilon_\ell = \frac{\varepsilon_0}{\mu^\ell}$ and a fixed parameter value $\mu > 1$. Afterwards, the residual $d^{(\ell-1)} = f - \overline{f}^{(\ell-1)}$ is required which captures the error of the approximation of f at level $\ell - 1$. A more detailed representation of f is afterwards given by

$$\overline{f}^{(\ell)} = \overline{f}^{(\ell-1)} + d^{(\ell-1)} * P^{(\ell)}.$$

Once the norm of the residual vector $d^{(\ell)}$ is smaller than a predefined tolerance, which is comparable to the admissible error of the Multiscale Geometric Harmonics, the algorithm stops at iteration L. On this level, the Laplacian Pyramids model has the form

$$\overline{f}^{(L)} = \overline{f}^{(0)} + \sum_{\ell=1}^{L} d^{(\ell-1)} * P^{(\ell)}.$$

Now extend this multiscale representation to a new data point $x \in \mathbb{R}^M$ by setting

$$\overline{f}^{(L)}(x) = \sum_j f\left(x_j\right) P^{(0)}\left(x, x_j\right) + \sum_{\ell=1}^{L} \sum_j d^{(\ell-1)}\left(x_j\right) P^{(\ell)}\left(x, x_j\right).$$

The smoothing kernels can directly be extended with $K^{(\ell)}\left(x, x_j\right) = g_{\varepsilon_\ell}\left(x, x_j\right)$ for a new point x as

$$P^{(\ell)}\left(x, x_j\right) = \frac{K^{(\ell)}\left(x, x_j\right)}{\sum_k K^{(\ell)}\left(x, x_j\right)}.$$

Fixing a small error threshold may easily cause overfitting since the error of the basic Laplacian Pyramids method decays fast [10]. The whole algorithm is outlined in Algorithm 3.

Algorithm 3: Laplacian Pyramids

Input: Function domain $X \subset \overline{X}$, function values f, target space $T \subseteq \overline{X}$, initial scale ε, scale divisor μ, admissible error e

Output: Extended function F

```
   // Precomputation phase
 1 d^(0) = f; f̄^(0) = 0; ε_1 = ε; ℓ = 1;
 2 while e ≤ error do
 3 |   K^(ℓ) = kernel matrix with entries K_ij = g_{ε_ℓ}(x_i, x_j) and x_i, x_j ∈ X;
 4 |   P^(ℓ) = normalized K^(ℓ);
 5 |   f̄^(ℓ) = f̄^(ℓ−1) + P^(ℓ) d^(ℓ−1);
 6 |   d^(ℓ) = f − f̄^(ℓ);
 7 |   error = ||d^(ℓ)||^2;
 8 |   ε_{ℓ+1} = ε_ℓ ÷ μ;
 9 |   ℓ++;
10 end
11 L = ℓ − 1;
   // Prediction phase
12 for ℓ = 1 to len(L) do
13 |   K^(ℓ) = kernel matrix with entries K_ij = g_{ε_ℓ}(x_i, x_j) and x_i ∈ T, x_j ∈ X;
14 |   P^(ℓ) = normalized K^(ℓ);
15 |   F = F + P^(ℓ) d^(ℓ−1);
16 end
```

2.2.2 Auto-adaptive Laplacian Pyramids

To prevent overfitting Fernández et al. [10] point out that a common way is to use an independent validation subset and stop the iteration when the error on that subset starts to increase. Since it introduces a random dependence on the choice of the particular validation subset, this can be problematic for small samples. Usually, k-fold Cross Validation is the standard solution to avoid this. One first distributes samples in k subsets and then uses $k-1$ of the in total N available samples for training. The left $N-(k-1)$ sets are then used for validation. Cross Validation then becomes Leave-One-Out Cross Validation (LOOCV) in the extreme case $N = k$ which has a rather high cost, in our case $\mathcal{O}\left(LN^3\right)$. The train process is stopped, when the error starts to increase. Even though it has originally a high cost, it is possible to apply LOOCV in $\mathcal{O}\left(LN^2\right)$. That means, Fernández et al. [10] showed how to perform LOOCV

for Laplacian Pyramids without essentially increasing cost. We can approximate the LOOCV validation values x_p with the values

$$\overline{f}^{(L)}\left(x_p\right) = \sum_j f\left(x_j\right) \overline{P}^{(0)}\left(x_p, x_j\right) + \sum_{\ell=1}^{L} \sum_j d^{(\ell-1)}\left(x_j\right) \overline{P}^{(\ell)}\left(x_p, x_j\right).$$

The introduced $\overline{P}^{(\ell)}$ modifies $P^{(\ell)}$ by setting its diagonal elements to zero. This time we furthermore, automatically pick a ε by first computing the matrix $W_{i,j} = x_i - x_j, \forall i, j$ and then fixing the initial scale $\varepsilon = 10 \max\left(W_{i,j}\right)$. Furthermore, Fernández et al. [10] bounds the maximum iterations as $maxIts = \log_2\left(5\varepsilon \div \min\left(W_{i,j}\right)\right)$ and in practice stops the iterations if the LOOCV error starts to grow which approximately equals the error in Algorithm 3. These changes finally result in Algorithm 4.

Algorithm 4: Auto-adaptive Laplacian Pyramids

Input: Function domain $X \subset \overline{X}$, function values f, target space $T \subseteq \overline{X}$, scale divisor μ

Output: Extended function F

```
// Precomputation phase
```

1 $W_{i,j} = x_i - x_j \forall i, j$; $\varepsilon_1 = 10 \max\left(W_{i,j}\right)$; $maxIts = \log_2\left(5\varepsilon \div \min\left(W_{i,j}\right)\right)$;
2 $d^{(0)} = f$; $\overline{f}^{(0)} = 0$; $\ell = 1$;
3 **while** $\ell < maxIts$ **do**
4     $K^{(\ell)}$ = kernel matrix with entries $K_{ij} = g_{\varepsilon_\ell}\left(x_i, x_j\right)$, $x_i, x_j \in X$ and 0-diagonal;
5     $P^{(\ell)}$ = normalized $K^{(\ell)}$;
6     $\overline{f}^{(\ell)} = \overline{f}^{(\ell-1)} + P^{(\ell)} d^{(\ell-1)}$;
7     $d^{(\ell)} = f - \overline{f}^{(\ell)}$;
8     $error^{(\ell)} = \|d^{(\ell)}\|^2$;
9     $\varepsilon_{\ell+1} = \varepsilon_\ell \div \mu$;
10   ℓ++;
11 **end**
12 $L = \arg\min_\ell \{error^{(\ell)}\}$;

```
// Prediction phase
```

13 **for** $\ell = 1$ *to len(L)* **do**
14     $K^{(\ell)}$ = kernel matrix with entries $K_{ij} = g_{\varepsilon_\ell}\left(x_i, x_j\right)$ and $x_i \in T, x_j \in X$;
15     $P^{(\ell)}$ = normalized $K^{(\ell)}$;
16     $F = F + P^{(\ell)} d^{(\ell-1)}$;
17 **end**

2.3 Datafold for data-driven model parametrization

The Python package datafold [15] provides data-driven models for point clouds to find an explicit manifold parametrization and to identify non-linear dynamical systems on these manifolds. A manifold is a usually unknown geometrical structure on which data is sampled. For high-dimensional point clouds a typical use case is to parametrize an intrinsic low-dimension manifold, with non-linear dimension reduction. For time series data, the underlying dynamical system is assumed to have a phase space that is a manifold.

The software architecture of datafold consists of three layers for maintaining a high degree of modularity [15]. Each one of the implemented data-driven models is either to be used on their own or in other model implementations, while dependencies should only appear unidirectional on functionality of lower levels or the same level. The associated base classes of each model are either directly from scikit-learn [19] or datafold its own provided specifications aligned to the scikit-learn API [4] in duck-typing manner. Those are for example the `BaseEstimator` which every base class inherits from, the `RegressorMixin` and the `MultiOutputMixin` which are later used to implement the considered interpolation algorithms. The three layers of datafold are shortly introduced in the following sections from higher to lower level.

datafold.appfold Package

The `datafold.appfold` Package is the highest level of datafold [15]. It accommodates those models that capture complex processing pipelines. Since the models in this layer provide a single point of access to multiple sub-models, the models are essentially "meta-models". They are intended to solve complex data-driven use cases or analysis tasks at the end of the machine learning process. Thanks to the modularization, there is a great flexibility in combining data process pipelines which makes testing model configurations and accuracies easier.

datafold.dynfold Package

The middle layer of the datafold library is the `datafold.dynfold` Package [15]. It contains all data-driven models which deal directly with point cloud manifolds or the dynamics of time series data. This package is the location of implementation for the algorithms considered in this paper. Actually, Laplacian Pyramids and single-scale Geometric Harmonics are already partially implemented and tested. Furthermore, first attempt of Multiscale Geometric Harmonics is also implemented but not working and not tested. The models of this layer can either be used in the `datafold.appfold`

"meta-models" or on their own for appropriate analysis tasks.

datafold.pcfold Package

The lowest level `datafold.pcfold` provides data structures like kernels and fundamental algorithms on data like an eigensolver `datafold.dynfold` [15]. The two data structures provided by datafold are the `PCManifold` which is part of the algorithm implementations and the `TSCDataFrame`. The first one is derived from the `numpy.nparray` [12] and furthermore, describes the local proximity between points with an attached kernel. It is applied to point cloud data with manifold assumption. Moreover, the `PCManifold` data structure can compute sparse/dense matrices of different distance metrics and eigenpairs with different backends. An important method of the `PCManifold` for the implementations is `optimize_parameters` which estimates a suitable scale and cut off for a Gaussian kernel. The `TSCDataFrame` on the other hand represents a collection of time series data. We do not go into detail on this data structure, since it is not important for this thesis.

3 Geometric Harmonics and Laplacian Pyramids

This chapter is about the implementation of the single- and Multiscale Geometric Harmonics and the Laplacian Pyramids. We use the provided scikit-learn API [4] to implement the algorithms since datafold [15] relies on it. In Section 3.1 we first focus on the implementation requirements and constraints, which come with datafold and the scikit-learn API. After considering the restrictions, in Section 3.2 the current state of the Geometric Harmonics algorithm is analyzed, we implement a better software architecture and optimizations. Based on the optimized `GeometricHarmonicsInterpolator`, we develop a new working model for Multiscale Geometric Harmonics in Section 3.3. Lastly, in Section 3.4 we consider the current implementation of Laplacian Pyramids and propose some optimizations.

3.1 Implementation requirements and constraints

Datafold is using the scikit-learn API [4] when implementing estimators for compatibility reasons [15]. The predominant object of this API is the estimator, which can be a classifier or regressor. In our case, all estimators are regressors. Since all estimators fit a model based on training data and are capable of inferring some properties on new data, they implement a method called `fit` to estimate necessary parameters. We first consider the instantiation in Section 3.1.1, then the purpose of the `fit` method in Section 3.1.2 and in the last Section 3.1.3 we take a look at the base classes required by our models.

3.1.1 Instantiation

Regarding to the scikit-learn API [4] the `__init__` method should not take the actual training data as argument, since this is part of the `fit` method. Rather, it concentrates on the creation of the object. It should accept arguments with predefined values, such that the user can instantiate an estimator without passing any arguments. Those parameters correspond to the problem the estimator is intended to solve. Therefore, they are

remembered as initial arguments. Instead of documenting them as “Attributes”, they are documented in the estimators “Parameters” section.

When doing model selection, scikit-learn relies on the arguments accepted by `__init__`, corresponding to an instance attribute. In this function, simple association should take place, meaning it should not implement data validation or any other logic. This kind of functionality must be implemented in the `fit` method, otherwise scikit-learn algorithms like the `GridSearchCV` would also perform a validation in their `set_params` method [19]. An initiation method would then look like this:

```
// Example
def __init__(self, param1="default1", param2="default2"):
    self.param1 = param1
    self.param2 = param2
```

The detailed parametrization of each algorithm follows in the corresponding implementation section.

3.1.2 Fitting the parameters

The next step is to estimate necessary parameters in the model by implementing the `fit` method. It takes the training data as arguments, which corresponds to one array for unsupervised learning and two arrays in the case of supervised learning [4]. Since our estimators correspond to supervised learning, our `fit` methods should accept two arrays. The arguments of the fit method for our implementations with their corresponding type are listed in Table 3.1. The method may accept additional keywords arguments if necessary, it should be restricted to directly data dependent variables. Calling it repeatedly must yield the same result. Therefore, `fit` is an idempotent function as long as it does not depend on some random process. Also, if the parameter `warm_start` is set to true and the estimator does support it, the previous state of trainable parameters are used instead of the default initialization strategy. This feature is not considered for our implementations. Furthermore, it should return the object itself to facilitate quick one-liners.

Parameter	**Type**
X	array-like, shape (n_samples, n_features)
y	array, shape (n_samples)
kwargs	optional data-dependent parameters.

Table 3.1: Arguments of the `fit` method.

Attributes estimated from the data must end with a trailing underscore. They are overridden every time `fit` is executed [4]. Furthermore, iterative algorithms should let specify the number of iterations by the integer parameter *n_iter*. Even though, the algorithms of this thesis are iterative, we do not implement this parameter, since the algorithms provide meaningful methodology to handle the iterations.

3.1.3 Necessary base classes

Since datafold [15] depends on scikit-learn [19], we can prevent a lot of boilerplate code by inheriting from their base classes. Even thought scikit-learn provides a lot of base classes, we only need the three base classes described in the following subsections for the implementation of our algorithms.

BaseEstimator

The `BaseEstimator` is the base class for all estimators in scikit-learn [19]. Since our algorithms are all estimators, they all extend the `BaseEstimator`. As one can see in Figure 3.1, it already provides the methods `get_params()`, `set_params` and `_get_tags()` which we otherwise should implement by ourselves regarding the scikit-learn API [4]. The `get_params()` method is optional and returns a dict of all `__init__` parameters, together with their values. With `set_params` one can set parameters during grid searches what makes it necessary. Also, since scikit-learn version 0.21 the `_get_tags()` method that returns the estimator tags as dictionary is necessary [19]. Tags allow programmatic inspection of the estimators capabilities and are used by the `check_estimator` function and the `parametrize_with_checks` decorator to determine which checks to run and what input data is appropriate. To add additional tags,

```
BaseEstimator
-------------
+n_features_in_
-------------
+get_params(deep = True)
+set_params(**params)
#_more_tags()
#_get_tags()
#_check_n_features(X, reset)
#_validate_data(X, y = "no_validation", reset = True, validate_separately = False, **check_params)
```

Figure 3.1: Class diagram of the scikit-learn `BaseEstimator`

we have to override the `_more_tags()` method and define the default values of the new tags. The `_more_tags()` provided by the `BaseEstimator` returns a dictionary of

tags and their default value. All tags and their precise meaning are available on the scikit-learn website.

