Spatialize

The Spatialize library: User Manual

Version 1.0.2

ALGES

August 8, 2025

Contents

A The Spatialize library: User Manual

A.1 Downloading and installation

The `spatialize` package is compatible with Python 3.8 and later versions, offering support across Linux, macOS, and Windows operating systems. This package is readily accessible through the Python Package Index (PyPI) at `https://pypi.org/project/spatialize`. To install `spatialize`, you have two convenient options:

1. Install from PyPI:

   ```
   pip install spatialize
   ```

2. Install from GitHub:

   ```
   pip install -U git+https://github.com/alges/spatialize.git
   ```

These installation methods ensure you can easily integrate `spatialize` into your Python environment, regardless of your preferred source. By using pip, Python's package installer, you can effortlessly manage dependencies and keep your project up-to-date with the latest `spatialize` features and improvements.

The `spatialize` Python package is built based on some existing dependencies, the most relevant of which are listed as requirements in the package itself. Among them, one can highlight **NumPy** ([2]) and **pandas** ([4]), which are used for data handling (grids and samples), for data import, and to export the estimation results. **Matplotlib** ([3]) It is widely used for illustration purposes. **Scikit-learn** ([5]) is used to standardise the parameter search. The next subsections will extend the description of the library.

A.2 General library description

The `spatialize` library is built on three levels. The first two constitute the API of this library, which is implemented in Python. The one closest to the user consists of three high-level functions that allow spatial interpolation in two dimensions on gridded and non-gridded data and a hyperparameter search for these cases. On a second level, aggregation functions are provided and functions for estimating precision. In the latter case, a class is also provided to implement this type of custom function. Finally, at a third level, an efficient implementation of the methods contained in the spatial estimation functions in the C++

language is found. At the third level, `spatialize` is designed and implemented in such a way that it can work in three or more dimensions. In the first two levels, the API provides only two in this first version.

In addition, `spatialize` includes an efficient implementation of the Inverse Distance Weighting (IDW) spatial estimation method, together with a hyperparameter search function for this tool. A detailed description of these functions is not included in this publication, as they are not part of the ESI method and its alternatives but are only add-ons. However, examples of their use are presented in the "Usage examples" section. On the other hand, future versions of the library are planned to integrate an efficient implementation of Kriging, which will include nested structures for the adjustment of the experimental variograms.

Our library is an efficient tool designed and implemented to work with large datasets. Therefore, the examples provided and described in the "Usage examples" section are in the form of scripts, and no Jupyter notebooks have been provided. If the user wants to use this type of environment, the library will work, but it will quickly accumulate rubbish in memory. This is why, in these cases, it is important to work with a limited depth of each tree (use a parameter `alpha` no greater than 0.90) and a number of trees in the order of 100. These values will depend on the amount of data, so they are a reasonable estimate for examples such as those presented in our Scripts.

A.3 High level API

The main objective of `spatialize` is to provide an easy-to-use tool for non-experts in classical geostatistics. Therefore, the high-level API is composed of functions that are very simple to call and hide all the details outlined in the previous sections from the user. In this section, we describe each of these functions in detail, along with their main arguments.

A.3.1 General input data format

Like any interpolation function, the basic input data are:

- `points`: Contains the coordinates of known data points. This is an $N_s \times D$ array, where N_s is the number of data points, and D is the number of dimensions.
- `values`: The values associated with each point in points. This must be a 1D array of length N_s.

- `xi`: If the data are gridded, they correspond to an array of grids of D components, each with the dimensions of one of the grid faces, $d_1 \times d_2 = N_{x^*}$, where N_{x^*} is the total number of unmeasured locations to estimate. Each component of this array represents the coordinate matrix on the corresponding axis, as returned by the functions `numpy.mgrid` in Numpy, or `meshgrid` in Matlab or R.

  If the data are not gridded, they are simply the locations at which to evaluate the interpolation. It is then an $N_{x^*} \times D$ array.

  In both cases, D is the dimensionality of each location, which coincides with the dimensionality of the `points`.

A.3.2 ESI estimation of gridded data

This is the function used to make an estimate with ESI in the case of sample data and unmeasured locations that are on a grid – i.e. where the format of `xi` is as detailed in the previous section.

