HIGH PERFORMANCE GEOSTATISTICS LIBRARY

HPGL

User Guide

HPGL: High Perfomance Geostatistics Library

version 0.9.3

User Guide

2009

Table of contents

1. Understanding Basics

1.1. System Requirements

Using HPGL requires a Windows (32-bit) or Linux (32/64-bit) operating system with installed Python Software version 2.5 or later (you may download the latest Python version at http://www.python.org/download/).

1.2. Software Description

HPGL is a C++ / Python library that realize geostatistical algorithms. The algorithms are implemented via scripts in the Python language, thus enabling creation of the required geostatistical modeling scenarios.

Version **0.9.3** implements the following algorithms:

- Simple Kriging (SK)
- Ordinary Kriging (OK)
- Indicator Kriging (IK)
- Local Varying Mean Kriging (LVM Kriging)
- Simple CoKriging (Markov Models 1 & 2)

- Sequential Indicator Simulation (SIS)
- Corellogram Local Varying Mean SIS (CLVM SIS)
- Local Varying Mean SIS (LVM SIS)

- Sequential Gaussian Simulation (SGS)
- Local Varying Mean SGS (LVM SGS)

- Truncated Gaussian Simulation (GTSIM)*
  * in the Python script collection

The attributes are set across an *ijk* space, meaning that all parameters (e.g. variogram or ellipsoid radiuses) are set in grid cells.

The following data import/export formats are currently supported:
- Text file with property

1.3. Used components

- modified version of GsTL (original version can be downloaded at https://sourceforge.net/projects/gstl, and modified one – from HPGL repository).
- TNT (Template Numerical Toolkit) – (can be downloaded from http://math.nist.gov/tnt/overview.html)

1.4. Installation

1.4.1. Windows

Install by running the file **`HPGL-X.Y.Z.win32.exe`**

MS Windows installations will also require installing Microsoft Visual C++ 2005 SP1 Redistributable Package (it can be downloaded from http://www.microsoft.com/downloads/details.aspx?familyid=200B2FD9-AE1A-4A14-984D-389C36F85647&displaylang=en).

1.4.2. Linux

1. Ubuntu 8.10 (32-bit):
   Install package **`hpgl_x32_X.Y.Z_ubuntu_8.10.deb`**.

2. Ubuntu 8.10 (64-bit):
   Install package **`hpgl_x64_X.Y.Z_ubuntu_8.10.deb`**.

All the required additional packages (e.g. boost libraries) will be installed by dependencies from the repository.

So far HPGL has binary packages only for Ubuntu 8.10. However, if you want to compile the project under another Linux system (or to create a package), feel free to contact the authors. The compilation instructions will presently appear on the site.

2. Core Features

2.1. Library import

Every Python script using HPGL functions must be started with import command:

```
from geo import *
```

2.2. Creating an IJK grid

An IJK (Cartesian) grid is created with the `SugarboxGrid` function:

```
grid_object = SugarboxGrid(I, J, K)
```

This command will create a grid object of dimensions *i*, *j*, *k*.

Example:

```
my_griddy = SugarboxGrid(42, 42, 10)
```

2.3. Importing data from text files

The data in the text files to import from must be in the following format:

```
-- comment (will be ignored)

PROPERTY_NAME
-- data in k, j, i order (by slices)
0
1
0
...
/
```

Importing properties from text files is implemented in two functions:

- `load_ind_property` – for indicator values;
- `load_cont_property` – for continuous values.

```
property_object = load_cont_property(filename, undefined_value)

property_object = load_ind_property(filename, undefined_value,
[indicators])
```

Both these commands will create an object (`property_object`) that will contain the property from file `filename`. Cells with values equal to `undefined_value` will be considered empty (undefined).

The last argument in the `load_ind_property` function is the codes of the indicators contained in the file.

Please notice: after loading from file, indicators are renumbered to 0,1,2… like they are counted in `indicators`.

