

---

# RS-HDMR-GPR Code Manual

Version 0.1.3

Owen Ren, Mohamed Boussaidi Ali, Sergei Manzhos, Dmitry Voytsekhovsky

Copyright ©by the authors.

Last Updated: April 13, 2021

# 1 Installation Guide

The latest version of the code can be obtained from [github](#). The code comes with a IPython jupyter notebook (examples.ipynb) that allows one to run and test the examples provided in the accompanying paper.

The **rshdmrgpr** package is installed as a Python package. Please install the 64-bit [Python](#) and [pip](#) (which allows for package installations) on your machine first. We also recommend the user install [anaconda distribution](#) to work with virtual environments and conda virtual environments. The code can also be run on google collab, or any environment that supports jupyter notebooks. In this section we will only provide a windows installation guide because the authors have only worked on this package through windows. However, there should not be anything that prevents a user from using the package on other operating systems (OS).

## 1.1 Windows Installation

On windows command prompt (or Anaconda prompt), create a virtual environment and install the package in a directory of your choosing.

```
C:\Users\new_user>python -m env env_name
C:\Users\new_user>env_name\Scripts\activate.bat
(env_name) C:\Users\new_user>pip install git+https://github.com/owen-
ren0003/rshdmrgpr.git
```

After the package is installed, the user should be able to use it in terminal or any Python IDE with the correct environment setup.

```
(env_name) C:\Users\new_user>python
Python 3.7.4 (default, Aug 9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)]
:: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from rshdmrgpr import *
```

Users can consult [virtualenv](#) for more details on how to work with virtual environments.

## 1.2 Anaconda Setup

If the user decides to install anaconda, this subsection provides instructions on creating a virtual environment and working on interactive jupyter notebooks. Once Anaconda is installed, create a conda environment:

```
conda create -n env_name python=3.6
```

Activate the environment, which in this case is called 'env\_name' (this can be named to the users choosing):

```
conda activate env_name
```

Install the rshdmrgpr package inside the conda environment:

```
pip install git+https://github.com/owen-ren0003/rshdmrgpr.git
```

Install jupyter lab:

```
conda install -c conda-forge jupyterlab
```

Finally, start jupyterlab to starting working interactively in a jupyter notebook:

```
jupyter lab
```

## 2 Python API – Function Reference

The entire implementation is in a single Python script `rs_hdmr_gpr.py` which consists of six helper functions and two classes. The docstrings and descriptions for these helper functions and classes are given below. The implementation uses `GaussianProcessRegressor` class's implementation from the Python `sklearn` library. Each Regressor will act as component functions for our HDMR implementation. In order to train each `GaussianProcessRegressor` we need to provide kernels which are also defined in the `sklearn` library. In the next section, we describe in detail how to use this code.

### 2.1 Helper function `kernel_matrices`

Before working with the algorithm, we need to first load the data sets. The helper function `load_data` will load one of our three built-in datasets. They are respectively: the dataset sampled from the PES of water, the KED dataset from physics, and the financial dataset of S&P 500.

```
rshdmrgpr.load_data(dataset)
```

This function is used to load one of our three built-in data sets as a pandas `DataFrame`.

#### Parameters

- **dataset:** `str` – Specifies which dataset to load. One of 'h2o', 'KED', 'financial'.

#### Returns

- `pandas DataFrame` – The desired dataset.

To conveniently create input parameters for training the standard  $k$ -dimensional HDMR models, the helper function `kernel_matrices` gives the user the option to return such parameters by specifying the dimension, order and the kernel to be used for training the RSHDMRGPR model. The returned lists of this function is used to initialize RSHDMRGPR class. For details on how this is used, please see next section.

```
rshdmrgpr.kernel_matrices(order, dim, kernel_function=RBF)
```

This function is used to create any list of kernels and matrices suitable as inputs for the training step of RSHDMRGPR class. The available list of kernels that can be used are given in kernels.

#### Parameters

- **order:** `int` – The order of HDMR to use.
- **dim:** `int` – The dimension of the feature space.
- **kernel\_function:** `any kernel from sklearn.gaussian_process` – Any kernel used for GPR. Please see documentation on the available kernels. Default is RBF.
- **\*\*kwargs:** `keyword arguments` – Arguments for the given GPR kernel.