Module:

```
spatialize.gs.esi
```

Signature:

```
def esi_griddata(points, values, xi, |\textit{<optional named arguments>}|)
```

In the optional named arguments, it is possible to set the local interpolation method to be used (IDW or Kriging), the partition process to generate the random partition set (Voronoi or Mondrian) and other options related to the interpolation methods. The following list contains all these options, with their default values and descriptions.

To define the process of generating random partitions:

- `p_process`: Indicates the stochastic process used to generate the random partitions. The options are `"mondrian"` and `"voronoi"`. The default value is `"mondrian"`.
- `n_partitions`: Number of random partitions (m) to be generated. Once the estimation process is completed, each partition corresponds to an estimation scenario. The default value is `500`.

- `data_cond`: In case of using `p_process="voronoi"`, this parameter indicates whether the random kernel generation process is conditioned by the sample data, as explained in the "Model training" section. The default value is `True`.
- `alpha`: Sets the coarseness level of each random partition. Corresponds to the $\alpha \in [0, 1)$ parameter described in the "Rule of thumb for parameter choice" section. As mentioned there, $\alpha = 0$ will generate the coarsest partition, while $\alpha \to 1$ will generate finer ones. The default value is `0.8`.

To define and configure the local interpolator:

- `local_interpolator`: This is where the function $\mathbf{S}_{\mathcal{L}_k}(x^*)$, described in the "Weak voter function set generation" section, is defined. In the current version, it can be `"idw"` or `"kriging"` when `p_process= "mondrian"` is defined. If `p_process="voronoi"` is defined, it can only be `"idw"`.

if `local_interpolator="idw"` is defined, the method described by Equation (1) is used to implement IDW.

$$IDW(\mathbf{x}^*) = \begin{cases} z(\mathbf{x}_i) & \text{if } \mathbf{x}_i = \mathbf{x}^* \\ \frac{\sum_i w_i \cdot z(\mathbf{x}_i)}{\sum_i w_i} & \text{otherwise} \end{cases} \tag{1}$$

where,

$$w_i = \frac{1}{d(\mathbf{x}_i, \mathbf{x}^*)^p} \tag{2}$$

and $z(\mathbf{x}_i)$ are the values of the known samples measured at locations $\mathbf{x}_i \in \mathcal{L}_k$, for some element of a particular partition or tree T_k. Then,

- `exponent`: It is the parameter p of Equation 2, which defines whether or not to smooth the weight w_i for the sample $\mathbf{x}_i$. Note that if $p = 0$, each sample is assigned the same weight in the interpolation (equivalent to the simple mean of the neighbours), and if $p \to \infty$, the nearest sample is assigned a weight of 1, while all other samples are assigned a weight of 0 (equivalent to a nearest-neighbour interpolation). The default value is `2.0`.

If `local_interpolator="kriging"` is defined, a standard ordinary Kriging calculation method is applied, as described in detail in [1]. The variogram model used is fixed (i.e. it is not fitted from an experimental variogram) and isotropic, as it is applied to a very small data set (in each cell of each partition). Then,

- `model`: Indicates the variographic model used for the ordinary Kriging. In `spatialize`, the general variogram has the form:

  $$\gamma(h) = \begin{cases} s \cdot \mathrm{m}(h) & 0 \leq \mathrm{m}(h) \leq 1 \\ 0 & \mathrm{m}(h) < 0 \\ 1 & \mathrm{m}(h) > 1 \end{cases}$$

  Where $\mathrm{m}(h) = (1 - n) \cdot (1 - \mathrm{model}(h))$ and $\mathrm{model}(h)$ represents one of the following functions depending on the chosen model:

  - `"spherical"`:
    $$\mathrm{spherical}(h) = 1.5\frac{h}{r} - 0.5\left(\frac{h}{r}\right)^3$$
  - `"exponential"`:
    $$\mathrm{exponential}(h) = 1 - e^{-3\frac{h}{r}}$$
  - `"cubic"`:
    $$\mathrm{cubic}(h) = \left(\frac{h}{r}\right)^2\left(7 - 8.75\frac{h}{r} + 3.5\left(\frac{h}{r}\right)^3 - 0.75\left(\frac{h}{r}\right)^5\right)$$
  - `"gaussian"`:
    $$\mathrm{gaussian}(h) = 1 - e^{-3\left(\frac{h}{r}\right)^2}$$

  The default model is `"spherical"`. Parameters n, r and s must be specified in arguments:

- `nugget` (with default value `0.1`), `range` (with default value `5000.0`) and `sill` (with default value `1.0`), respectively.