Example:

```
my_cont_property = load_cont_property("d:\CONT.INC", -99)

my_ind_property = load_ind_property("d:\IND.INC", -99, [0,1])
```

2.4. Exporting data to text files

A property is written to a text file by `write_property`:

```
write_property(prop_object, filename, prop_name, undefined_value)
```

This commands will create the file named `filename`, that will contain the property `prop_name` extracted from object `prop_object`. Empty cells (if any) will be written as `undefined_value`. Indicator values in saved property is defined by `indicators`. If `indicators` not set, indicator values in saved property will be 0,1,2,…

The property type (indicator / continuous) is saved automatically.

Example:

```
write_property(my_cont_prop, "MY_CON_PROP.INC", "PROPCON", -99)

write_property(my_ind_prop, "MY_IND_PROP.INC", "PROP_IND", -99,
[indicators])
```

2.5. Releasing data from memory

When a property is no longer needed, it can be deleted to free up computer memory. This is done by the `del` command defined as

```
del(prop_object)
```

Example:

```
del(my_prop)
```

2.6. Vertical Proportion Curve (VPC) property calculation

A VPC property is one that contains the mean value of a property for each of the available vertical layers (slices). It may be used as a container of mean values in variable-mean algorithms (LVM SIS, LVM Corellogram SIS, LVM Kriging, etc.)

A VPC property is computed, given the pre-loaded property `prop`, as shown below:

```
vpc_cube = calc_vpc(prop, grid, marginal_probs)
```

where `grid` is the grid on which the calculation is run, and `marginal_probs` are the mean values for each indicator (these values will be set in places where the vertical layer is empty).

You should bear in mind that the property `vpc_cube` consists of means for each indicator; so, e.g., if there are two indicators, the property must be addressed as `vpc_cube[0]` and `vpc_cube[1]`.

Saving a VPC property to a file is performed by the function `write_mean_data`.

A cube created by `calc_vpc` can be used in all algorithms with a varying mean option as a `mean_data` parameter.

Example:

```
grid = SugarboxGrid(55, 52, 1)
prop = load_ind_property("IK_HARD_DATA.INC", -99, [0,1])

vpc_cube = calc_vpc( prop, grid, [0.8, 0.2] )


write_mean_data(vpc_cube[0], "prob_0.inc", "VPC_PROB_0");

write_mean_data(vpc_cube[1], "prob_1.inc", "VPC_PROB_1");
```

2.7. Working with Properties

Every property created by any algorithm is an *object* having a number of methods.

1) **Creating a void property of defined type and size**

There are two ways to create a property: make it 'from scratch' (the resulting property will be full of undefined values) or copy it from an existing one.

There are 2 functions to make property from scratch:

- for *continious* property:
  `hpgl.cont_property_array(`**`size`**`)`

- for *indicator* property:
  `hpgl.byte_property_array(`**`size`**`, [`**`indicators`**`])`

The copying of an existing property is performed by the `clone()` function:

```
property_copy = property.clone()
```

Example:

```
prop_cont = hpgl.cont_property_array(100*100*20)
prop_ind = hpgl.byte_property_array(100*100*20, [0,1])

prop_cont_copy = prop_cont.clone()
prop_ind_copy = prop_ind.clone()
```

Please notice: To get the property type (continuous or indicator), you can use the following code:

```
if type(property) is hpgl.cont_property_array:
     # actions for continious property
else:
     # actions for indicator property
```

WARNING: The statement below:

```
property_1 = property_2
```

will not make a copy of `property_2`! Instead, `property_1` will be set as a **pointer** to `property_2`. To copy a property, use the `clone()` function outlined above.

2) **Operating Property values using *get_at/set_at***

Property values are defined by index, from 0 to N-1, where N is the property size. A property is indexed by *k*, *j*, and *i*, respectively (by vertical slices).

The functions discussed in this section are useful for fast and low-memory-cost property calculations **without getting the actual locations of the values on the grid**.

All property values can be *informed* (imparted with a certain value) or *undefined* (set without a value, just indexed). To get the informed/undefined value status, use the `is_informed()` function:

```
result = property.is_informed(index)
```

If value is informed `result` will be `True`, otherwise - `False`.

If you need to take to account values coordinates in *ijk* space, use a function to convert propery in 3D numpy-array (described below).