#### Returns

- `list of 2D-numpy Array` – List of matrices (2D-numpy arrays), used to select the features to train the component functions. The list should have size equal to the number of component functions the user wants to use to train RSHDMRGPR model.
- `list of sklearn.gaussian_process kernels` – List of to use for training the component functions. The list should have size equal to the number of component functions.

To supplement the RSHDMRGPR class, the helper function below allows for sequential fitting as in equations (5) and (6) in the paper. The inputs are a list of RSHDMRGPR instances to fit in chronological order. Each model is fit to the difference of the label minus the predictions of the previous models.

```
rshdmrgpr.sequential_fitting(x_train, y_train, models, **params)
```

This function fits a list of RSHDMRGPR models sequentially (please see equations (5) and (6) in the paper). Each model is fitted to the difference of the label minus the predictions of the previous models.

#### Parameters

- **x\_train**: `pandas DataFrame` – The training set.
- **y\_train**: `1d - array` – The target vector.
- **models**: `list of RSHDMRGPR` – The list of RSHDMRGPR instances (of class RSHDMRGPR) to fit.
- **\*\*params**: `dict` – variable number of key word arguments. Used to specify arguments for RSHDMRGPR.

#### Returns

- **list of 2D-numpy Array** – List of matrices, used to select the features to train the component functions. Has size (dim choose order).
- **list of RBF kernels** – List of RBF kernels to use for training component function. Has size (dim choose order).

```
rshdmrgpr.sequential_prediction(x_test, models)
```

This function returns a list of predictions for each order of fit. The  $i$ -th (0-based index) element contains the sequential fits up to order  $i$  (i.e. the sum of the predictions from `model[j]` for all  $j = 0, 1, \dots, i$ ). This is used in conjunction with `sequential_fitting` and essentially provides the prediction part to the fitting part.

#### Parameters

- **x\_test**: `pandas DataFrame` – The data set to be predicted on.
- **models**: `list of RSHDMRGPR models` – Contains the list of RSHDMRGPR models to be fitted sequentially.

#### Returns

- **list of 1d-array** – The  $i$ -th element contains the sequential fits up to order  $i$ .

## 2.2 RSHDMRGPR class

This class is used to fit a single instance of an hdmr model. A single instance is the sum of component functions determined by the `matrices` argument (a list of numpy matrices) that determines the component functions via right multiplication with the feature matrix of a given dataset. To perform sequential fitting, we need a list of RSHDMRGPR models and fit each sequentially where successive models fit to the difference of the label and the sum of the previous model predictions. Please see the main article Section 1, equations 5 and 6 for details.

```
class rshdmrgpr.RSHDMRGPR(matrices, kernels)
```

Bases: `object`

This class initializes the RS-HDMR-GPR model.

#### Parameters

- **matrices**: `list of 2D numpy Array` – The matrices that define the HDMR component functions. Every matrix must have the same number of rows equaling the dimension of the feature space.
- **kernels**: `list of kernels` – The list of `sklearn.gaussian_process.kernels` to use for each HDMR component function.

### 2.2.1 RSHDMRGPR class methods

`RSHDMRGPR.verbose_print(msg, end=None, on=True)`

Used to print messages during the train method execution. Used with the verbose argument.

#### Parameters

- **msg:** `str` – The desired message to print.
- **end:** `None` or `str` – Specifies the ending of a line.
- **on:** `bool` – Specifies whether to print or not. This is controlled by the verbose argument in the train method.

Returns `None`

`RSHDMRGPR.train(self, x_train, y_train, alphas=1e-7, n_restarts=1, cycles=50, scale_down=(0.2, 2), optimizer=None, opt_every=5, use_columns=None, initializer='even', report_loss=False, verbose=0)`

Trains the RS-HDMR-GPR model (trains all the GPR models – the component functions).