MultiOutputMixin

By extending the multi-output mixin, we get an additional tag through its extension of the `_more_tags()` method [4]. As one can see in Figure 3.2, this is its only purpose. The tag added by this mixin is `'multioutput': True` which is `False` by default. It tells, whether a regressor supports multi-target outputs or a classifier supports multi-class multi-output.

Figure 3.2: Class diagram of the scikit-learn `MultiOutputMixin`

RegressorMixin

Since our algorithms are supervised learning algorithms, they extend the `RegressorMixin` class [4]. The estimator type is delegated by the attribute declaration `_estimator_type = "regressor"`. Furthermore, this mixin adds necessary tags for regressors with its `_more_tags()` override, namely `{'requires_y': True}`. The tag `requires_y` implies that the estimator requires target values as input for the `fit` method. Last but not least, the `RegressorMixin` provides the method `score(X, y, sample_weight = None)` with the parameters `X` providing test samples, `y` being the true target values and `sample_weight` providing sample weights if necessary. It returns a score ≤ 1 and may be negative, since models may be arbitrarily bad. This means, the higher the score the better the model predicts the data.

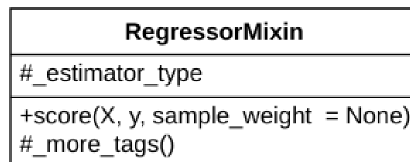


Figure 3.3: Class diagram of the scikit-learn `RegressorMixin`

3.2 Implementation of Geometric Harmonics

This section is about the Geometric Harmonics algorithm. We first document and review the old implementation in Section 3.2.1. In the second Section 3.2.2 we remedy the upcoming issues and then implement optimizations like an automated scale and cut-off selection.

3.2.1 Old Software Design

The single-scale "Geometric Harmonics" algorithm is already implemented in datafold [15] as the class `GeometricHarmonicsInterpolator` which inherits from the `RegressorMixin`, `MultiOutputMixin` and `BaseEstimator` as it should. The class diagram of the `GeometricHarmonicsInterpolator` is depicted in Figure 3.4.

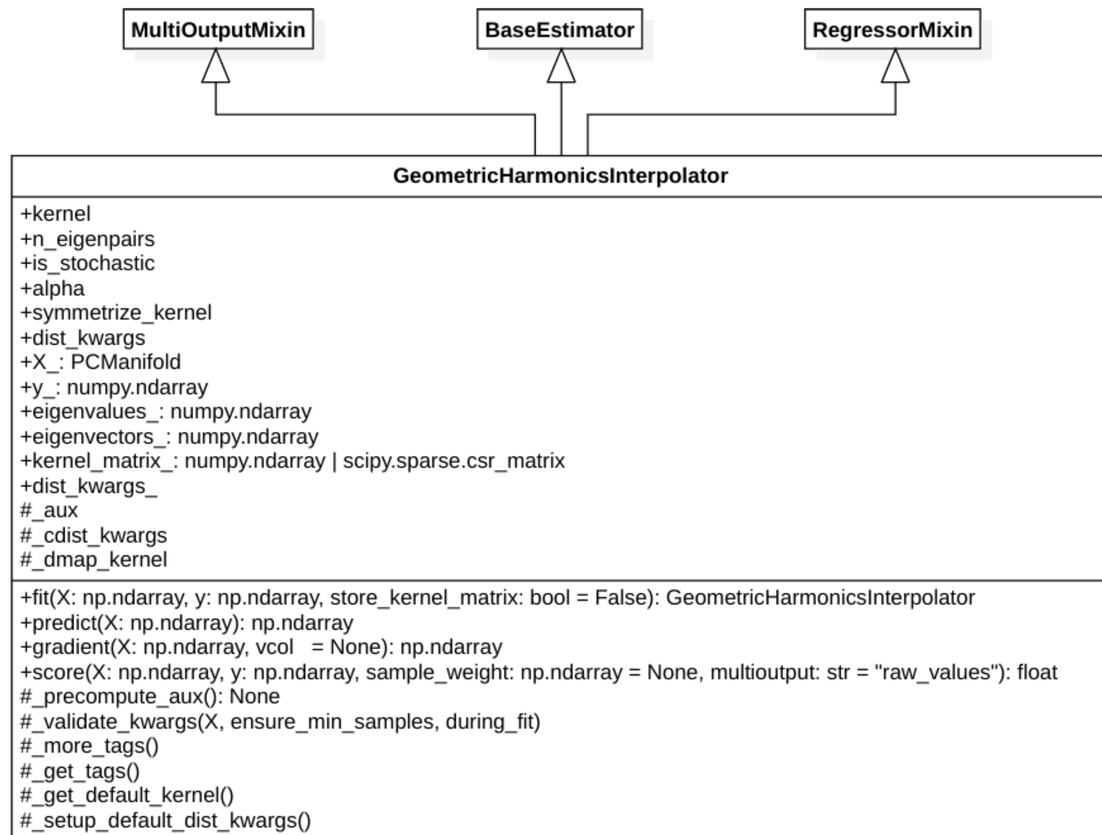


Figure 3.4: Class diagram of of the `GeometricHarmonicsInterpolator`.

Parameters

It accepts the parameters listed in Table 3.2 with their matching default value on initialization. Corresponding to the extension algorithm described in Section 2.1.2 the `kernel` parameter equals the type of kernel used to calculate the kernel matrix K and `n_eigenpairs` gives some restriction to the amount of calculated eigenvalues λ and eigenvectors ψ. This software structure does not provide an input parameter for the condition number $\frac{1}{\delta}$, yet.

Parameter	Description	Default
`kernel`	kernel to describe proximity between points	`None`
`n_eigenpairs`	Number of eigenpairs to compute from kernel matrix	`10`
`is_stochastic`	If True, the diffusion kernel matrix is normalized	`False`
`alpha`	Re-normalization parameter. "alpha=1" corrects the sampling density in the data as an artifact of the collection process.	`1`
`symmetrize_kernel`	If True, a conjugate transformation is performed if the current settings would lead to a non-symmetric kernel matrix.	`True`
`dist_kwargs`	Keyword arguments passed to the internal distance matrix computation.	`None`

Table 3.2: Parameters of the `GeometricHarmonicsInterpolator`

Attributes

In the fitting process described in the Section 3.2.1, the attributes listed in Table 3.3 are estimated to later calculate the prediction, gradient and the score of the model. The attribute `kernel_matrix_` is currently only in use for test cases. Furthermore, the estimator hast three protected attributes. Two of them are the vectors necessary for function extension `_aux` and the diffusion map kernel `_dmap_kernel`. The last one is `_cdist_kwargs` which is assigned to a dict that contains keys for a component-wise kernel computation [15].

Attribute	Description
`X_`	Training data during fit of shape '(n_samples, n_features)'. Required to be stored to perform out-of-sample interpolations.
`y_`	Target function values of shape '(n_samples, n_targets)', can be multi-dimensional.
`eigenvalues_`	Eigenvalues of diffusion kernel in decreasing order.
`eigenvectors_`	Eigenvectors of the kernel matrix.
`kernel_matrix_`	Computed kernel matrix, stored if 'store_kernel_matrix=True' during fit.
`dist_kwargs_`	Actual keyword dictionary arguments passed to the internal distance matrix computation.
`_aux`	Extension function to perform out-of-sample interpolation.
`_cdist_kwargs`	Keys required for a component-wise kernel computation.
`_dmap_kernel`	The computed diffusion map kernel.

Table 3.3: All attributes of the `GeometricHarmonicsInterpolator`.

Fitting the parameters

After fitting, the interpolator should be able to estimate function values for any $x \in \overline{X} \backslash X$. To achieve this, the vector $\hat{f}$ of Algorithm 1 must be precomputed and stored for prediction. This corresponds to the precomputation phase from line 2 to 11 in Algorithm 1. With the old implementation in datafold [15] we do this by first computing the kernel matrix K with the given parameter `kernel` of X, which is either defined by the user or is given as internal default as `GaussianKernel(epsilon=1)` by the function `_get_default_kernel()`. Subsequently, its eigenvalues λ and eigenvectors ψ are computed. The amount of eigenpairs is at the very beginning constrained by the parameter `n_eigenpairs`. Since computing eigenpairs is expensive this restriction can be very important to reduce computation time for certain cases. To generate the matrix K, we first construct a new `DmapKernelFixed` with the selected kernel method and the values of the parameters `is_stochastic`, `alpha` and `symmetrize_kernel`, and assign it to `_dmap_kernel`:

```
self._dmap_kernel = DmapKernelFixed(
    internal_kernel=internal_kernel,
    is_stochastic=self.is_stochastic,
    alpha=self.alpha,
    symmetrize_kernel=self.symmetrize_kernel,
)
```

The next step is to link the fixed diffusion kernel with the provided data X by creating a point cloud with data X and the kernel just constructed `_dmap_kernel`. In general, to construct such an object we use the data structure `PCManifold` introduced in Section 2.3 which is provided by datafold [15]. On the created point cloud object, we call the function `compute_kernel_matrix()` and unpack the result with the static method `read_kernel_output` of the class `PCManifoldKernel`:

```
kernel_output = self.X_.compute_kernel_matrix()
(
    kernel_matrix_,
    self._cdist_kwargs,
    ret_extra,
) = PCManifoldKernel.read_kernel_output(kernel_output)
basis_change_matrix = ret_extra["basis_change_matrix"]
```

The returned values of `read_kernel_output` are required to compute the eigenpairs of K which corresponds to the local variable `kernel_matrix_` and `_cdist_kwargs` is necessary to later compute the kernel matrix for the prediction. Hence, we can compute the eigenpairs λ and ψ of K. Since we use a diffusion map kernel, we can fall back to the class `_DmapKernelAlgorithms` which constitutes a collection of reusable algorithms for models with diffusion map kernels [15]. The algorithm we need is provided by its `solve_eigenproblem` method, which returns the largest `n_eigenpairs` eigenvalues and eigenvectors of a given kernel matrix. This method itself calls the datafold eigensolver function `compute_kernel_eigenpairs` which currently only provides the computation of the eigenpairs with the python library SciPy [25] but could potentially provide more backends to compute eigenpairs in the future. The datafold eigensolver wrapper should then select the best option for the given kernel matrix, depending on the parameters `n_eigenpairs`, `is_stochastic` and `is_symmetric`. In the case of the `scipy_eigensolver` function provided by datafold [15], it is decided between the SciPy eigensolvers `eig`, `eigh` and `eigsh`. Initially, we call the `solve_eigenproblem` method and pass the recently calculated kernel matrix stored in `kernel_matrix_`, the parameters `n_eigenpairs`, `is_stochastic` and `is_symmetric` necessary for optimization and the variable `basis_change_matrix`. The method then returns the optimally calculated eigenvalues and

eigenpairs for the given `kernel_matrix_` which we store in the estimator instance to use them in the ongoing steps:

```
(
    self.eigenvalues_,
    self.eigenvectors_,
) = _DmapKernelAlgorithms.solve_eigenproblem(
    kernel_matrix=kernel_matrix_,
    n_eigenpairs=self.n_eigenpairs,
    is_symmetric=self._dmap_kernel.is_symmetric,
    is_stochastic=self.is_stochastic,
    basis_change_matrix=basis_change_matrix,
)
```

After the computation of the eigenpairs λ and ψ, we should compute the projection $P_\delta f$ of the function `y` to bound the condition of the algorithm. Since this implementation does not provide an initialization parameter for the condition $\frac{1}{\delta}$, it also does not feature the calculation of the projection defined in Equation (2.3). Among other optimizations, this feature is implemented in the reworked model in Section 3.2.2. While the original extension algorithm would now calculate further with the projection $P_\delta f$ instead of the actual function values, we continue with the initially passed `y`.