With the above in place, the aggregation function only remains to be defined for the preliminary estimate returned by this function. Then,

- `agg_function`: Aggregation function *G* to calculate the estimate at locations `xi` from the ESI samples, as explained in Sections A.3.4. This function can be any of the functions defined in the module `spatialise.gs.esi.aggfunction` or a user-defined custom function (see more details in Section A.4.1). The default function is `spatialize.gs.esi.aggfunction.mean`.

Returns:

- (`ESIResult`): As a result, the function returns an object, which is an instance of the class `ESIResult`, containing the preliminary estimate according to the provided arguments. This class provides a set of methods to display aspects of the result, such as the aggregate estimate, the scenarios of the different partitions, or a precision calculation based on some loss function – for full details, see A.3.4.

A.3.3 ESI estimation of non-gridded data

This function generates an estimate in ESI space, from a set of sample points (i.e. measured locations), at a set of unmeasured points at arbitrary locations in space. This implies that the `xi` positions must come in the appropriate format, as described in A.3.1.

Module:

```
spatialize.gs.esi
```

Signature:

```
def esi_nongriddata(points, values, xi, |\textit{<optional named arguments>}|)
```

Except for `xi`, all the arguments of this function are the same as those of the function for gridded data detailed in the previous section (A.3.2).

Returns:

- (`ESIResult`): As a result, this function also returns an object, which is an instance of the class `ESIResult`, containing the preliminary estimate according to the provided arguments. This class provides a set of methods to display aspects of the result, such as the aggregate estimate, the scenarios of the different partitions, or a precision calculation based on some loss function – for full details, see A.3.4.

A.3.4 ESIResult class and its methods

Both functions for generating ESI estimates, `esi_nongriddata` and `esi_griddata`, return instances of the `ESIResult` class. An object of this class allows the manipulation of the results in a disaggregated form, the calculation of the final estimate based on different aggregation functions, and the calculation of its precision based on different loss functions. Finally, it allows plotting results in different useful ways.

```
class ESIResult(EstimationResult)
```

This class has the following set of methods:

- `esi_samples`: The central concept for dealing with ESI estimation results is the *ESI sample*. In this sense, it should be noted that each random partition delivers an estimate for each of the locations provided in the argument `xi` (for both gridded and non-gridded data). The set of estimates for a particular partition is what in `spatialize` is considered an *ESI sample*.

  This method then returns the set of all ESI samples, one for each random partition, calculated for the estimation.

  - **Signature:**

    ```
    def esi_samples()
    ```

  - **Returns:**

    * (`ndarray`): An array of dimension $N_{x^*} \times m$ (m = `n_partitions` in both function `esi_griddata` and `esi_nongriddata`), for non-gridded data, and of dimension $d_1 \times d_2 \times m$ for gridded data – remember that, in this case, $d_1 \times d_2 = N_{x^*}$ (see Section A.3.1).

- `estimation`: Returns the estimated values at locations `xi` by aggregating all ESI samples using the aggregation function provided in the `agg_function` argument (in both functions `esi_griddata` and `esi_nongriddata`). This estimate can be changed using another aggregation function with the `re_estimate` method of this same class.

  - **Signature:**

    ```
    def estimation()
    ```

  - **Returns:**
    * (`ndarray`): An array of dimension N_{x^*}, for non-gridded data, and of dimension $d_1 \times d_2$ for gridded data – remember that, in this case, $d_1 \times d_2 = N_{x^*}$ (see Section A.3.1).

- `re_estimate`: It recalculates the final estimate based on the aggregation function provided (e.g. by taking the mean of the ESI samples). This method updates the internal estimate and returns the new result. Then, the next time the `estimation` method is called, this is the estimate it will return.
  - **Signature:**

    ```
    def re_estimate(agg_function)
    ```
  - **Parameters:**
    * `agg_function`: Aggregation function G to calculate the estimate at locations `xi` from the ESI samples, as explained in the "Ensemble spatial interpolation" section and A.3.4. It can be any of the functions defined in the module `spatialise.gs.esi.aggfunction` or a user-defined custom function (see more details in Section A.4.1). The default function is `spatialize.gs.esi.aggfunction.mean`.
  - **Returns:**
    * (`ndarray`): An array of dimension N_{x^*}, for non-gridded data, and of dimension $d_1 \times d_2$ for gridded data – remember that, in this case, $d_1 \times d_2 = N_{x^*}$ (see Section A.3.1).