#### Parameters

- **x\_train:** `pandas DataFrame` – The DataFrame containing the features.
- **y\_train:** `pandas Series` – The Series containing the target values.
- **alphas:** `int` or `(list of int)` – The noise level to be set for the component functions, each of which is a GPR model. If `int`, the noise level for training each of these models is the set to this for all cycles. To set different noise levels for different component functions, a `list of int` must be specified.
- **scale\_down:** `tuple` – Must be a `tuple`, say  $(s, e)$  of size two. Training predictions for each component function will be multiplied by  $\min\left\{s + \frac{(1-s)ec}{T}, 1\right\}$  on cycle  $c$ , where  $T$  denotes the total number of cycles. This is used to prevent overfitting of a single component function.
- **cycles:** `int` – The number of self-consistent cycles to use for training.
- **optimizer:** `str` or `(list of str)` – Must be a GPR optimizer or list of such. If a list is provided, each component function will use the corresponding optimizer specified in the list. Please see the sklearn documentation for GaussianProcessRegressor optimizer.
- **opt\_every:** `int` – Specifies that the optimizer will be applied every (opt\_every) cycles. Default=1 (optimizer applied every cycle). The optimizer will always be applied to the first and last cycles. This argument provides no meaning if optimizer is `None` or a list of `None`.
- **use\_columns:** `list of bool` or `None` – Specifies which column to use (True indicates use, false otherwise) for training. If `None` (the default) is provided, all columns are used.
- **n\_restarts:** `int` – Positive integer indicating the number of restarts on the optimizer (this is a hyperparameter for GPR).
- **initializer:** `list of float` or `'even'` – Initializes the starting targets for each component function. If `'even'`, every target is equal to `y_train / len(use_columns)`.
- **report\_loss:** `bool` – If True, saves the loss from the prediction on the training set over every cycle. Only RMSE (Root mean-squared error) loss is supported for now.
- **verbose:** `int` – Sets the various print details option during training. Takes on values 0, 1, or 2. Default is 0. A level of 2 shows the least detail and a level of 0 shows the most detail.

Returns **self**, `pandas DataFrame` – Returns only the trained instance of self if `report_rmse=False`, otherwise returns the trained instance of self and a DataFrame containing the rmse values.

`RSHDMRGPR.predict(test_data, return_std=False)`

Computes the output of the trained RSHDMRGPR model on a given data set.

### Parameters

- **test\_data:** `pandas DataFrame` – The DataFrame containing the features of the test set. Should have one less column than the data used for training.
- **return\_std:** `bool` – If True, the sum of all the component functions standard-deviation of the predictive distribution at the query points is returned along with the mean.

### Returns

- **y\_pred:** `1D-numpy Array` – Sum of the mean of predictive distribution at query points of all component functions.
- **y\_std:** `1D-numpy Array` – Sum of the standard deviation of predictive distribution at query points of all component functions. Only returned when `return_std=True`.

`RSHDMRGPR.get_models()`

Returns the trained RSHDMRGPR component functions.

Parameters None

Returns `list of GaussianProcessRegressor` – List of `GaussianProcessRegressor` from the `sklearn.gaussian-process` library, their trained instances.

## 2.3 FirstOrderHDMRImpute class

This class uses 1D-HDMR trained RSHDMRGPR instances to impute data sets where each row has at most 1 missing value. It is important that the dataset satisfies this criterion, otherwise the code will not run or yield unexpected behaviour. The initialization of this class uses the trained models returned by `get_models()` method from the RSHDMRGPR class.

**class** `rshdmrgpr.FirstOrderHDMRImpute(models, division=1000)`

Bases: `object`

This class is used for imputation based on first order HDMR. The class is initialized by vectorizing the 1D-HDMR component functions and creating a dictionary of output values with `division` subdivisions. This is done because we need the inverse relation of each component function. This acts as a lookup table for inverse values where the endpoints of the subdivisions are inverse values. The larger the value of `division`, the more precise the inverse will be at approximating.

Parameters None

- **models:** `list of GaussianProcessRegressor` – list of trained `GaussianProcessRegressor` models which represents the hdmr component functions of first order. Must be of first order (each has 1 input and 1 output).
- **division:** `int` – The number of divisions in the lookup table

### 2.3.1 FirstOrderHDMRImpute class methods

The class below provides

`FirstOrderHDMRImpute.get_table()`

Returns the lookup table for the component functions.