We continue with the calculation of the lines 7 to 10 of Algorithm 1 with the method `_precompute_aux()` provided by our geometric harmonics interpolator [15]. It first asserts whether the just calculated eigenvalues and eigenpairs are not `None`. Then, instead of calculating the $\hat{f}$ values with the for loop proposed, we use matrix multiplication operations provided the NumPy library [12]. First we consider the calculation of the vectors Ψ_j which corresponds to the multiplication of the reciprocal of λ_j and ψ_j:

$$\Psi_j = \frac{1}{\lambda_j}\psi_j.$$

As each Ψ_j in this discrete implementation is a vector of the length of given function values, we define Ψ as matrix of same dimensions as the calculated matrix containing the eigenvectors `self.eigenvectors_`. The dimensionality of matrix `self.eigenvectors_` is determined by the shape `(len(y), n_eigenpairs)`. To finally map `self.eigenvectors_` and the corresponding array `self.eigenvalues_` with this considerations to the matrix Ψ, we first take the reciprocal of `self.eigenvalues_`. This can be done for the whole array with the NumPy function `reciprocal` [12]. The next step is to multiply each reciprocal $\frac{1}{\lambda_j}$ with its eigenvector ψ_j and to directly store the resulting Ψ_j in the jth column of

the matrix Ψ which is exactly the result of multiplying the matrix `self.eigenvectors_`- with the diagonalization of the eigenvalues `self.eigenvalues_` in this order.

$$\psi = \begin{bmatrix} \psi_0(x_0) & \psi_1(x_0) & \cdots & \psi_n(x_0) \\ \psi_0(x_1) & \psi_1(x_1) & \cdots & \psi_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_0(x_m) & \psi_1(x_m) & \cdots & \psi_n(x_m) \end{bmatrix}, \lambda^{-1} = \begin{bmatrix} \frac{1}{\lambda_0} & 0 & \cdots & 0 \\ 0 & \frac{1}{\lambda_1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\lambda_n} \end{bmatrix}$$

The datafold function `mat_dot_diagmat` can do this by taking the eigenvector matrix as first argument and the array of eigenvalues as second argument [15]. The result of this function is then our aimed matrix Ψ:

$$\begin{aligned} \Psi &= \psi * \lambda^{-1} \\ &= \begin{bmatrix} \psi_0(x_0) & \psi_1(x_0) & \cdots & \psi_n(x_0) \\ \psi_0(x_1) & \psi_1(x_1) & \cdots & \psi_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_0(x_m) & \psi_1(x_m) & \cdots & \psi_n(x_m) \end{bmatrix} \begin{bmatrix} \frac{1}{\lambda_0} & 0 & \cdots & 0 \\ 0 & \frac{1}{\lambda_1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\lambda_n} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\psi_0(x_0)}{\lambda_0} & \frac{\psi_1(x_0)}{\lambda_1} & \cdots & \frac{\psi_n(x_0)}{\lambda_n} \\ \frac{\psi_0(x_1)}{\lambda_0} & \frac{\psi_1(x_1)}{\lambda_1} & \cdots & \frac{\psi_n(x_1)}{\lambda_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\psi_0(x_m)}{\lambda_0} & \frac{\psi_1(x_m)}{\lambda_1} & \cdots & \frac{\psi_n(x_m)}{\lambda_n} \end{bmatrix} \end{aligned}$$

To get the desired vector $\hat{f}$, we now have to multiply the scalar products $\langle f, \psi_j \rangle$ and the matching Ψ_j and add the result to the current $\hat{f}$ as described in Algorithm 1 line 10. Again, this can be simplified by utilizing the NumPy functionalities [12]. We transpose the eigenvalue matrix `self.eigenvectors_` and calculate its product with the function

values `self.y_` which are equal to `y`:

$$\begin{aligned}
\langle f, \psi \rangle &= \psi * f \\
&= \begin{bmatrix} \psi_0(x_0) & \psi_1(x_0) & \cdots & \psi_n(x_0) \\ \psi_0(x_1) & \psi_1(x_1) & \cdots & \psi_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_0(x_m) & \psi_1(x_m) & \cdots & \psi_n(x_m) \end{bmatrix}^T \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_m) \end{bmatrix} \\
&= \begin{bmatrix} \psi_0(x_0) & \psi_0(x_1) & \cdots & \psi_0(x_m) \\ \psi_1(x_0) & \psi_1(x_1) & \cdots & \psi_1(x_m) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_n(x_0) & \psi_n(x_1) & \cdots & \psi_n(x_m) \end{bmatrix} \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_m) \end{bmatrix} \\
&= \begin{bmatrix} \sum_{i=0}^m \psi_0(x_i) f(x_i) \\ \sum_{i=0}^m \psi_1(x_i) f(x_i) \\ \vdots \\ \sum_{i=0}^m \psi_n(x_i) f(x_i) \end{bmatrix} = \begin{bmatrix} \langle f, \psi_0 \rangle \\ \langle f, \psi_1 \rangle \\ \vdots \\ \langle f, \psi_n \rangle \end{bmatrix}
\end{aligned} \tag{3.1}$$

Last but not least, the multiplication of the matrix Ψ and of the vector $\langle f, \psi \rangle$ yields the desired vector $\hat{f}$ which can then be used to extend the function as described in Section 3.2.1.

$$\begin{aligned}
\hat{f} &= \Psi * \langle f, \psi \rangle \\
&= \begin{bmatrix} \frac{\psi_0(x_0)}{\lambda_0} & \frac{\psi_1(x_0)}{\lambda_1} & \dots & \frac{\psi_n(x_0)}{\lambda_n} \\ \frac{\psi_0(x_1)}{\lambda_0} & \frac{\psi_1(x_1)}{\lambda_1} & \dots & \frac{\psi_n(x_1)}{\lambda_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\psi_0(x_m)}{\lambda_0} & \frac{\psi_1(x_m)}{\lambda_1} & \dots & \frac{\psi_n(x_m)}{\lambda_n} \end{bmatrix} \begin{bmatrix} \langle f, \psi_0 \rangle \\ \langle f, \psi_1 \rangle \\ \vdots \\ \langle f, \psi_n \rangle \end{bmatrix} \\
&= \begin{bmatrix} \sum_{i=0}^n \langle f, \psi_i \rangle * \frac{\psi_i(x_0)}{\lambda_i} \\ \sum_{i=0}^n \langle f, \psi_i \rangle * \frac{\psi_i(x_1)}{\lambda_i} \\ \vdots \\ \sum_{i=0}^n \langle f, \psi_i \rangle * \frac{\psi_i(x_m)}{\lambda_i} \end{bmatrix}
\end{aligned}$$

All these steps combined results in the following sequence of operations which directly stores the outcome in the variable `self._aux`:

```
self._aux = mat_dot_diagmat(
    self.eigenvectors_, np.reciprocal(self.eigenvalues_)
) @ (self.eigenvectors_.T @ self.y_)
```

The last segment of the fitting method to mention is the possibility to pass the boolean argument `store_kernel_matrix` to store the kernel in the instance which is by default `False` [15]. This feature is currently applied to assert features of the kernel matrix in unit tests.

Predicting target values

A shorter procedure is the prediction of target values for arbitrary $x \in \overline{X}$ which is described in the prediction phase of Algorithm 1 line 12 and 13. The implemented method just takes the argument `X` representing the set $T \subseteq \overline{X}$ for which the function values should be calculated while the set X is represented by the variable `self.X_`-from the fitting process [15]. With these variables we compute the kernel matrix $\overline{K}$ by again using the function `compute_kernel_matrix` of `self.X_` but this time we pass the given argument `X` as parameter `Y` and the `_cdist_kwargs` of the fitting kernel matrix computation:

```
kernel_output = self.X_.compute_kernel_matrix(Y=X, **self._cdist_kwargs)
kernel_matrix_, _, _ = PCManifoldKernel.read_kernel_output(
    kernel_output=kernel_output
)
```

The resulting `kernel_matrix_` is then used to yield the extended function F by multiplying $\overline{K}$ with the vector $\hat{f}$ represented by `self._aux`. Furthermore, we apply the NumPy `squeeze` method [12] to remove axis of length one:

```
np.squeeze(kernel_matrix_ @ self._aux)
```

Lastly, the computed function F is returned by the predict method.

Calculating a Score and the Gradient

The last two interesting methods of the old `GeometricHarmonicsInterpolator` are the `score` method, which is important to compare the results of different estimators and parameter constellations, and the `gradient` method, which computes the gradient of the interpolator at given points [15]. The `gradient` method is not content of the optimization process but contains known bugs and therefore, should be considered in future work on the model. The `score` method calculates the negated root mean squared error for a prediction on the two passed arguments `X` representing the data which is to be evaluated, and `y` representing the actual target values on the set `X`. Furthermore, one

can pass the arguments `sample_weight` to define a weight on the sample weights and `multioutput` to tell whether it should return the "raw values" or the "uniform average" of the set of errors. As an error is by nature better if it is smaller, the score is negated to comply with the scikit-learn API constraint "higher score is better" [4]. Since this approach forces us to define the tag `poor_score` as True to pass the scikit-learn tests, we later change the value of the score to better comply with this constraint.

3.2.2 Optimized Software Design

After getting to know the old software design of the `GeometricHarmonicsInterpolator` one can optimize a few things regarding the constraints to be fulfilled, the software structure and the algorithm itself. First we take a look at the parameters and attributes of the estimator. Then we abstract some processes to make them available for subclasses, especially for the multiscale extension, and complete the algorithm itself by adding the projection of the function to calculate the actually desired extension [7]. Finally, we implement the automated calculation of an optimized initial scale and cut-off. The new class diagram in Figure 3.5 gives a first insight on changes to the model.

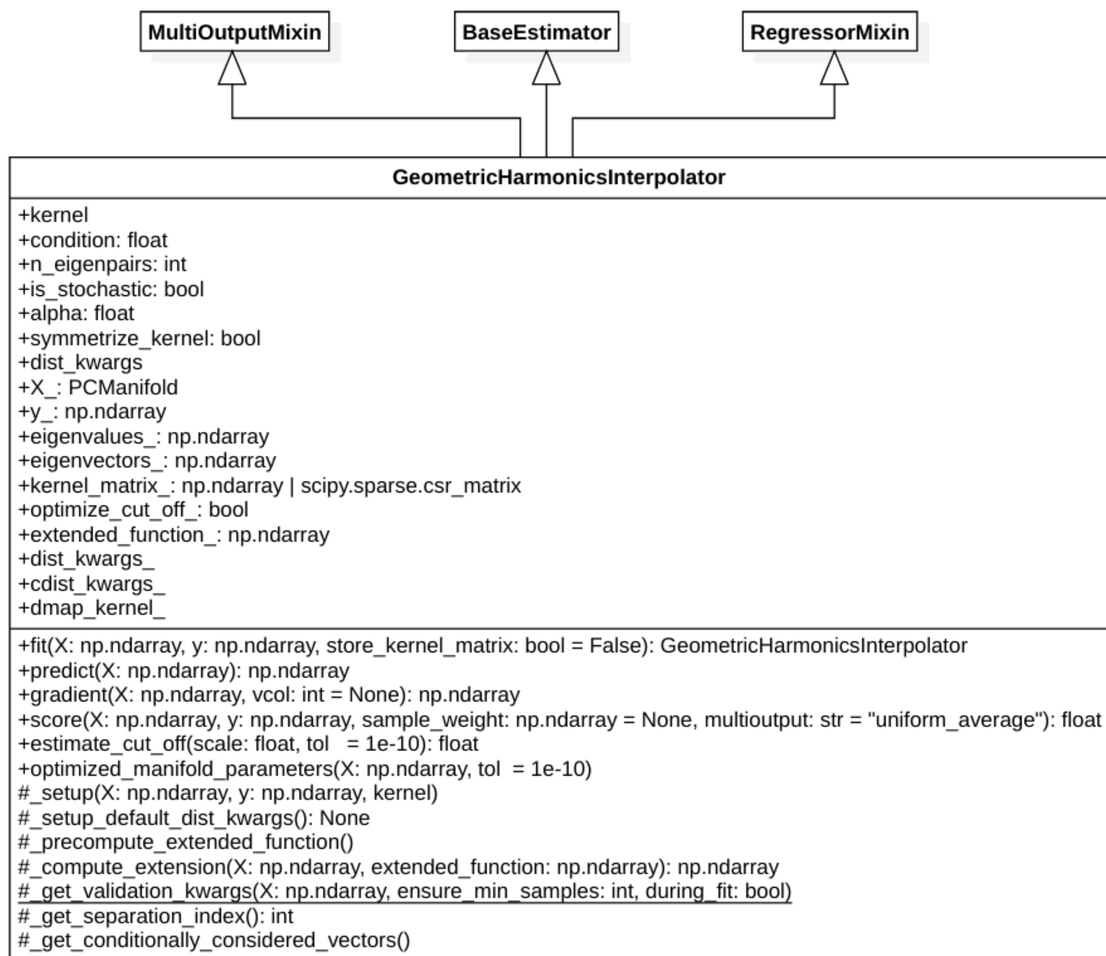


Figure 3.5: Class diagram of of the optimized `GeometricHarmonicsInterpolator`

Parameters

For the initialization parameters, one important argument was missing, namely the condition parameter defined as $\frac{1}{\delta}$ in Algorithm 1. This parameter is necessary to calculate the projection $P_\delta f$ which is also skipped in the old implementation [7, 15]. All other parameters did not change in naming or meaning.

Parameter	Description	Default
kernel	Kernel to describe proximity between points.	None
condition	Number to bound the condition of the extension from above.	50
n_eigenpairs	Number of eigenpairs to compute from kernel matrix	10
is_stochastic	If True, the diffusion kernel matrix is normalized	False
alpha	Re-normalization parameter. "alpha=1" corrects the sampling density in the data as an artifact of the collection process.	1
symmetrize_kernel	If True, a conjugate transformation is performed if the current settings would lead to a non-symmetric kernel matrix.	True
dist_kwargs	Keyword arguments passed to the internal distance matrix computation.	None

Table 3.4: Parameters of the optimized GeometricHarmonicsInterpolator.

Attributes

As for the old implementation, the attributes listed in Table 3.5 are estimated in the fitting process to later calculate predictions. The only changes here are on the one hand, the new naming of the function extended_function_ which was called _aux before. Since the name was incomprehensible, it already was marked as issue [15]. On the other hand, the accessibility of the attributes that were protected in the old implementation, namely the just mentioned extended_function_ and the two attributes cdist_kwargs_- and dmap_kernel_ are now publicly accessible. This change comes from the scikit-learn constraint mentioned in Section 3.1.2 which tells us to end all attributes estimated from the data with an underscore.

Attribute	Description
`X_`	Training data during fit of shape '(n_samples, n_features)'. Required to be stored to perform out-of-sample interpolations.
`y_`	Target function values of shape '(n_samples, n_targets)', can be multi-dimensional.
`eigenvalues_`	Eigenvalues of diffusion kernel in decreasing order.
`eigenvectors_`	Eigenvectors of the kernel matrix.
`kernel_matrix_`	Computed kernel matrix, stored if 'store_kernel_matrix=True' during fit.
`dist_kwargs_`	Actual keyword dictionary arguments passed to the internal distance matrix computation.
`extended_function_`	Extension function to perform out-of-sample interpolation.
`cdist_kwargs_`	Keys required for a component-wise kernel computation.
`dmap_kernel_`	The computed diffusion map kernel.

Table 3.5: All attributes of the optimized `GeometricHarmonicsInterpolator`.