- `precision`: Calculates the precision (or error) between the estimate and the ESI samples using the specified loss function as explained in the "Interpolation precision modelling" section.
  - **Signature:**

    ```
    def precision(loss_function)
    ```
  - **Parameters:**
    * `loss_function` (`function`, optional): A loss function to calculate the precision, defaulting to `spatialize.gs.esi.lossfunction.mse_loss`. It can

be any of the functions defined in the module `spatialise.gs.esi.loss function` or a user-defined custom loss function (see more details in Section A.4.2).

  - **Returns:**
    * (`ndarray`): An array of dimension N_{x^*}, for non-gridded data, and of dimension $d_1 \times d_2$ for gridded data – remember that, in this case, $d_1 \times d_2 = N_{x^*}$ (see Section A.3.1).

- `precision_cube`: It applies a loss (error) function to each ESI sample with respect to the current estimate. The difference with the `precision` method is that it does not aggregate the result over the total calculated losses, returning the total data “cube" whose dimensions are the same as the ESI samples cube.
  - **Signature:**

```python
def precision_cube(loss_function=mse_cube)
```

  - **Parameters:**
    * `loss_function` (`function`, optional): A loss function to calculate the precision cube, defaulting to `spatialise.gs.esi.lossfunction.mse_cube`. It can be any of the functions contained in module `spatialise.gs.esi.loss function` (or a custom user-defined) whose aggregation function is the `identity` function (which produces a data cube as a result).
  - **Returns:**
    * (`ndarray`): An array of dimension $N_{x^*} \times m$ (m = `n_partitions` in both function `esi_griddata` and `esi_nongriddata`), for non-gridded data, and of dimension $d_1 \times d_2 \times m$ for gridded data – remember that, in this case, $d_1 \times d_2 = N_{x^*}$ (see Section A.3.1).

- `plot_estimation`: Plots the estimation using `matplotlib`.
  - **Signature:**

```python
def plot_estimation(ax, w, h, **figargs)
```

  - **Parameters:**

    * `ax` (`matplotlib.axes.Axes`, optional): The `Axes` object to render the plot on. If `None`, a new `Axes` object is created.
    * `w` (`int`, optional): The width of the image (if the data is reshaped).
    * `h` (`int`, optional): The height of the image (if the data is reshaped).
    * `**figargs` (optional): Additional keyword arguments passed to the figure creation (e.g., DPI, figure size).

- `plot_precision`: Plots the precision using `matplotlib`. If the precision has not been computed yet, it calls the `self.precision()` method to calculate it.
  - **Signature:**

    ```
    def plot_precision(ax, w, h, **figargs)
    ```

  - **Parameters:**
    * `ax` (`matplotlib.axes.Axes`, optional): The `Axes` object where the precision plot will be rendered. If `None`, a new `Axes` object will be created.
    * `w` (`int`, optional): The width of the image (if the data is reshaped).
    * `h` (`int`, optional): The height of the image (if the data is reshaped).
    * `**figargs` (optional): Additional keyword arguments passed to the figure creation (e.g., DPI, figure size).

- `quick_plot`: Creates a quick, side-by-side plot of the `estimation` and `precision` using `matplotlib`. It uses `self.plot_estimation()` and `self.plot_precision()` to render the plots. The figure is returned for further use or display.
  - **Signature:**

    ```
    def quick_plot(w, h, **figargs)
    ```

  - **Parameters:**
    * `w` (`int`, optional): The width of the image (if the data is reshaped).
    * `h` (`int`, optional): The height of the image (if the data is reshaped).
    * `**figargs` (optional): Additional keyword arguments passed to the figure creation (e.g., DPI, figure size).

A.3.5 ESI hyperparameter search

Apart from providing a general-purpose geostatistical tool (i.e. for non-experts in geostatistics), `spatialize` also aims to make the process as automatic as possible. Thus, the function presented in this section allows a grid search of the best parameters for an estimate. This is done by using parameterizable cross-validation on training and test sets, either with *k*-fold, with random subsets, or *leave-one-out* for exhaustive estimation.