Parameters None

Returns `panda DataFrame` – The lookup table for the HDMR component functions.

`FirstOrderHDMRImpute.get_yi(df_na)`

Modifies the DataFrame to contain the output columns of the first-order hdmr component functions. The output columns are concatenated after the last column of `df_na`. The DataFrame in question should have missing values and for each row there should be at most one missing value.

**Parameters** None

- **df\_na:** `pandas DataFrame` – The DataFrame to impute. Must contain the output column and that column must be the last column.

**Returns** `list` – list of column indices for each row having a missing value (of `df_na`).

```
FirstOrderHDMRImpute.impute(df_na, get_candidates=False, threshold=0.001)
```

This function imputes the missing values given the input. Every single input row is expected to have at most one missing value.

**Parameters** None

- **df\_na:** `pandas DataFrame` – The DataFrame to impute. It is expected to contain the columns corresponding to 1D hdmr outputs (i.e. as an output of the `get_yi` function).
- **get\_candidates:** `bool` – If True, returns all the candidates for imputation.
- **threshold:** `float` – The threshold distance to set for selecting candidates from look-up table. Correspond to  $\delta$  in the paper (Section 2.2).

**Returns**

- 1) `pandas DataFrame` – The imputed DataFrame `df_na` if `get_candidates=False`.
- 2) `pandas DataFrame, pandas Index, pandas Series, list of float` – The imputed DataFrame `df_na`, the index of rows with null entries, the column names (indexed by the index of null entries) containing the null entry, and a list of candidates for imputing that missing value if `get_candidates=True`.

## 3 Usage

The function docstrings were given in the previous section. The present section serves as a tutorial on how to invoke these functions and classes.

### 3.1 Training RSHDMRGPR

As a first basic example, we fit a three-dimensional dataset which is obtained by sampling the PES of  $\text{H}_2\text{O}$  which is one of the build-in datasets for this package. The first thing we need to do is import the required packages to begin our analysis.

```
1 import numpy as np
2 import pandas as pd
3
4 from sklearn.metrics import mean_squared_error
5 from sklearn.preprocessing import MinMaxScaler
6 from sklearn.model_selection import train_test_split
7 from sklearn.gaussian_process.kernels import *
8
9 from rshdmrgpr.rs_hdmr_gpr import *
```

Next, we extract the dataset via the helper function `load_data`

```
10 data = load_data('h2o')
11 print(data.shape)
12 data.info()
13 <class 'pandas.core.frame.DataFrame'>
14 RangeIndex: 10001 entries, 0 to 10000
15 Data columns (total 4 columns):
16 #   Column   Non-Null Count  Dtype
17 ---  ---
18 0    a1      10001 non-null   float64
19 1    a2      10001 non-null   float64
20 2    a3      10001 non-null   float64
21 3    out     10001 non-null   float64
22 dtypes: float64(4)
23 memory usage: 312.7 KB
```

The dataset consists of 4 columns where the first three columns are features and the last column is the target. To run RS-HDMR-GPR, we need to re-scale the dataset linearly so that the columns are between  $[0, 1]$ . (Alternatively any re-scale method works). We do this using `MinMaxScaler` from the `sklearn` package.

```
13 scale = data['out'].max() - data['out'].min() # Keep the original scales for the target column
14 scaler = MinMaxScaler(feature_range=(0, 1)) # Initializes MinMaxScaler instance
15 scaler.fit(data) # Fits the instance to the current dataset
16 # Transforms the data set to be between [0, 1]
17 data_scaled = pd.DataFrame(scaler.transform(data), columns=['a1', 'a2', 'a3', 'out']) # The \
    transformed data set.
```

After transforming the dataset, we are now ready to split for training and prediction. Our training set will consist of 1000 rows and our test set will be the entire PES dataset. We split the data using `train_test_split`.

```
18 x_train, x_test, y_train, y_test = train_test_split(data_scaled.drop(columns=['out']), data_scaled['\
    out'], train_size=1000, test_size=None, random_state=42)
```