Projection of data

As already mentioned, the most significant problem with this implementation is the missing projection $P_\delta f$ of f, since the projection is important to bound the condition of the algorithm [7]. One could say that the condition can be bounded by tweaking the amount of eigenpairs via the `n_eigenpairs` parameter, but this approach is error-prone. Especially for the Multiscale algorithm we would have to pick the correct amount of eigenpairs for each level. This problem can currently only be solved with the set

$$\{\psi_j, \lambda_j | \lambda_j \geq \delta * \lambda_0\}. \tag{3.2}$$

To include the projection of f, we first have to calculate the set (3.2) of considered eigenpairs. In the new implementation, this is realized by the method `_get_conditionally_considered_vectors()` which returns the considered eigenpairs and the considered scalar products. They are required to calculate the projection and extension regarding Equations (2.3) and (2.4). Furthermore, it returns the scalar products that are not considered for the given iteration, we call them "excluded scalar products". Those are necessary to later calculate the error $\|f - P_\delta f\|$ for the projection of each iteration in the multiscale approach. To compute these considered sets, we first calculate the largest index j of the set S_δ, defined in Equation (2.2), which is returned by the method `_get_separation_index()`. It first calculates the threshold determined by the largest

eigenvalue λ_0 and the condition $C = \frac{1}{\delta}$ as $\lambda_0 * \delta = \frac{\lambda_0}{C}$. With this threshold we yield a boolean array indicating whether an eigenvalue is chosen for projection $P_\delta f$ or not:

```
threshold_eval = self.eigenvalues_[0] / self.condition
dyadic_separation = self.eigenvalues_ >= threshold_eval
```

Then we simply take the index of the first value being `False` which is our desired separation index. This is done with the NumPy function `argmin` [12]. It returns the index of the smallest value. If the array contains some equal values which are at the same time the smallest values, it returns the index of the first smallest value. This is most of the time the case for our implementation and deliberated. But with this feature we also have to consider the case when all values are `True`, meaning every value of the array is equally "small". In this case we wrongly obtain `0` as separation index while the actual separation index should resemble the length of the `dyadic_separation` array. We can handle this error by checking the first index of the `dyadic_separation` array. If it yields `True`, we should assign the length of the array as separation index. Otherwise, we can continue with the already calculated index:

```
if separation_index == 0 and dyadic_separation[0]:
    separation_index = len(dyadic_separation)
```

With this separation index s we can obtain the considered elements. But to also split the scalar products we first have to calculate them as described in Equation (3.1). This calculation is already implemented in the old model. Thus, we just have to extract it from the old extension calculation and assign it to the variable `separation_index`. Now we can calculate all conditionally considered subsets necessary for the projection of f and the calculation of the prescribed error in the multiscale extension:

```
scalar_products = self.eigenvectors_.T @ self.y_
considered_scalar_products = scalar_products[:separation_index]
excluded_scalar_products = scalar_products[separation_index:]
considered_evals = self.eigenvalues_[:separation_index]
considered_evecs = self.eigenvectors_[:, :separation_index]
```

Finally, we can calculate the projection as described in Algorithm 1, but instead of

calculating it in a loop we again, fall back on the NumPy features [12].

$$
\begin{aligned}
P_\delta f &= \begin{bmatrix} \psi_0(x_0) & \psi_1(x_0) & \cdots & \psi_s(x_0) \\ \psi_0(x_1) & \psi_1(x_1) & \cdots & \psi_s(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_0(x_m) & \psi_1(x_m) & \cdots & \psi_s(x_m) \end{bmatrix} \begin{bmatrix} \langle f, \psi_0 \rangle \\ \langle f, \psi_1 \rangle \\ \vdots \\ \langle f, \psi_s \rangle \end{bmatrix} \\
&= \begin{bmatrix} \sum_{i=0}^{s} \psi_i(x_0) \langle f, \psi_i \rangle \\ \sum_{i=0}^{s} \psi_i(x_1) \langle f, \psi_i \rangle \\ \vdots \\ \sum_{i=0}^{s} \psi_i(x_m) \langle f, \psi_i \rangle \end{bmatrix} = \begin{bmatrix} P_\delta f(x_0) \\ P_\delta f(x_1) \\ \vdots \\ P_\delta f(x_m) \end{bmatrix}
\end{aligned} \tag{3.3}
$$

Regarding Equation (3.3) we can calculate the projection of f by calculating the matrix multiplication of `considered_evecs` and `considered_scalar_products`:

```
projection = considered_evecs @ considered_scalar_products
```

Last but not least, with the projection the function extension $\hat{f}$ can be computed as in the old implementation, but this time with the considered eigenpairs and the scalar products of the projection $P_\delta f$ instead of f:

```
projected_scalar_products = considered_evecs.T @ projection
extended_function = (
    mat_dot_diagmat(considered_evecs, np.reciprocal(considered_evals))
    @ projected_scalar_products
)
```

Refactorings

Since the old software design does not meet all requirements of the scikit-learn API [4] constraints and more important, since it does not provide a sufficient API for the multiscale extension, some refactoring is necessary [15].

First, we take a look at the issues regarding scikit-learn API. One issue is the wrong naming of attributes estimated in the fitting process, but this was already fixed in Section 3.2.2. Another issue is, that the method `_get_tags()` is already implemented by the parent class `BaseEstimator` and therefore, we should not override it. Thus, we can delete it from the model. Furthermore, the method `_more_tags()` should only return additional tags which are not defined by parent methods or tags which should override a value. Since the tags `{"requires_y": True, "multioutput": True,}` are already defined by the base classes `MultiOutputMixin` and `RegressorMixin` with these values,

it is unnecessary to declare them again. Additionally, we will calculate the score as $1 \div (1 + RMSE)$ where $RMSE$ is the root mean squared error calculated as before, but not negated. The best result reachable is then one which satisfies the scikit-learn API constraint "higher score is better" [4]. Since this score returns better results better the `"poor_score"` tag can be deleted and hence, we can safely delete the whole method.

To make the `GeometricHarmonicsInterpolator` suitable for the multiscale extension, we extract some functionality into reusable methods. The data validation, the construction of the diffusion map kernel, the point cloud and the computation of the kernel and its eigenvalues are moved from the `fit` method to the separate protected method called `_setup`, which is called in the `fit` method. The setup sequence returns the variables `kernel_matrix_` and `basis_change_` matrix to allow the storage in the fitting process of the kernel matrix like in the old implementation [15]. Furthermore, we extract the computation of the extension in the `predict` method described in Section 3.2.1 to the method `_compute_extension` for better reusability. Last but not least, the function `_get_validation_kwargs` is now static since it does not depend on the instance state.

Automatic scale and cut-off selection

The last part of the software design optimization is the automatic selection of an appropriate scale and cut-off. Even though, the original algorithm does not consider the automated selection of the scale, selecting the right scale is crucial for good results in both interpolators, the `GeometricHarmonicsInterpolator` and the `MultiscaleGeometricHarmonicsInterpolator`. Furthermore, the right cut-off for the `dmap_kernel_` is important for a faster computation and for saving large amounts of storage. The old implementation sets the cut-off to `np.inf` per default, which means the kernel matrix computes and stores all $m \times n$ entries in the memory [15].

One can see in the `optimize_parameters` method of the `PCManifold` class that these two parameters scale and cut-off of the kernel and point cloud are closely related to each other. This function first estimates the cut-off and subsequently, estimates the scale based on the cut-off. The cut-off is computed in the function `estimate_cutoff` [15] which is located in a collection of functions for estimators. In short, this function computes the distance matrix of the point cloud and returns the maximum distance to the k-th nearest neighbors where k is per default 10. To calculate epsilon, we first assume a fixed number t, which corresponds to the variable `tol`, greater than or equal to the Gaussian kernel (2.1):

$$t \geq exp\left(-\frac{\|x-y\|^2}{\varepsilon}\right) \tag{3.4}$$

Then we approximate the scale by solving Equation (3.4) for ε and simply setting it

equal:

$$\varepsilon = -\frac{\|x-y\|^2}{\log(t)} \tag{3.5}$$

In this function the cut-off corresponds to $\|x-y\|$. Another datafold function `estimate_scale` [15] of the function collection for estimators computes the scale with Equation (3.5) by inserting the previously calculated cut-off and a fixed t which is by default `1e-8`. We encapsulate the optimization method of the `PCManifold` in the public method `optimized_manifold_parameters` which first validates the passed array `X` and then constructs a `PCManifold` with this data and a default Gaussian kernel. On this `PCManifold` we then call the optimization method with the passed `tol` and return the results. In this way, the function may also be used by the user to pick an appropriate scale and cut-off by himself. We also apply our optimization method in the `_setup` method when the `kernel` parameter equals `None` but only to estimate the scale rather than both parameters, since we have to define another method which adjusts the cut-off to the scale in the multiscale approach.

We can compute an estimation of the cut-off $c = \|x-y\|$ from a given scale by solving Equation (3.5) for c:

$$c = \sqrt{-\log(t) * \varepsilon}$$

This equation is then implemented as the function `estimate_cut_off` which accepts a `scale` and a `tol` as arguments:

```
def estimate_cut_off(self, scale: float, tol=1e-10) -> float:
    return float(np.sqrt(-np.log(tol) * scale))
```

With this method, we then automatically select an appropriate cut-off in the `_setup` method if the user did not provide one in `dist_kwargs`. In the function `_setup_default_dist_kwargs()` the attribute `optimize_cut_off_` is assigned `True` if either `dist_kwargs` is `None` or the key `"cut_off"` is not defined on `dist_kwargs`.

3.3 Implementation of Multiscale Geometric Harmonics

As the single-scale Geometric Harmonics model now meets the requirements to design a well-structured multiscale extension, we elaborate the new software design for the multiscale extension algorithm. In the first Section 3.3.1 we document the old unfinished software design and then provide a reworked model in Section 3.3.2.

3.3.1 The current state

Again, the current state of the code is documented as for the single-scale approach. The multiscale extension algorithm is implemented as the `MultiscaleGeometricHarmonicsInterpolator` class which inherits from the `GeometricHarmonicsInterpolator` [15]. Since this implementation is incomplete, it is marked with the `@NotImplementedError` decorator. The class diagram of the multiscale extension is depicted in Figure 3.6.

First we shortly analyze the fitting process. The current `MultiscaleGeometricHarmonicsInterpolator` [15] more or less implements the basic Geometric Harmonic algorithm. In this case with considering a condition and thus, it is using the projection of f. Another difference to the old single-scale implementation is that it repeats the algorithm in a loop with a shrinking scale and the corresponding Gaussian kernel. The initial scale is reduced by two in each iteration and the termination condition depends on the levels prescribed error. That means, that the algorithm reduces the scale until the prescribed error is small enough and then only the last scale is used to calculate one extension function $\hat{f}$. Thus, it is not a multiscale algorithm but single-scale. Even though it computes the Gaussian kernel and all necessary variables for an extension function at every scale. Another issue is the calculation of eigenpairs with the `eigsh` function, which is perfect for sparse hermitian matrices but with the eigensolver methods used in the `GeometricHarmonicsInterpolator` implementation we always use the best matching eigensolver [15].

One more important observation of the old implementation is, that it does not implement an own `predict` method as typical for a scikit-learn regressor [4] but a `__call__` method, which lets one handle an instance like calling a function. It contains the procedure which should be provided by a `predict` method. Last but not least, the algorithm implements an own `score` method but actually only calls the parent's method, namely the `score` method of the `GeometricHarmonicsInterpolator`.

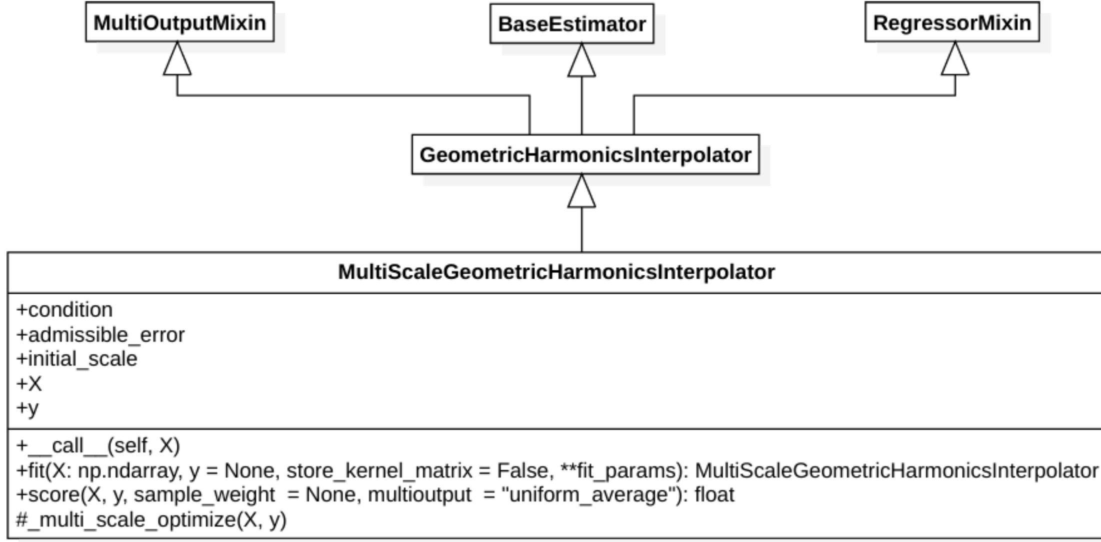


Figure 3.6: Class diagram of of `MultiscaleGeometricHarmonicsInterpolator`.

Parameters

The `MultiscaleGeometricHarmonicsInterpolator` is initialized with the same parameters as the `GeometricHarmonicsInterpolator` class accepts listed in Table 3.2 on the one hand and on the other hand, with the three additional parameters listed in Table 3.6. The necessity of the three parameters was already explained in the previous subsection.

Parameter	Description	Default
`initial_scale`	The scale to begin with.	`1.0`
`condition`	The number $C = \frac{1}{\delta}$ to bound the condition of the algorithm from above.	`1.0`
`admissible_error`	The maximal acceptable prescribed error.	`1.0`

Table 3.6: Additional parameters of the `MultiscaleGeometricHarmonicsInterpolator`.

Attributes

The only additional attributes of the multiscale approach are `X` and `y`. First, they do not fulfill the scikit-learn API constraint of the trailing underscore. Second, they would be unnecessary because the old `GeometricHarmonicsInterpolator` is equipped with the correctly named attributes `X_` and `y_` [15]. Since it extends the `GeometricHarmonicsIn-`

terpolator, it naturally inherits the attributes listed in Table 3.3. Most of them are not used by the MultiscaleGeometricHarmonicsInterpolator.

3.3.2 The reworked model

For the multiscale approach, we have to completely rework the implementation as already implied by the @NotImplementedError decorator of the MultiscaleGeometricHarmonicsInterpolator class [15]. The UML class diagram of the reworked software design is depicted in Figure 3.7.