Module:

```
spatialize.gs.esi
```

Signature:

```
def esi_hparams_search(points, values, xi, |\textit{<optional named arguments>}|)
```

The mode of use is very similar to that of `esi_griddata` and `esi_nongriddata`. The following parameters are fixed and have the same meaning as in these functions (Sections A.3.1, A.3.2 and A.3.3):

- `points`, `values`, `xi`
- `p_process`
- `local_interpolator`

In addition to the latter, the following must also be defined:

- `k`: Number of subsets for the *k*-fold round. If `k`=N_{x^*} or `k=-1` a *leave-one-out* round is run. The default value is `10`.
- `griddata`: If `True`, it is to indicate that the estimation is on a grid (`xi`). The default value is `False`.

The grid search parameters must be entered as a set of options in the domain of the argument to be searched – each argument has the same description and scope as in `esi_griddata` and `esi_nongriddata`. Then:

- `data_cond (list)`
  - Valid only when `p_process="voronoi"`.
  - **Usage example**: `data_cond=[True, False]`

- `n_partitions (list)`
  - **Usage example**: `n_partitions=[150, 100, 50]`
- `alpha (list)`
  - **Usage example**: `alpha=list(np.flip(np.arange(0.70, 0.90, 0.01)))`

if `local_interpolator="idw"` is defined,

- `exponent (list)`
  - **Usage example**: `exponent=list(np.arange(1.0, 15.0, 1.0))`

if `local_interpolator="kriging"` is defined,

- `model (list)`
  - **Usage example**: `model=["spherical", "exponential", "cubic", "gaussian"]`
- `nugget (list)`:
  - **Usage example**: `nugget=[0.0, 0.5, 1.0]`
- `range (list)`
  - **Usage example**: `range=[10.0, 50.0, 100.0, 200.0]`
- `sill (list)`
  - **Usage example**: `sill=[0.9, 1.0, 1.1]`

Finally, one can also search for an optimal aggregation function with:

- `agg_function (dict)`:
  - This dictionary can include any of the functions defined in the `spatialise.gs.esi.aggfunction` module or any user-defined custom functions (see more details in the A.4.1 section).
  - **Usage example**: `agg_function={"mean": af.mean, "median": af.median}` (if `af` has been imported with `spatialize.gs.esi.aggfunction as af`).

Returns:

- (`ESIGridSearchResult`) This function returns an instance of the class `ESIGridSearch Result`, detailed below (Section A.3.6). In addition, an example of the use of these tools is shown in the "Hyperparameter search" section.

A.3.6 ESIGridSearchResult class and its methods

This class contains the results returned by the parameter search function `esi_hparams_search()`, described in Section A.3.5.

```
class ESIGridSearchResult(GridSearchResult)
```

It has two methods:

- `best_result`: Constructs a dictionary with the parameters to make the estimate (gridded or non-gridded) corresponding to the smallest cross-validation error in the hyperparameter search made by the `esi_hparams_search()` function. This dictionary is not intended to be manipulated by the user but to be passed directly to the `esi_griddata` and `esi_nongriddata` functions. Thus, a typical call to this method would be:

```
search_result = esi_hparams_search(points, values, (grid_x, grid_y),
                                   griddata=True,
                                   <...>)
result = esi_griddata(points, values,
                      (grid_x, grid_y),
                      best_params_found=search_result.best_result())
```

  - **Signature:**

    ```
    def best_result()
    ```

  - **Returns:**

    * (`dict`): The dictionary with the optimal values found in the cross-validation.

- `plot_cv_error`: It shows a graph of the cross-validation errors of the hyperparameter search process. The graph has two components: the first is the error histogram, and the second is the error level for each of the estimation scenarios generated by the gridded parameter search.

  - **Signature:**

```
        def plot_cv_error()
```

A.4 Low level API

The high-level API is intended for users who are not experts in either geostatistics or computer programming. In this section, we describe some features of `spatialize` that require some Python programming skills and are intended to make the most of the output of ESI estimations in terms of analysis.

A.4.1 Aggregation functions

The aggregation functions G, as presented in the "Ensemble spatial interpolatio" section, are contained in the module:

```
spatialize.gs.esi.aggfunction
```

Their main use is to aggregate the ESI samples generated by the ESI estimation process. However, they are also useful for aggregating the samples generated by the precision calculation between an aggregated estimate and the ESI samples of the estimate. The latter is discussed in more detail in the section A.4.2.