In order to initialize RS-HDMR-GPR class we need the list of matrices and kernels to be used to define and train the component functions. We start with training the 1d-hdmr-gpr model. First we call the helper function `kernel_matrices` to quickly give us the matrices and kernels. For the kernels, we will use the RBF kernels each with length scales of 0.6.

```
19 matrices, kernels = kernel_matrices(1, 3, kernel_function=RBF, length_scale=0.6)
20 print(matrices)
21 print(kernels)
22 [array([[1.],
23        [0.],
24        [0.]])],
25 array([[0.],
26        [1.],
27        [0.]])],
28 array([[0.],
29        [0.],
30        [1.]])]
31 [RBF(length_scale=0.6), RBF(length_scale=0.6), RBF(length_scale=0.6)]
```

We can see that there are three matrices in the list each corresponding to the basis vector of  $\mathbb{R}^3$ . Each of these matrix is left multiplied with the data set to give only the column corresponding to the 1. For details, please see accompanying documentation, section 2.1 for more details. With the matrices and kernels we can initialize the class and train on the training set:



```
22 hdmr = RSHDMRGPR(matrices, kernels)
23 hdmr.train(x_train, y_train)
```

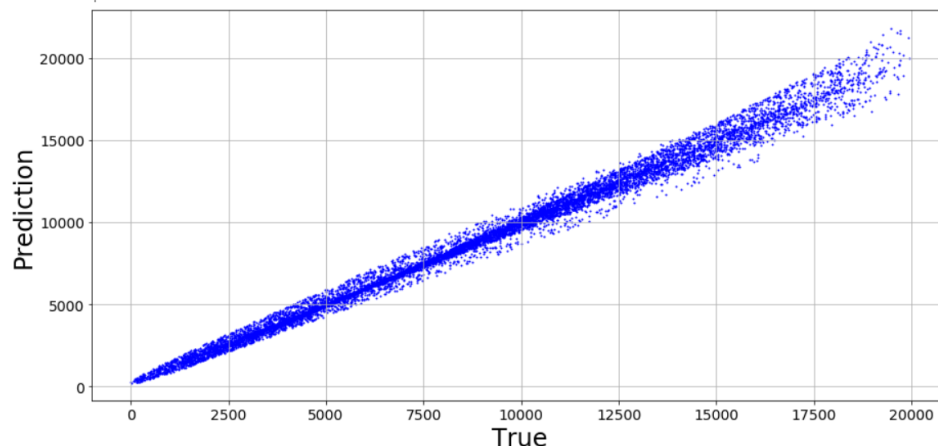
After training, we predict on the entire data set

```
24 y_pred = hdmr.predict(data_scaled.iloc[:, :3])
```

The correlation plot between the true values vs the predicted values is shown below:

```
25 correlation_plot(data_scaled["out"] * scale, y_pred * scale, xlabel='True', ylabel='Prediction', \
    figsize=(15,7))
```

Root mean squared error: 452.88959847675716



In this example, we used the default parameters for training. The user has control of 8 hyperparameters: `scale_down`, `optimizer`, `n_restarts`, `alphas`, `cycles`, `use_columns`, `opt_every`, and `initializer` that accounts for overfitting/regularization and other tasks. These were described in the previous examples. We won't go into detail here as they are explained in the docstrings as well as the accompanying document. However, we have attached a notebook that explains each of these parameters in detail as a part of the supporting information. For further details, please see Section 5 of this manual.

### 3.2 1D - HDMR imputation

We demonstrate how to imputation class works on a synthetic example. The imports are the same as the previous subsection, so we omit the import code. We create a synthetic data set that has 3 feature columns and the target column is the sum of these three columns:

```
26 def f0(*args):
27     sumed = 0
28     for a in *args:
29         sumed += a
30
31     return sumed
32
33 column_names = []
34 n = 3
35 for i in range(n):
36     column_names.append(f'a{i}')
37 np.random.seed(42) # Fixes the random seed so the results are always the same.
38 data = pd.DataFrame(np.random.rand(10000, 3), columns=column_names)
39 inputs = [data.iloc[:, i] for i in range(n)]
40 data['out'] = f0(*inputs)
41
42 # Scales the data to be between [0, 1]
43 scale = data['out'].max() - data['out'].min()
44 trans = data['out'].min()
45 data['out'] = (data['out'] - trans) / scale
46 x_train, x_test, y_train, y_test = train_test_split(data, data['out'], train_size=400, test_size=None, \
    random_state=42)
```

Again we use the entire data set for prediction.