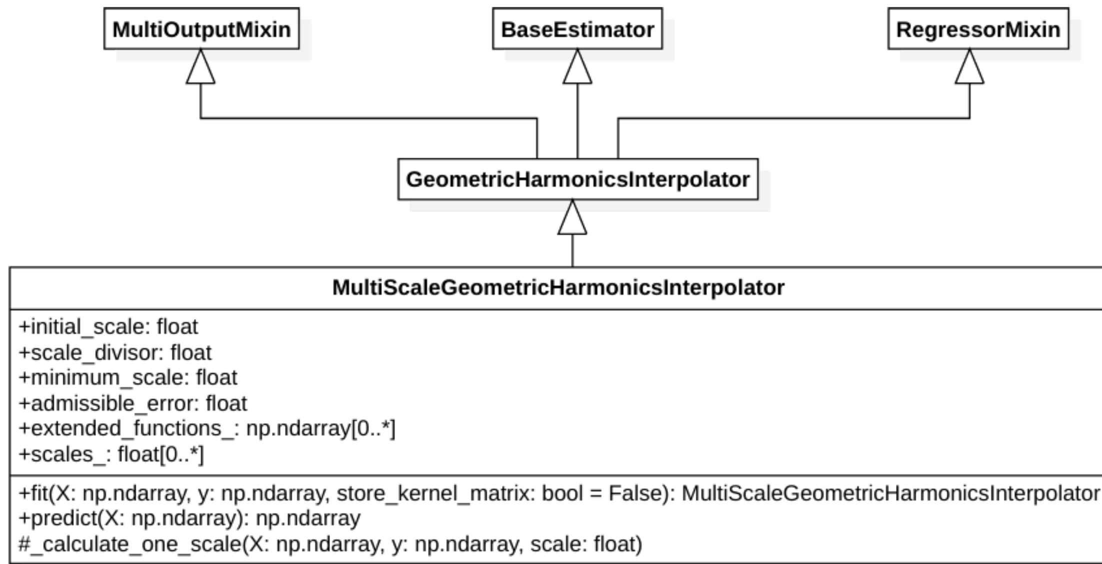


Figure 3.7: Class diagram of of the reworked MultiscaleGeometricHarmonicsInterpolator.

Parameters

For the new model, four additional parameters to the parameters of the GeometricHarmonicsInterpolator defined in Table 3.4 are necessary. Two of them are already declared in the incomplete implementation, namely the arguments initial_scale and admissible_error. The initial_scale is per default None. That tells the algorithm to pick an appropriate initial scale automatically with the procedure introduced in Section 3.2.2 if none is defined by the user. One of the new parameters is the scale_divisor corresponding to μ in Algorithm 2 which defines the size of the scale reduction per iteration. It must be larger than one for the algorithm to converge. The second

new parameter is `minimum_scale` which tells the algorithm when to stop even though the prescribed error is not smaller than `admissible_error`. On the one hand, this is important for the implementation, since otherwise, the construction of a kernel with a scale smaller than `minimum_scale` its default value would throw an error [15]. On the other hand, with this argument the user can adjust the smallest scale the algorithm should consider. This value may of course not be smaller than the default value. All additional attributes of the reworked `MultiscaleGeometricHarmonicsInterpolator` class are listed in Table 3.7.

Parameter	**Description**	**Default**
`initial_scale`	The scale we begin with.	`None`
`scale_divisor`	The number the scale is divided by. It must be larger than one.	`2.0`
`minimum_scale`	The minimum scale which should be reached. It must be larger than the default value, otherwise the kernel would throw an error.	`2.220446049250313e-16`
`admissible_error`	The maximal acceptable prescribed error.	`1.0`

Table 3.7: Additional parameters of the reworked `MultiscaleGeometricHarmonicsInterpolator`.

Attributes

While the incomplete `MultiscaleGeometricHarmonicsInterpolator` does not define extra meaningful attributes, we have to specify two new attributes listed in Table 3.8 which are necessary for the multiscale approach, additionally to the attributes of the optimized `GeometricHarmonicsInterpolator` from Table 3.5. Both extra attributes are required by the adjusted `predict` procedure which is later introduced.

Attribute	**Description**
`extended_functions_`	An array of each levels extension function.
scales_	An array of each levels scale.

Table 3.8: Additional attributes of the reworked `MultiscaleGeometricHarmonicsInterpolator`.

Fitting the parameters

The fitting process of Multiscale Geometric Harmonics corresponds to the precomputation phase of Algorithm 2 line 1 to 8. First, we have to set up the necessary variables. The attributes `extended_functions_` and scales_, required for the prediction explained in Section 3.3.2, get initialized with empty arrays. Subsequently, we call the `_setup_default_dist_kwargs()` method in which it is also decided whether to use a predefined value if one is passed, or to compute an automatically selected cut-off in each iteration as described in Section 3.2.2. Furthermore, we check the values of the parameters `scale_divisor` and `minimum_scale`. As specified in Table 3.7, the `scale_divisor` must be larger than one and the `minimum_scale` may not be smaller than the default value. Thus, we throw an exception if the constraints are not satisfied. Another parameter to be checked is the initial scale. If it equals `None`, we automatically pick an appropriate initial scale by passing the argument `X` to the `optimized_manifold_parameters` method of its improved parent class `GeometricHarmonicsInterpolator`. The respective scale is then assigned to the variable `scale`:

```
if self.initial_scale is None:
    (_, opt_scale) = self.optimized_manifold_parameters(X)
    scale = opt_scale
else:
    scale = self.initial_scale
```

We need the extra internal variable `scale`, since we will reduce it in every iteration of the algorithm and do not want to change the value of the initialization parameter `initial_scale`.

The next step is to start a loop which either stops when the calculated error $\|f - P_\delta f\|$ is smaller than the given admissible error `admissible_error` as described in Algorithm 2 or when the scale for the next iteration $\varepsilon_{j+1} = \varepsilon_j * \frac{1}{\mu}$ is smaller than the value of `minimum_scale`. Both cases are handled with a break condition, since we do not initialize the error before entering the loop. And since the initial scale may be smaller than a manually selected minimum scale, the algorithm should at least calculate one iteration instead of preventing the computation completely. Obviously, before checking the scale of the next iteration we calculate it. The main computation is done by the protected function `_compute_one_scale` which performs the single-scale geometric harmonics for a given scale and returns the vector $\overline{f}$ in line 5 of Algorithm 2 and the norm of it which corresponds to the error. The obtained values $\overline{f}$ are then passed to the function call in the next iteration. The whole loop then looks as follows:

```
while True:
    (y, error) = self._calculate_one_scale(X, y, scale)
    if error <= self.admissible_error:
        break
    scale = scale / self.scale_divisor
    if scale < self.minimum_scale:
        break
```

In the function `_compute_one_scale` we first construct a Gaussian kernel with the scale of the current iteration and pass it together with `X_` and `y` to the `_setup` function of the adjusted parent class `GeometricHarmonicsInterpolator`. This function call validates the data sets all attributes necessary to compute the extension of one scale as for the single-scale algorithm. Now it is time to get involved with the newly returned variables of the adjusted `_precompute_extended_function` function which is called next. With the variables `considered_evecs` and `considered_scalar_products` we again, compute the projection as described in Equation (3.3) which is required to calculate the target values $\overline{f}$ as difference of the validated `self.y_` and the projection for the next iteration:

```
projection = considered_evecs @ considered_scalar_products
y = self.y_ - projection.reshape(self.y_.shape)
```

Of course, one could now use `y` to calculate the error as $\|\overline{f}\| = \|f - P_\delta f\|$ [7, 6], but the better option is to calculate the norm of `excluded_scalar_products`, since this is roughly equal to the norm of the new $\overline{f}$ but consists of fewer values. Hence, the computation is less expensive. If the array `excluded_scalar_products` does not contain any value, the error is assigned to zero:

```
if len(excluded_scalar_products) > 0:
    error = np.sqrt(np.mean(np.abs(y) ** 2))
else:
    error = 0
```

Lastly, we return the calculated `y` and `error` and continue the iteration with these values until we reach one of the break conditions.

Predicting of data

To complete the algorithm the projection of the data itself is required. We use exactly the process defined as prediction phase in Algorithm 2 line 9 to 12. The precomputed

vectors `extended_functions_` correspond to the $\hat{f}_j$ and the scalars scales_ are the scales ε_j for the corresponding level j. The amount of levels is thus, given by the length of the arrays `extended_functions_` and scales_. For each level j we do not explicitly construct a new Gaussian kernel, but rather substitute the epsilon of the stored `dmap_kernel_`-with the scale ε_j in each iteration:

```
self.dmap_kernel_.internal_kernel.epsilon = self.scales_[j]
```

When we then compute the required kernel matrix of `self.X_`, the exchanged scale is applied. The matrix computation is done in `_compute_extension` provided by the parent class `GeometricHarmonicsInterpolator`. It accepts T and $\hat{f}_j$ as inputs for which the extension should be calculated and returns it:

```
extension = self._compute_extension(X, self.extended_functions_[j])
```

The computed extension for level j is then added to the accumulator `extension_acc` which corresponds to the actual extension of the multiscale algorithm. In the loop the variable is initialized with the NumPy function `np.zeros` [12] of the shape of the first calculated extension. After exiting the loop, `predict` returns the extended function F as `extension_acc`.

3.4 Implementation of Laplacian Pyramids

The last procedure we consider is Laplacian Pyramids which is introduced in Section 2.2. Those are also multiscale approaches which are for instance applicable in high-dimensional data analysis [17]. In Section 3.4.1 we outline and analyze the existing implementation.

Parameters

The parameters of this implementation [15] listed in Table 3.9 already meet the requirements of the Laplacian Pyramids procedure described in Section 2.2.1. Those are the initial scale `initial_epsilon`, `mu` as scale divisor and `residual_tol` which is comparable to `admissible_error` of the `MultiscaleGeometricHarmonicsInterpolator`. Since this model also implements Auto-adaptive Laplacian Pyramids, it provides the boolean parameter `auto_adaptive` to tell whether to use the auto-adaptive strategy. In this case, it would ignore the value provided in `residual_tol`. Either `residual_tol` must contain a value or `auto_adaptive` must be true. Otherwise, the estimator throws an error.

Parameter	Description	Default
`initial_epsilon`	The scale we begin with.	`10`
`mu`	The number the scale is divided by. Must be larger than one.	`2.0`
`residual_tol`	The tolerance at which the iteration is terminated. If "auto_adaptive=False" a value must be provided.	`None`
`auto_adaptive`	If True, decreasing the kernel scale terminates based on LOOCV estimation in each iteration.	`False`
`alpha`	A parameter handled to the diffusion maps kernel that in internally used.	`0`

Table 3.9: Parameters of the `LaplacianPyramidsInterpolator`.

Attributes

The attributes listed in 3.10 are all estimated to later calculate the prediction. The attribute `_level_tracker` is a dictionary storing all information of the considered levels. To later assign the accumulator variable F in Algorithm 3 correctly, we remember the number of target functions in `n_targets_`.

Attribute	Description
`X_`	Training data during fit of shape '(n_samples, n_features)'. Required to be stored to perform out-of-sample interpolations.
`n_targets_`	The number of target functions during fit.
`_level_tracker`	A dictionary containing the information of each level.

Table 3.10: Attributes of the `LaplacianPyramidsInterpolator`.

3.4.1 The current state

This algorithm is already well implemented in the LaplacianPyramidsInterpolator [15], even though mingled with the Auto-adaptive approach [10]. The UML class diagram of the interpolator is depicted in Figure 3.8. As already mentioned, the class LaplacianPyramidsInterpolator does not only contain Algorithm 3 of basic Laplacian Pyramids, but also the Auto-adaptive Laplacian Pyramids procedure described in Algorithm 4. The fit procedure corresponds to the precomputation phase in the pseudocode algorithms. It first checks the initialization parameters and the input X and y of the fit method. If for example auto_adaptive is False and residual_tol is None, the algorithm throws an exception since none of both algorithms can then be applied. After the validation we set up a empty dictionary for _level_tracker. Lastly, we assign n_targets_ to y.shape([1]) and proceed with the Laplacian Pyramids algorithm. Overall, the algorithm is implemented similar to the proposed Algorithm 3 and Algorithm 4. It manages an internal state for the levels via the enum type called _LoopCond and provides many methods to handle it. Overall, the algorithm is fine, but the code structure is confusing especially since the LaplacianPyramidsInterpolator contains both algorithms at once. buhmann2000radial

The prediction also first validates the passed X and then creates the initial F with X and n_targets:

```
y_hat = np.zeros([X.shape[0], self.n_targets_])
```

Then in the loop, we compute the kernel matrix with stored kernel of the corresponding level and the distance matrix returned by the method _distance_matrix. We then compute the new approximation for the active indices with the kernel_matrix and the target values:

```
active_indices = level_content["active_indices"]
y_hat[:, active_indices] += kernel_matrix @ level_content["target_values"]
```

3.4.2 Proposed enhancements

The first enhancement should be the abstraction of the Auto-adaptive Laplacian Pyramids procedure as own estimator which inherits from LaplacianPyramidsInterpolator. This would make the code much clearer and better structured. Thus, it is also less error-prone when everything is better encapsulated. Furthermore, one could then provide an additional class for the Local Auto-adaptive Laplacian Pyramids [10]. In addition, one should implement the automatic initial scale selection and one may add the threshold of the maximum amount of iterations as described for the Auto-adaptive

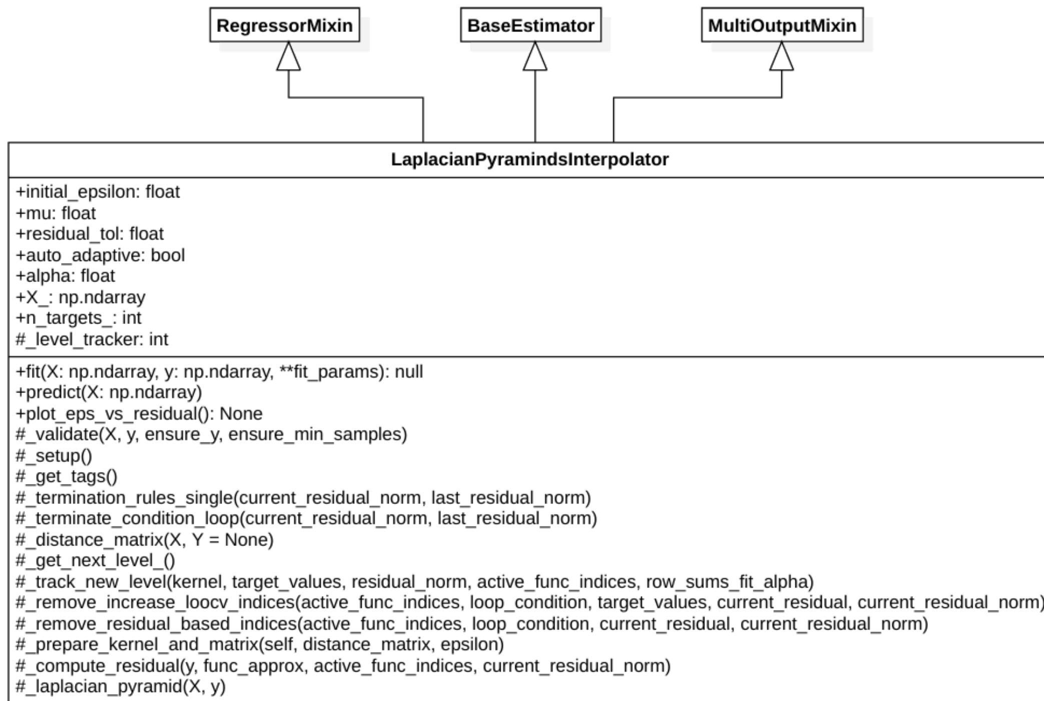


Figure 3.8: Class diagram of of the `LaplacianPyramidsInterpolator`.