Getting value by `index` from `property` is obtained by `get_at()` function.

Please notice that before using the `get_at` function, you must check the value by the `is_informed()` function:

```
if(property.is_informed(index) == True):
     value = property.get_at(index)
```

Changing an existing or undefined `value` at `index` in `property` is performed by the `set_at()` function:

```
property.set_at(index, value)
```

After executing, point **index** will be informed.

Example:

```
If(property.is_informed(10) == True):
     value = property.get_at(10)
     print 'property value at 10 is', value
else:
     property.set_at(10, 1000)
     print 'property value at 10 is 1000 now'
```

3) **Operating Property values using *NumPy* arrays**

Will be included in the next release.

3. Using the Algorithms

3.1. Simple Kriging

Simple Kriging is implemented in the function `simple_kriging`:

```
def simple_kriging(
        prop,                   # property with initial values (hard data)
        grid,                   # the grid in which SK is performed
        radiuses,               # search ellipsoid radiuses
        max_neighbours,         # maximum interpolation points
        covariance_type,        # variogram type:
                                # it may take different values:
                                # covariance.spherical, covariance.exponential
                                # and covariance.gaussian
        ranges,                 # variogram ranges (radiuses)
        sill,                   # variogram sill value
        nugget=None,            # nugget-effect value
        angles=None,            # variogram rotation angles
        mean=None               # mean value
                                # if undefined, it will be set automatically by
                                # the initial data
)
```

Example:

```
grid = SugarboxGrid(55, 52, 1)
prop = load_cont_property("SK_HARD_DATA.INC", -99)

prop_result = simple_kriging(prop, grid,
          radiuses = (20, 20, 20),
          max_neighbours = 12,
          covariance_type = covariance.exponential,
          ranges = (10, 10, 10),
          sill = 1,
          mean = 1.6)
```

```
write_property(prop_result, "R_SK.INC", "SK_RESULT", -99)
del(prop_result)
```

3.2. Ordinary Kriging

Ordinary Kriging is implemented in the function `ordinary_kriging`:

```
def ordinary_kriging(
        prop,               # property with initial values (hard data)
        grid,               # the grid in which OK is performed
        radiuses,           # search ellipsoid radiuses
        max_neighbours,     # maximum interpolation points
        covariance_type,    # variogram type:
                            # it may take different values:
                            # covariance.spherical, covariance.exponential
                            # and covariance.gaussian
        ranges,             # variogram ranges (radiuses)
        sill,               # variogram sill value
        nugget=None,        # nugget-effect value
        angles=None,        # variogram rotation angles
)
```

Example:

```
grid = SugarboxGrid(55, 52, 1)
prop = load_cont_property("OK_HARD_DATA.INC", -99)

prop_result = ordinary_kriging(prop, grid,
     radiuses = (20, 20, 20),
     max_neighbours = 12,
     covariance_type = covariance.exponential,
     ranges = (10, 10, 10),
     sill = 1)

write_property(prop_result, "R_OK.INC", "OK_RESULT", -99)
del(prop_result)
```

3.3. Indicator Kriging

Before calling the `indicator_kriging` function, a structure of parameters must be created as shown below:

```
ik_data =    [

                # Variogram parameteres for 1st indicator:

                {
                "cov_type": cov_type,            # variogram type:

                                                 # it may take different values:
                        # covariance.spherical, covariance.exponential
                                                 # and covariance.gaussian

                "ranges": (R1, R2, R3),          # variogram ranges (radiuses)

                'sill': sill,                    # variogram sill value

                "radiuses": (SR1, SR2, SR3),     # search ellipsoid radiuses

            "max_neighbours": neigh_count,       # maximum interpolation points

                "marginal_prob": marg_prob,      # indicator a priori probability

                "value": 0                       # indicator value (code)
                },

                # Variogram paramteres for 2nd indicator:

                {
                "cov_type": cov_type,            # variogram type:

                                                 # it may take different values:
                                                 #          covariance.spherical,
    covariance.exponential
                                                 # and covariance.gaussian

                "ranges": (R1, R2, R3),          # variogram ranges (radiuses)
```

'sill': **sill**, # variogram sill value

"radiuses": (**SR1**, **SR2**, **SR3**), # search ellipsoid radiuses

"max_neighbours": **neigh_count**, # maximum interpolation points

"marginal_prob": **marg_prob**, # indicator a priori probability

"value": **1** # indicator value (code)

}

]

A variogram is required for each indicator variable.