We train the model on 100 points and set 300 of them aside for imputation. For these 300 points the first 100 rows have their first column set to missing, the second 100 rows have the second column set to missing, and finally the last 100 rows have the third column set to missing.

```
47 # Setting up DataFrame to be imputed.
48 df_na = x_train[100:].copy()
49 df_na.iloc[100, 0] = np.nan
50 df_na.iloc[100: 200, 1] = np.nan
51 df_na.iloc[200:, 2] = np.nan
```

Now we initialize first order imputation class. This class must be initialized

```
52 foimp = FirstOrderHDMRImpute(gpr.get_models())
```

In order to impute for the missing values, we need to compute the outputs of the component functions and append it to the DataFrame.

```
53 # Appends the output columns for the component functions
54 foimp.get_yi(df_na)
```

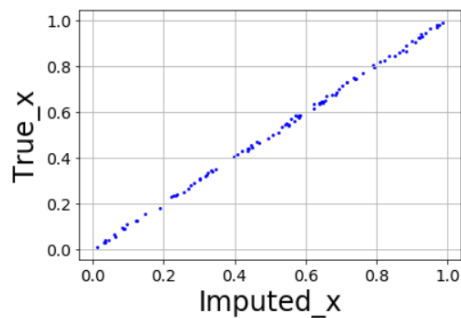
With these output values from the component functions, we now impute for the missing values.

```
55 # Imputes the missing values
56 foimp.impute(df_na)
```

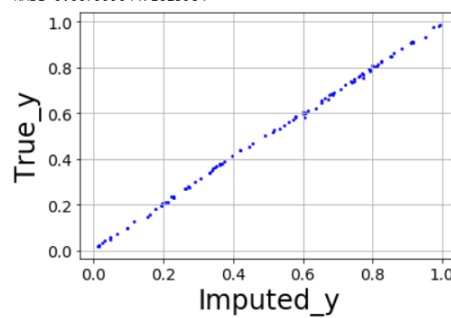
Finally, with the imputed values, we show how close they are compared to the actual values.

```
57 labels = ['x', 'y', 'z']
58 for i in range(3):
59     correlation_plot(df_na.iloc[100*i: 100*(i + 1), i], x_train.iloc[100*(i+1): 100*(i+2), i], xlabel=f'\
        f'Imputed_{labels[i]}', ylabel=f'True_{labels[i]}')
```

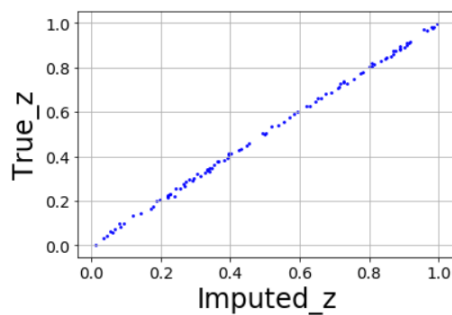
RMSE 0.007580697422636046



RMSE 0.00799904471623984



RMSE 0.007480492417202555



## 4 Algorithms

This section explains the implementation of the following three algorithms in pseudo-code format: The RS-HDMR-GPR algorithm, the algorithm for sequentially fitting the RS-HDMR-GPR models, and the imputation using 1D-HDMR. Notations:

1.  $m$  – number of entries for the data.
2.  $n$  – number of features the data contains.
3.  $\alpha(c)$  – A scale down function as a function of the cycle number  $c$ .
4.  $P_\ell(i)$  – A permutation of  $\{1, 2, \dots, \ell\}$  applied to element  $i$ .
5.  $X$  denotes a  $m \times n$  matrix containing data corresponding to the features.
6.  $y$  denotes a  $m \times 1$  vector containing the target data.