Laplacian Pyramids [10]. However, the estimator already shows good results as one can see in Section 3.5. Another optimization could be the implementation of the optimized cut-off that we introduced for Geometric Harmonics in Section 3.2.2. This could allow the estimation with larger samples. The scale should also be checked in every iteration in the fitting process, since the algorithm throws an error when the scale gets too small [15]. Last but not least, the `LaplacianPyramidsInterpolator` should implement the same score calculation as `GeometricHarmonicsInterpolator` for better comparability. This would be $1 \div (1 + RMSE)$ with the Root Mean Squared Error [4, 15].

3.5 Testing and Demonstrations

In the first Section 3.5.1 the conducted tests are briefly outlined, in Section 3.5.2 a typical "Geometric Harmonics" example is introduced, then in Section 3.5.3 we observe an example of image interpolation of the and in the last Section 3.5.4 we reproduce the results of a synthetic example with the `LaplacianPyramidsInterpolator`.

3.5.1 Conducted tests

The tests are conducted as python unit tests in the datafold [15] testing environment. Most of the tests, already existing and newly added tests consist of demonstrations like those presented in the next three sections. They assert, whether the predictions fulfill predefined constraints regarding the score of the estimator. Additionally, tests to validate the satisfaction of important constraints are added to the newly introduced `MultiScaleGeometricHarmonicsTest` testing class. One important test for every estimator is the `check_estimator` function provided by the scikit-learn API [4]. It goes through all important estimator tests regarding fitting and predictions depending on the tags returned by the respective `_get_tags` method of the tested estimator. Since this test executes successfully for the new software design of the `MultiscaleGeometricHarmonicsInterpolator`, the class can be called a valid estimator. Furthermore, this test also succeeds for the old and new `GeometricHarmonicsInterpolator` and for the `LaplacianPyramidsInterpolator`. Other old tests were adapted to agree on the new implementations. Even though, some important tests are provided and old tests were adjusted, additional tests should be considered one time.

3.5.2 Unit circle to the plane

The first example, the unit circle to the plane is prevalent as demonstration case in the papers on geometric harmonics [6, 23, 14, 8, 7] but most of the time, the circumstances are not described very well. This results in a variety of extrapolations with diverse appearances. Therefore, we want to reproduce the most convincing results and then summarize the setup. Also, we consider the results on the actual two-dimensional interpolation of the different observed frequencies. We will demonstrate this with Multiscale Geometric Harmonics on the one hand, and on the other hand, we will also apply basic Laplacian Pyramids to compare both procedures directly [6]. All charts are created with the data visualization tool Plotly [13].

First, we set up the demonstration data. The train data consists of 100 points uniformly distributed in the interval $[0, 2\pi]$ which correlates to the circumference of the unit circle. Each point x is then assigned to its coordinates $(\sin(x), \cos(x))$. The corresponding function value of the point x is $\cos(f\theta)$ with frequency f. We consider the frequencies $1, 2, 4$ and 8 and fit and predict each separately. The `MultiscaleGeometricHarmonicsInterpolator` is initialized with the maximum number of eigenpairs, in this case 99, a condition of 50 and with `dist_kwargs={"cut_off": np.inf}`, such that we do not optimize the cut-off. Since `LaplacianPyramidsInterpolator` does not implement this feature, the results would not be comparable. Both estimators are initialized with the initial scale 1, scale divisor 2 and an admissible error of `1e-10`.

Next, the test data in the three-dimensional case is a matrix arranged in the interval $[-3, 3]$ with a distance of 0.01 to another point in each dimension x and y. We then correlate each point with another, such that we have an array of all coordinate pairs. After fitting the data, the prediction results in Figure 3.9 for Multiscale Geometric Harmonics and in Figure 3.10 for Laplacian Pyramids. The results show very good conformity with the visualizations presented by Coifman and S. Lafon [7] in the case of Multiscale Geometric Harmonics, and by Chiavazzo et al. [6] in the case of Laplacian Pyramids. It is interesting to see how different the extrapolations of the two algorithms look. While Geometric Harmonics seems to be very local, Laplacian Pyramids appear very diffusing.

Lastly, we consider the two-dimensional case. The test data stays the same, we only adjust the train data such that we obtain an interpolation instead of an extrapolation. This time, it consists of 150 points uniformly distributed, again, in the interval $[0, 2\pi]$. We assign the coordinates to each point as for the training samples and then let the estimators predict the corresponding values. The resulting predictions calculated by the `MultiscaleGeometricHarmonicsInterpolator` are plotted in Figure 3.11 and the predictions of `LaplacianPyramidsInterpolator` are displayed in 3.12. Additionally, to each frequency, one can see the absolute difference between the prediction and the actual values in the plots. While the largest error of `MultiscaleGeometricHarmonicsInterpolator` is approximately `1.5e-10`, the smallest error of Laplacian Pyramids is roughly `1e-8`.

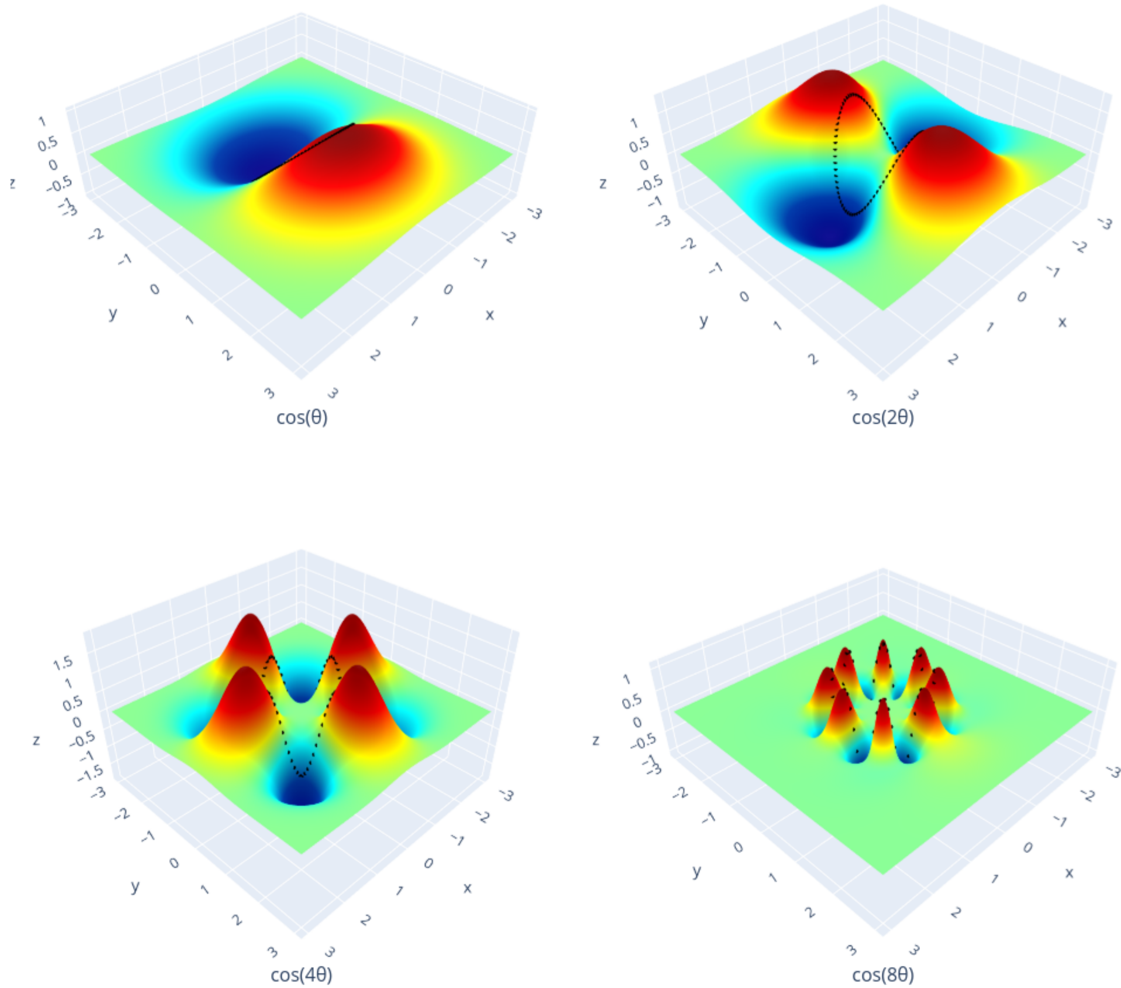


Figure 3.9: The unit circle in the plane for $\cos(\theta)$, $\cos(2\theta)$, $\cos(4\theta)$ and $\cos(8\theta)$ computed with the `MultiscaleGeometricHarmonicsInterpolator`. The interpolator is trained and initialized with `n=100` train points, `condition=50`, `initial_scale=1`, `scale_divisor=2` and `admissible_error=1e-10`.

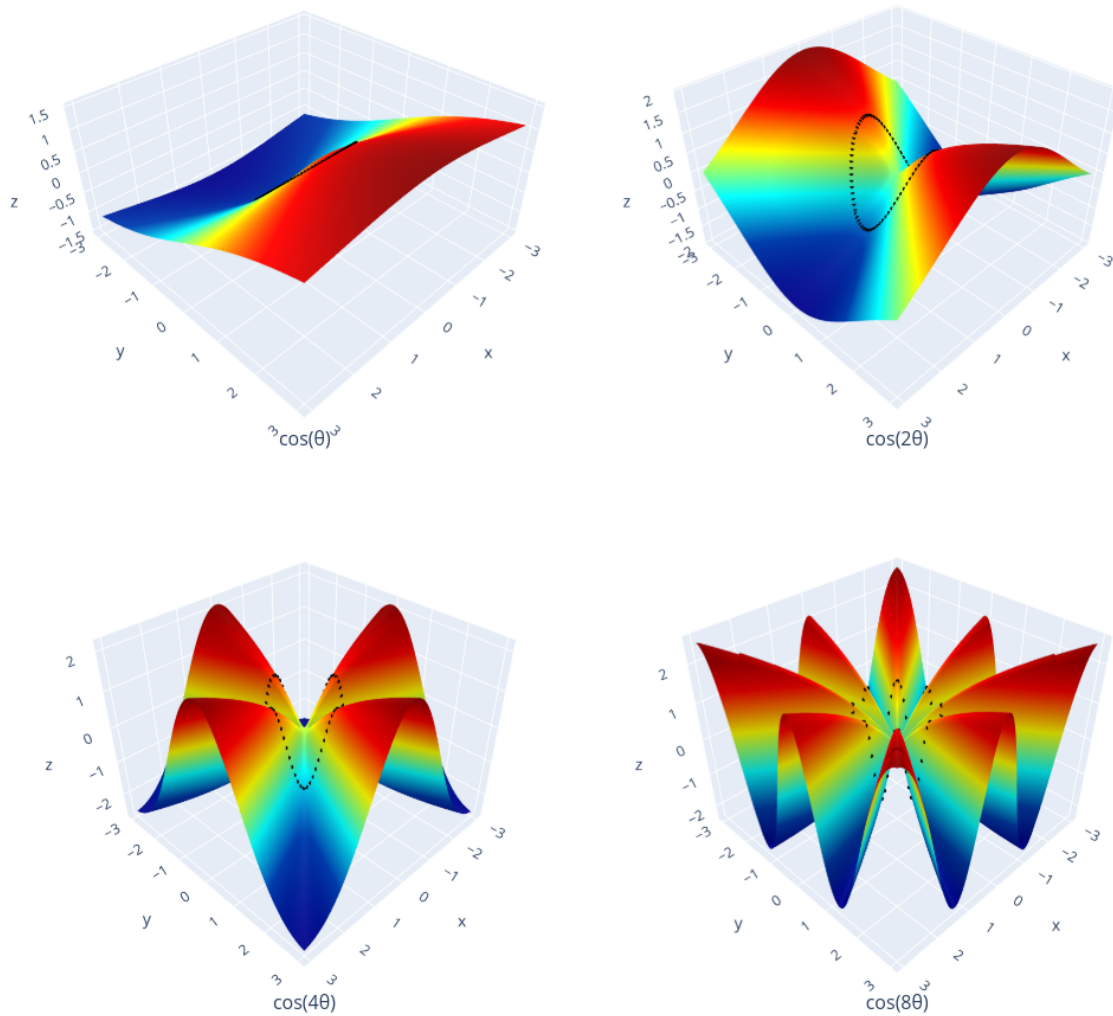


Figure 3.10: The unit circle in the plane for $\cos(\theta)$, $\cos(2\theta)$, $\cos(4\theta)$ and $\cos(8\theta)$ computed with the `LaplacianPyramidsInterpolator`. Each trained and initialized with `n=100` train points, `initial_epsilon=1`, `mu=2` and `residual_tol=1e-10`.