In practice, any function with the following signature can be considered as an aggregation function in `spatialize`:

```python
def f(samples):
    ...
```

The `samples` argument of `f` is always considered to be non-gridded – that is, an array of dimension $N_{x^*} \times m$ (m = `n_partitions` in both `esi_griddata` and `esi_nongriddata`). When the estimates are gridded, `spatialize` flattens them to treat them internally as non-gridded and reshapes them on return. In this context, the expected behaviour of `f` is to return an array of dimension N_{x^*} with the aggregate on the second axis of the samples, i.e. aggregating the m `samples`. For example, the code for the predefined `mean` aggregation function in `spatialize` is:

```python
import numpy as np

```

```python
def mean(samples):
    return np.nanmean(samples, axis=1)
```

The standard aggregation functions included in the library are:

- `mean`: Calculates the mean of a set of samples.
  - **Signature:**

    ```python
    def mean(samples)
    ```

  - **Returns:** The arithmetic mean of the provided samples.
- `median`: Calculates the median of a set of samples.
  - **Signature:**

    ```python
    def median(samples)
    ```

  - **Returns:** The median value of the provided samples.
- `MAP`: Calculates the Maximum A Posteriori (MAP) estimate.
  - **Signature:**

    ```python
    def MAP(samples)
    ```

  - **Returns:** In the current version, the mode of the empirical distribution is returned.
- `class Percentile`: This is a class for creating functions (callable instances) that belong to a family of functions indexed by a given `percentile`.
  - **Constructor signature:**

    ```python
    class Percentile(percentile)
    ```

  - **Returns:** When the constructor of the class is called, it returns a function that calculates the `percentile`, passed as an argument, over the samples. The returned function has the general signature of an aggregation function. For example, the code to obtain an aggregation function p75 that calculates the 75th percentile is:

    ```python
    p75 = Percentile(75)
    ```

    The function p75, for example, can be used to make a call such as

```
result = esi_griddata(...,
                      agg_function=p75,
                      ...)
```

- `class WeightedAverage`: This is a class for creating functions (callable instances) that belong to a family of functions indexed by a given set of `weights`.
  - **Constructor signature:**

    ```
    class WeightedAverage(weights, normalize, force_resample)
    ```

  - **Returns:** When the class constructor is called, it returns a function, say `ws`, which calculates a weighted average of the samples provided. By default, when the `weights` are not provided as an argument in the constructor, they are generated randomly, drawn from a Dirichlet distribution. If `force_resample=True` is also specified (this is the default), the weights will be generated each time `ws` is called. Finally, if `normalize=True` is defined (default value is `False`), the result returned by `ws` is normalised to the range of the samples to avoid excessive smoothing for some set of weights.

    For example, when used in the following way, `ws` is called only once:

    ```
    ws = WeightedAverage()
    result = esi_griddata(...,
                          agg_function=ws,
                          ...)
    ```

    But in the following lines of code, `e1` and `e2` will be different estimates because the weights will be different for each call:

    ```
    result.re_estimate(ws)
    e1 = result.estimation()

    result.re_estimate(ws)
    e2 = result.estimation()
    ```

- `bilateral_filter`: Non-linear filter acting on ESI samples as an aggregation function. It works in both spatial and sample dimensions. Its effect is that in the final estimation, it reduces noise while preserving the areas where there are abrupt changes (edges or high-frequency areas) in the variable to be estimated ([6]).

- **Signature:**

  ```
  def bilateral_filter(samples)
  ```

- **Returns:** A filtered version of the input cube, with reduced noise and preserved edges.

• `identity`: Its main function is to be passed as an aggregation function in the definition of “cube" loss functions, where only the loss (error) between the ESI samples and the estimate is calculated, but the result is not aggregated. This leaves the possibility to explore more complex aggregation functions to calculate precision – more details in Section A.4.2.

- **Signature:**

  ```
  def identity(samples)
  ```

- **Returns:** Returns the input data as-is (identity function).