---

### Algorithm 1: RS-HDMR-GPR

---

**Input:**

1.  $\ell$  matrices  $A_1, A_2, \dots, A_\ell$ , each of size  $n \times k$ .
2. GPR kernels  $K_i$  for  $i = 1, 2, \dots, \ell$ . (Usually successive orders of HDMR. The kernels supported for now are RBF and Matern. For this, the noise level and length scales needs to be provided as user inputs. Both anisotropic and isotropic kernels work).
3. The feature matrix  $X$  and label vector  $y$ .
4. A scale-down function  $\alpha(c)$ .

**Output:** Trained RS-HDMR-GPR model

- 1 Initialize vectors  $\vec{y}_i = \frac{1}{\ell} \vec{y}$  and GPR models  $\mathbf{model}_i = \emptyset$  for all  $i = 1, 2, \dots, \ell$ .
  - 2 Set  $P_\ell$  to be the identity permutation. **for**  $1 \leq c \leq \# \text{ cycles}$  **do**
  - 3     **for**  $1 \leq i \leq \ell$  **do**
    1. Train  $\mathbf{model}_{P(i)}$  using kernel  $K_{P(i)}$  with input  $XA_{P(i)}$  and output  $\vec{y} - \sum_{j \neq P(i)} \vec{y}_j$ .
    2. Update  $\vec{y}_{P(i)} := (\mathbf{model}_{P(i)})$ 's prediction on  $XA_{P(i)} \times \alpha(c)$
    3. Update  $K_{P(i)}$  parameters to trained  $\mathbf{model}_{P(i)}$ 's new length scale if optimizer is applied.
  - 4     Choose a random permutation  $P_\ell$  if permutation option is chosen.
- 

For Algorithm 1, we did not include the various parameters from the implementation. These are explained in the accompanying notebooks.

---

### Algorithm 2: Sequential Fitting

---

**Input:**

1. A list of untrained RS-HDMR-GPR models:  $\mathbf{model}_i$  for  $i = 1, 2, \dots, k$ .
2. The feature matrix  $X$  and label vector  $y$ .

**Output:** Trained instances of  $\mathbf{model}_i$  for  $i = 1, 2, \dots, k$ .

- 1 **for**  $1 \leq i \leq k$  **do**
  - 2     Train  $\mathbf{model}_i$  with respect to  $X$  and  $\vec{y}$
  - 3     update  $\vec{y}$  to  $\vec{y} - \vec{z}$  where  $\vec{z}$  is the prediction of  $\mathbf{model}_i$  on  $X$ .
- 

Algorithm 2 takes a collection of RSHDMRGPR instances and fits them in the order they are given. It is done via the `sequential_fitting` function from the code. The function `sequential_prediction` is used to predict for after sequential fitting is performed. Please see API and the notebook in SI for further details.

---

**Algorithm 3:** First order RS-HDMR-GPR imputation

---

**Input:**

1. 1D-HDMR component functions  $f_1, f_2, \dots, f_n$
2. A  $m \times n$  feature matrix  $X$  whose columns entries are scaled to be between  $[0, 1]$ . For each row of  $X$ , there is at most one entry that is missing.
3. An integer  $d > 0$ .

**Output:** A  $m \times n$  feature matrix  $X'$  that has it's missing values imputed

- 1 Compute the lookup table for the  $f_i$ 's with  $d$  subdivisions of  $[0, 1]$ .
  - 2 **for** each row  $r$  of  $X$  with index  $i$  having a missing value **do**
  - 3     Compute  $y_i = \vec{y}_i - \sum_{j \neq i} y_j$
  - 4     Determine a list of input values in the lookup table closest to or within  $\frac{2}{d}$  of  $y_i$ .
  - 5     The first element from this list is used for imputing the missing entry.
- 

For Algorithm 3, it is important that input data matrix contains at most 1 missing value per row. The method implemented only works only for this case.

## 5 rshdmrgpr Examples

The supplementary IPython jupyter notebooks (`examples.ipynb` and `physics_example.ipynb`) demonstrates in detail how to use this package on the built-in datasets. We have tried to make the examples exhaustive in terms of code usage. That is we explain through these examples the various hyperparameters and code usages of RS-HDMR-GPR implementation. All of the examples presented in the accompanying document are worked out in these notebooks.