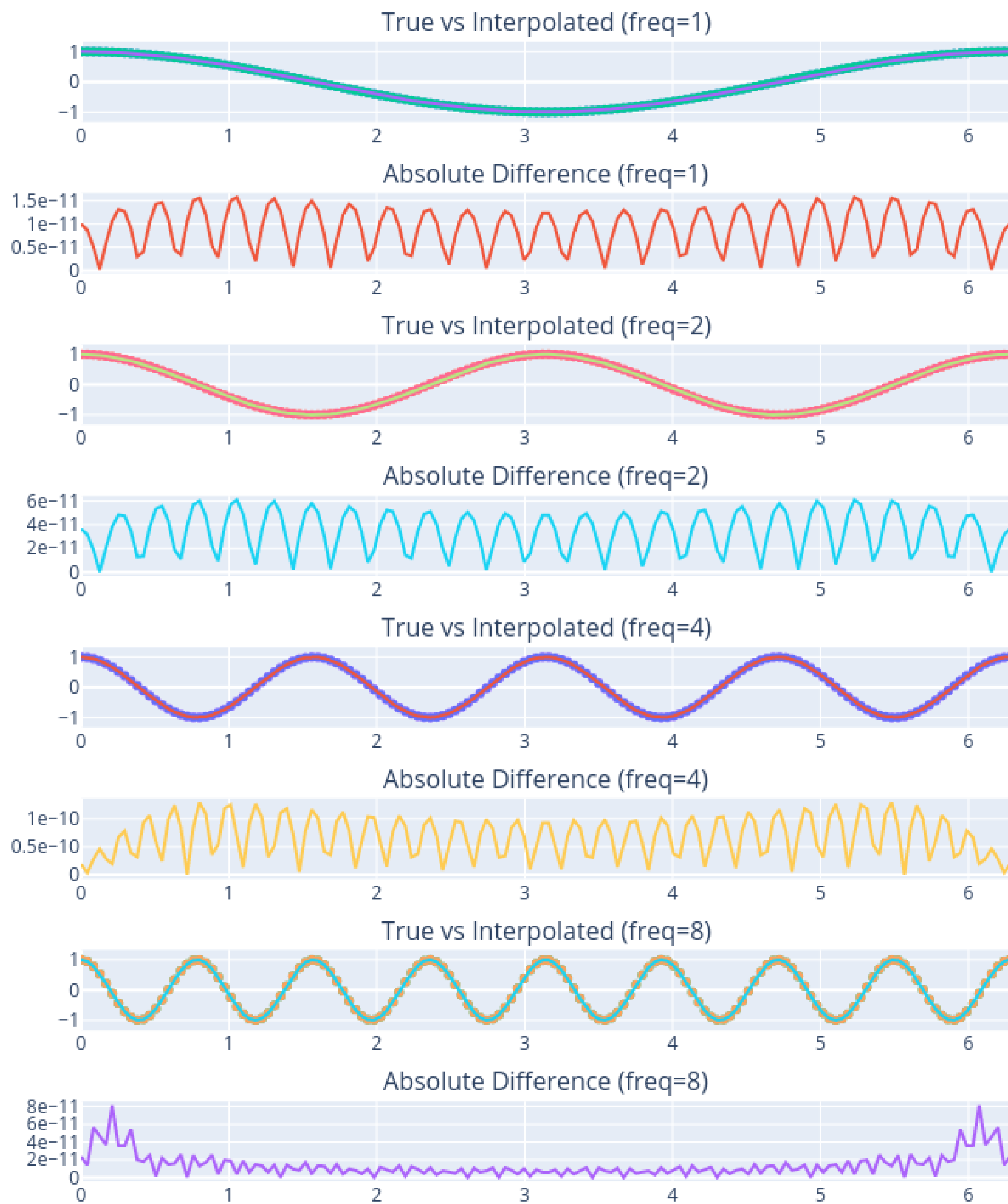


Figure 3.11: The actual interpolation of the cosinus functions in the interval $[0, 2\pi]$ with `MultiscaleGeometricHarmonicsInterpolator`. The interpolated values are displayed as dots.

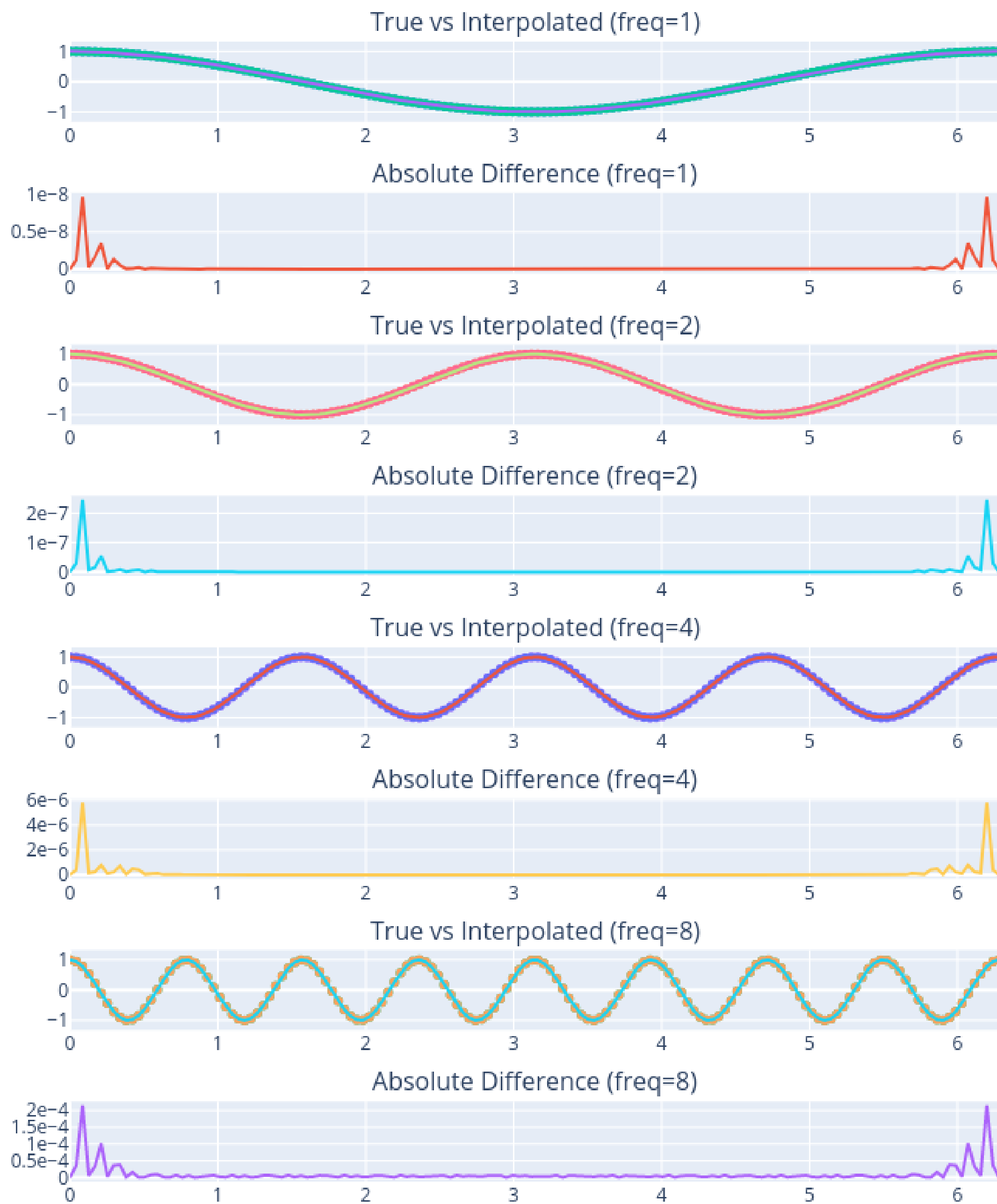


Figure 3.12: The actual interpolation of the cosinus functions in the interval $[0, 2\pi]$ with `LaplacianPyramidsInterpolator`. The interpolated values are displayed as dots.

3.5.3 Image interpolation

The next demonstration is literally classic for Laplacian Pyramids [5] and likewise very descriptive. In addition, we apply Multiscale Geometric Harmonics on this example for comparison reasons. One will once again, clearly see the difference between the two approaches. For the image data, we use the image processing Python package scikit-image [24]. The plots are again, generated with the data visualization tool Plotly [13]. Two problems appeared when setting up the demonstration, one is the computation time required by the eigensolver applied in the `MultiscaleGeometricHarmonicsInterpolator`. The original image has `256x256` pixels which corresponds to an array of 65536 values. For reasonable good results, one should use about two-thirds of the theoretically available eigenpairs which would be approximately 43690 but the eigensolver already requires minutes for ten thousand eigenpairs. The second problem is already fixed for the Geometric Harmonics estimator but still exists in the `LaplacianPyramidsInterpolator` class [15], namely the optimized cut-off when calculating the kernel matrix on larger data sets. When the cut-off is set to `np.inf` and we want to compute the kernel matrix with 65536^2 values the program crashes. To prevent crashes and to compute the results in reasonable time, we only use a fraction of the original dataset. Instead of `256x256` pixels the images are shrinked to `64x64` pixels. Thus, our train set consists of $64^2 = 4096$ pixel coordinates (x, y) and their corresponding pixel values as function values. The `MultiscaleGeometricHarmonicsInterpolator` is initialized with `n_eigenpairs=pixels ** 2 - 1`, `condition=5` and again for comparison reasons `dist_kwargs={"cut_off": np.inf}`. Both estimators have an initial scale of 2, $\mu = 2$ and an admissible error of `1e-10`. The resulting levels of restoring the image for each estimator are shown in Figure 3.13 and Figure 3.14.

One can clearly see the difference between the images of the first iteration. While the image with Laplace Pyramids gets less and less blurry in Figure 3.14, with Multiscale Geometric Harmonics the frequencies get higher and higher in Figure 3.13. Even though, Geometric Harmonics needs fewer iterations, the difference in calculation time is enormous. In this setup, the Laplacian Pyramids interpolator is about 830 times faster with approximately one second fitting process time than the Multiscale Geometric Harmonics approach which required about 830 seconds.

In 3.15 the interpolation of the image of size `128x128` is generated for each estimator. Even though, they can interpolate the missing pixels, the prediction is not very good in comparison with the original image. Obviously, the finer scales are missing in the interpolated versions, since the smaller image used to fit the estimators did not contain the information for this scale.

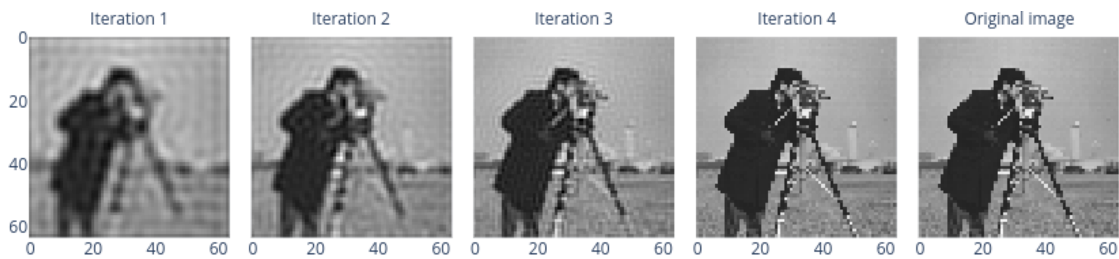


Figure 3.13: All four levels of the `MultiscaleGeometricHarmonicsInterpolator` predicting the train image beginning with the coarsest scale.

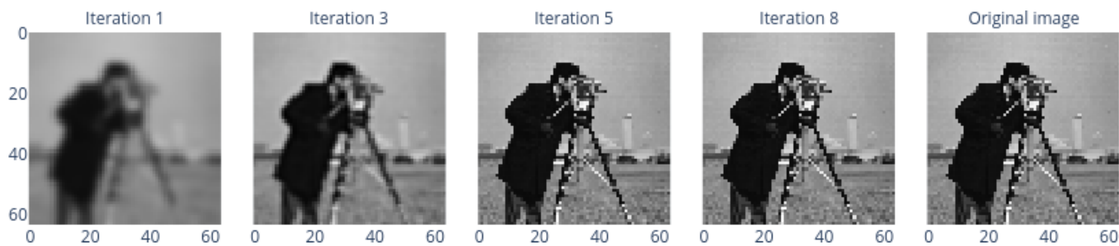


Figure 3.14: The levels 1, 3, 5 and 8 of the `LaplacianPyramidsInterpolator` predicting the train image beginning with the coarsest scale.

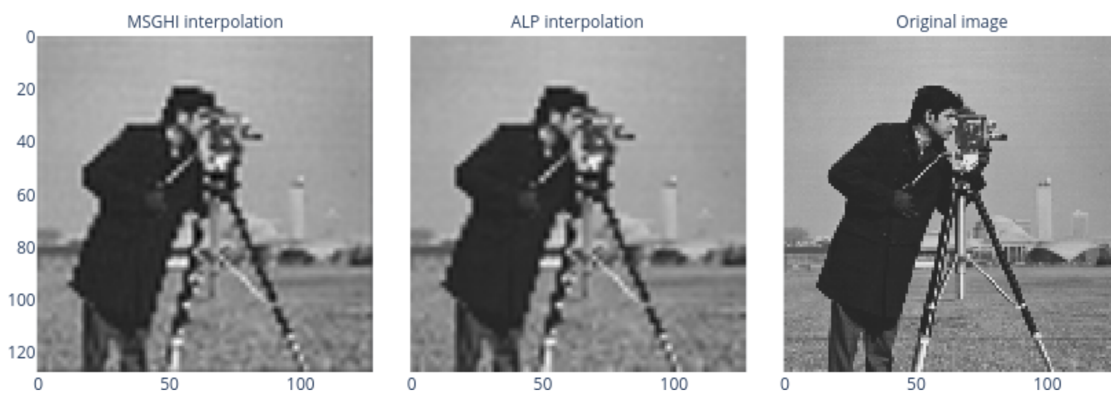


Figure 3.15: The generated interpolations with the `MultiscaleGeometricHarmonicsInterpolator` and the `LaplacianPyramidsInterpolator`.