A.4.2 Loss functions

We will now review the module content.

```
spatialize.gs.esi.lossfunction
```

Before going into the functions themselves, as mentioned above, `spatialize` provides a powerful framework to easily build any uncertainty quantification scheme based on the precision model. It is useful to review that model here to identify its components and to understand how it is expressed in the `spatialize` idiom. Thus, $\mathbb{E}$ represents an aggregation function, $\mathbb{L}$ a loss (error) function, e^* is the estimate and $\{x_k^*\}_m$ are the ESI samples. With these components, the precision function p^* can be implemented in `spatialize` with the following idiomatic structure:

```python
from spatialize.gs.esi.lossfunction import loss

@loss(E)
def p(x, y):
    return L(x, y)
```

Note that in this structure, neither e^* nor $\{x_k^*\}_m$ appears explicitly. This is because the `@loss` decorator internally transforms the function p to have the signature:

```python
def p(estimation, esi_samples):
    ...
```

The aim of this is that the design effort is put into the development of the function L, leaving `spatialize` to handle the internal processing of the more complex behaviour of p. In this way, for example, the precision function can be implemented very easily as:

```python
from spatialize.gs.esi.lossfunction import loss
from spatialize.gs.esi.agg_function import mean

@loss(mean)
def p_E(x, y):
    return (x - y) ** 2
```

In other words, when decorating the function `p_E` with `@loss`, giving it an aggregation function (`mean`, in this case), it becomes the implementation of a precision model. Once this has happened, it can be passed as an argument when calculating the precision of the result of an ESI estimation, as shown below:

```python
result = esi_griddata(...)
p = result.precision(p_E)
```

Two very common loss functions are included in this module:

- `mse_loss`: Implements the precision model.
- `mae_loss`: Implements the precision model given by

$$p^*_{\mathbb{E}} = \frac{1}{m}\sum_{k=1}^{m} |x^*_k - e^*_{\mathbb{E}}| \tag{3}$$

An example of a call for both is

```python
from spatialize.gs.esi.lossfunction import mse_loss

result = esi_griddata(...)
p = result.precision(mse_loss)
```

Where p is an array of dimension $d_1 \times d_2 = N_{x^*}$ in the case of gridded data, or simply an array of dimension N_{x^*}, for the non-gridded case.

In addition, versions without aggregation are included, i.e. where the aggregation function is `identity`. These are:

- `mse_cube`
- `mse_cube`

Recall that these last two are to leave the possibility to explore more complex aggregation functions to calculate accuracy. Both can be passed as arguments when calculating the precision, as shown below:

```python
from spatialize.gs.esi.lossfunction import mse_cube

result = esi_griddata(...)
p = result.precision(mse_cube)
```

Just note that p is no longer an array of dimensions $d_1 \times d_2 = N_{x^*}$, but an array of dimension $d_1 \times d_2 \times m$ (m = `n_partitions` in both `esi_griddata` and `esi_nongriddata`) in the case of gridded data (or an array of dimension $N_{x^*} \times m$, for the non-gridded case), where m is the number of ESI samples.

Finally, the module also includes:

- `class OperationalErrorLoss`: This is a class for creating functions (callable instances) that belong to a family of functions indexed by a given *dynamic range*.
  - **Constructor signature:**

    `class OperationalErrorLoss(dyn_range, use_cube)`
  - **Returns:** When the constructor of the class is called, it returns a function that implements the following precision model:

$$p^*_{\mathbb{E}} = \frac{1}{m \cdot d_r} \sum_{k=1}^{m} |x^*_k - e^*_{\mathbb{E}}| \quad (4)$$

Where d_r is an expected dynamic range for output errors, passed in the argument `dyn_range` (default value is `None`). If `dyn_range` is not passed in the argument, the dynamic range of the estimate ($e^*_{\mathbb{E}}$) is used. If `use_cube=True` (default value

False), then the returned function is decorated with the identity, having the same output behaviour as mse_cube or mae_cube.

An example of a call is as follows:

```
from spatialize.gs.esi.lossfunction import OperationalErrorLoss

result = esi_griddata(...)

op_error_loss = OperationalErrorLoss(100)

p = result.precision(op_error_loss)
```

References

[1] Jean-Paul Chilès and Nicolas Desassis. *Fifty Years of Kriging*, pages 589–612. Springer International Publishing, 2018.

[2] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[3] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(03):90–95, 2007.

[4] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[5] 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. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[6] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, pages 839–846, 1998.