3.5.4 Synthetic example

The last demonstration is a reproduction of the synthetic example proposed by Fernández et al. [10]. Since this example is originally performed with the Auto-adaptive Laplacian Pyramids procedure, and we only want to compare the actual datafold implementation [15] with the proposal, we only demonstrate this for the `LaplacianPyramidsInterpolator`. We show that the implementation meets the expectations. The plots are, again, generated with Plotly for visualization purposes [13]. We consider the sample with 4000 points uniformly distributed in the interval $[0, 10\pi]$. The function for our target values is

$$f = \sin(x) + \sin(3x) I_2(x) + 0.25 \sin(9x) I_3(x) + \varepsilon.$$

The indicator functions I_2 and I_3 represent the intervals $(10\frac{\pi}{3}, 10\pi]$ and $(10\frac{2\pi}{3}, 10\pi]$. The noise $\varepsilon \sim \mathcal{U}([-\delta, \delta])$ is uniformly distributed [10]. We only consider the small noise example, thus $\delta = 0.1$ for our demonstration. The `LaplacianPyramidsInterpolator` is initialized with an initial scale of 10π, $\mu = 2$ and is passed `auto_adaptive=true` to use Auto-adaptive Laplacian Pyramids. The demonstration shows good results in Figure 3.16 compared to the results produced by Fernández et al. [10]. One can see how finer scales are covered in each iteration.

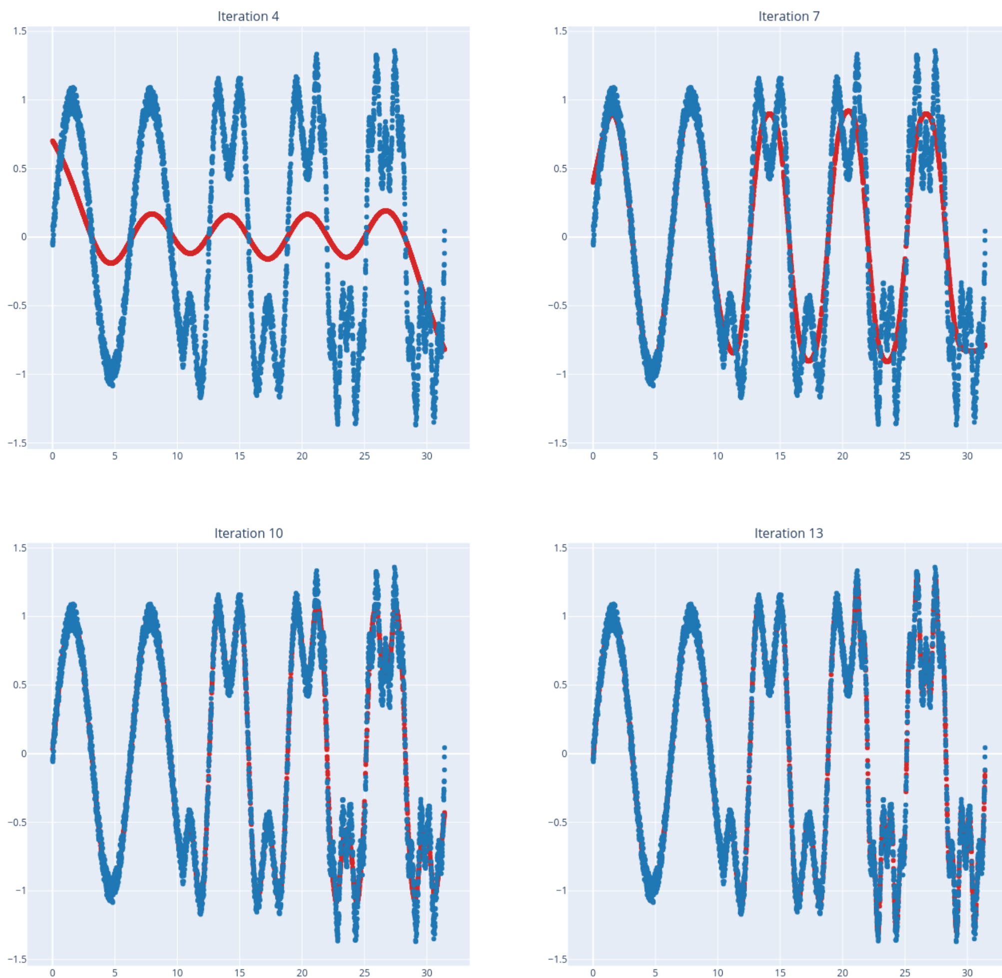


Figure 3.16: The iterations 4, 7, 10 and 13 of the `LaplacianPyramidsInterpolator` for the synthetic example.

4 Conclusion

In conclusion, we elaborated an optimization and rework of both Geometric Harmonic extensions. Furthermore, all out-of-sample extension schemes are thoroughly documented and the software structure was analyzed. In addition, we introduced a scale and cut-off optimization strategy for both Geometric Harmonics interpolators. Inconsistencies in the data structure were removed and the `GeometricHarmonicsInterpolator` class was adjusted, such that the multiscale estimator can properly extend it and reuse its functionality. We outlined some ideas on how to improve the `LaplacianPyramidsInterpolator` and showed missing features regarding the Auto-adaptive Laplacian Pyramids proposed by Fernández et al. [10]. Lastly, we summarized the conducted tests and demonstrated the multiscale approaches with details about the setup of the respective example and generated very good insightful results.

Even though, this thesis focsed on the optimization of the considered out-of-sample extensions and the implementation of optimizations, there is still much to do. Maybe one could find a better solution for calculating the right amount of eigenpairs in the Geometric Harmonics algorithms or even a better eigensolver for a faster computation in general. Also, a better way for stopping the Multiscale Geometric Harmonics iteration could be elaborated, which could work similar to the Auto-adaptive Laplacian Pyramids approach. The Laplacian Pyramids estimator has also still a lot to do, since in this paper optimizations were just proposed. As already mentioned, it is missing some originally intended key features for automatically selecting a scale and the maximal amount of iterations. Also, one should split the basic Laplacian Pyramids and Auto-adaptive Laplacian Pyramids for a better model structure. Based on that, one could implement the Local Auto-adaptive Laplacian Pyramids proposed by Fernández et al. [10] as another child class, which serves better for data that is not equally distributed and that has different density characteristics.

List of Figures

1.1 The levels of the MultiscaleGeometricHarmonicsInterpolator predicting the train image beginning with the coarsest scale. 1

3.1 Class diagram of the scikit-learn BaseEstimator 17
3.2 Class diagram of the scikit-learn MultiOutputMixin 18
3.3 Class diagram of the scikit-learn RegressorMixin 18
3.4 Class diagram of of the GeometricHarmonicsInterpolator. 19
3.5 Class diagram of of the optimized GeometricHarmonicsInterpolator . 28
3.6 Class diagram of of MultiscaleGeometricHarmonicsInterpolator. . . . 36
3.7 Class diagram of of the reworked MultiscaleGeometricHarmonicsInterpolator. 37
3.8 Class diagram of of the LaplacianPyramidsInterpolator. 44
3.9 The unit circle in the plane for $\cos(\theta)$, $\cos(2\theta)$, $\cos(4\theta)$ and $\cos(8\theta)$ computed with the MultiscaleGeometricHarmonicsInterpolator. The interpolator is trained and initialized with n=100 train points, condition=50, initial_scale=1, scale_divisor=2 and admissible_error=1e-10. 47
3.10 The unit circle in the plane for $\cos(\theta)$, $\cos(2\theta)$, $\cos(4\theta)$ and $\cos(8\theta)$ computed with the LaplacianPyramidsInterpolator. Each trained and initialized with n=100 train points, initial_epsilon=1, mu=2 and residual_tol=1e-10. 48
3.11 The actual interpolation of the cosinus functions in the interval $[0, 2\pi]$ with MultiscaleGeometricHarmonicsInterpolator. The interpolated values are displayed as dots. 49
3.12 The actual interpolation of the cosinus functions in the interval $[0, 2\pi]$ with LaplacianPyramidsInterpolator. The interpolated values are displayed as dots. 50
3.13 All four levels of the MultiscaleGeometricHarmonicsInterpolator predicting the train image beginning with the coarsest scale. 52
3.14 The levels 1, 3, 5 and 8 of the LaplacianPyramidsInterpolator predicting the train image beginning with the coarsest scale. 52

3.15 The generated interpolations with the `MultiscaleGeometricHarmonicsInterpolator` and the `LaplacianPyramidsInterpolator`. 52
3.16 The iterations 4, 7, 10 and 13 of the `LaplacianPyramidsInterpolator` for the synthetic example. 54

List of Tables

3.1 Arguments of the `fit` method. 16
3.2 Parameters of the `GeometricHarmonicsInterpolator` 20
3.3 All attributes of the `GeometricHarmonicsInterpolator`. 21
3.4 Parameters of the optimized `GeometricHarmonicsInterpolator`. 29
3.5 All attributes of the optimized `GeometricHarmonicsInterpolator`. . . . 30
3.6 Additional parameters of the `MultiscaleGeometricHarmonicsInterpolator`. 36
3.7 Additional parameters of the reworked `MultiscaleGeometricHarmonicsInterpolator`. 38
3.8 Additional attributes of the reworked `MultiscaleGeometricHarmonicsInterpolator`. 38
3.9 Parameters of the `LaplacianPyramidsInterpolator`. 42
3.10 Attributes of the `LaplacianPyramidsInterpolator`. 42

Bibliography

[1] S. Axler, P. Bourdon, and R. Wade. *Harmonic function theory*. Vol. 137. Springer Science & Business Media, 2013.

[2] Y. Bengio, J.-f. Paiement, P. Vincent, O. Delalleau, N. Roux, and M. Ouimet. "Out-of-sample extensions for lle, isomap, mds, eigenmaps, and spectral clustering." In: *Advances in neural information processing systems* 16 (2003), pp. 177–184.

[3] M. D. Buhmann. "Radial basis functions." In: *Acta numerica* 9 (2000), pp. 1–38.

[4] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux. "API design for machine learning software: experiences from the scikit-learn project." In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[5] P. J. Burt and E. H. Adelson. "The Laplacian pyramid as a compact image code." In: *Readings in computer vision*. Elsevier, 1987, pp. 671–679.

[6] E. Chiavazzo, C. W. Gear, C. J. Dsilva, N. Rabin, and I. G. Kevrekidis. "Reduced models in chemical kinetics via nonlinear data-mining." In: *Processes* 2.1 (2014), pp. 112–140.

[7] R. R. Coifman and S. Lafon. "Geometric harmonics: a novel tool for multiscale out-of-sample extension of empirical functions." In: *Applied and Computational Harmonic Analysis* 21.1 (2006), pp. 31–52.

[8] R. R. Coifman, S. Lafon, A. B. Lee, M. Maggioni, B. Nadler, F. Warner, and S. W. Zucker. "Geometric diffusions as a tool for harmonic analysis and structure definition of data: Diffusion maps." In: *Proceedings of the national academy of sciences* 102.21 (2005), pp. 7426–7431.

[9] M. N. Do and M. Vetterli. "Framing pyramids." In: *IEEE Transactions on Signal Processing* 51.9 (2003), pp. 2329–2342.

[10] Á. Fernández, N. Rabin, D. Fishelov, and J. R. Dorronsoro. "Auto-adaptive multiscale Laplacian Pyramids for modeling non-uniform data." In: *Engineering Applications of Artificial Intelligence* 93 (2020), p. 103682. ISSN: 0952-1976. DOI: `https://doi.org/10.1016/j.engappai.2020.103682`.

[11] C. Fowlkes, S. Belongie, and J. Malik. “Efficient spatiotemporal grouping using the nystrom method.” In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*. Vol. 1. IEEE. 2001, pp. I–I.

[12] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. “Array programming with NumPy.” In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: `10.1038/s41586-020-2649-2`.

[13] P. T. Inc. *Collaborative data science*. 2015. URL: `https://plot.ly`.

[14] S. S. Lafon. “Diffusion maps and geometric harmonics.” In: (2004).

[15] D. Lehmberg, F. Dietrich, G. Köster, and H.-J. Bungartz. “datafold: data-driven models for point clouds and time series on manifolds.” In: *Journal of Open Source Software* 5.51 (2020), p. 2283. DOI: `10.21105/joss.02283`.

[16] A. W. Long and A. L. Ferguson. “Landmark diffusion maps (L-dMaps): Accelerated manifold learning out-of-sample extension.” In: *Applied and Computational Harmonic Analysis* 47.1 (2019), pp. 190–211.

[17] G. Mishne and I. Cohen. “Multiscale anomaly detection using diffusion maps.” In: *IEEE Journal of selected topics in signal processing* 7.1 (2012), pp. 111–123.

[18] E. A. Nadaraya. “On estimating regression.” In: *Theory of Probability & Its Applications* 9.1 (1964), pp. 141–142.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. “Scikit-learn: Machine Learning in Python.” In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[20] N. Rabin and R. R. Coifman. “Heterogeneous datasets representation and learning using diffusion maps and Laplacian pyramids.” In: *Proceedings of the 2012 SIAM International Conference on Data Mining*. SIAM. 2012, pp. 189–199.

[21] D. Slepian. “Prolate spheroidal wave functions, Fourier analysis and uncertainty—IV: extensions to many dimensions; generalized prolate spheroidal functions.” In: *Bell System Technical Journal* 43.6 (1964), pp. 3009–3057.

[22] D. Slepian and H. O. Pollak. “Prolate spheroidal wave functions, Fourier analysis and uncertainty—I.” In: *Bell System Technical Journal* 40.1 (1961), pp. 43–63.

[23] B. Sonday. *Systematic model reduction for complex systems through data mining and dimensionality reduction*. Princeton University, 2011.

[24] S. Van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, and T. Yu. "scikit-image: image processing in Python." In: *PeerJ* 2 (2014), e453.

[25] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[26] C. Williams and M. Seeger. "Using the Nyström Method to Speed Up Kernel Machines." In: *Advances in Neural Information Processing Systems*. Ed. by T. Leen, T. Dietterich, and V. Tresp. Vol. 13. MIT Press, 2001, pp. 682–688.