Please notice: If only two indicators are used, *Median IK* will be performed automatically.

The parameters in the structure being assigned, `indicator_kriging` can now be called as follows:

def indicator_kriging

(

ik_prop, # algorithm parameters structure

grid, # the grid on which Indicator Kriging is performed

ik_data, # property with initial values (hard data)

)

Example:

```
grid = SugarboxGrid(55, 52, 1)
ik_prop = load_ind_property("IK_HARD_DATA.INC", -99, [0,1])

ik_data =  [    {
                "cov_type": 0,
                "ranges": (10, 10, 10),
                'sill': 0.4,
                "radiuses": (10, 10, 10),
                "max_neighbours": 12,
                "marginal_prob": 0.5,
                "value": 0
           },
           {
                "cov_type": 0,
                "ranges": (10, 10, 10),
```

```
                "sill": 0.4,
                "radiuses": (10, 10, 10),
                "max_neighbours": 12,
                "marginal_prob": 0.5,
                "value": 1
            }]

ik_result = indicator_kriging(ik_prop, grid, ik_data)
write_property(ik_result, "RESULT_IK.INC", "PROP_IK", -99)
```

3.4. LVM Kriging (Local Varying Mean)

Kriging with Local Varying Means (LVM) is implemented in the function `lvm_kriging`:

```
def lvm_kriging
(
    lvm_prop,           # initial property values (hard data)

    grid,               # the grid in which lvm kriging is performed
    mean_data,          # property with LVM values
    radiuses,           # search ellipsoid radiuses
    max_neighbours,     # maximum interpolation points
    covariance_type,    # variogram type:
                        # it may take different values:
                        # covariance.spherical, covariance.exponential
                                # and covariance.gaussian

    ranges,             # variogram ranges (radiuses)
    sill,               # variogram sill value
    nugget=None,                # nugget effect value
    angles=None,                # variogram rotation angles
)
```

The LVM value property must be loaded with the `load_mean_data` function as shown below:

```
mean_data_obj = load_mean_data(filename)
```

where `filename` is the LVM data file path;
and `mean_data_obj` is the data object to use in the algorithm.

Example:

```
grid = SugarboxGrid(55, 52, 1)

mean_data = load_mean_data("cube_with_local_means.inc")

lvm_prop = load_ind_property("LVM_PROP.INC", [0,1])

prop_lvm = lvm_kriging(lvm_prop, grid, mean_data,
          radiuses = (20, 20, 20),
          max_neighbours = 12,
          covariance_type = covariance.exponential,
          ranges = (10, 10, 10),
          sill = 1)

write_property(prop_lvm, "lvm_result.inc", "lvm_kriging", -99)

del(mean_data)
del(prop_lvm)
```

3.5. Sequential Indicator Simulation (SIS) (LVM, Corellogram)

The SIS parameters structure is identical to the Indicator Kriging's desctibed above.

The algorithm is executed by the `sis_simulation` function:

```
def sis_simulation(

        ik_prop,          # algorithm parameters structure

        grid,             # grid on which SIS is performed

        ik_data,          # initial property data (hard data)

    seed,                 # random seed (a stochastic realization number)

    mask = False,         # modeling region -
                          # in case not all points need to be simulated

                          # mask must be an indicator type property with
```

```
                    # 1 (ones) for points to be simulated, and 0 (zeros)
                    # for the ones to leave out

                    # if mask = False, all points will be simulated

          mean_data = None,      # the LVM property for LVM SIS

          use_corellogram = False

                    # Correlogram LVM SIS (only with mean data)

                    # True — use Correlogram SIS
                    # False — use Classic LVM SIS
)
```

Example:

```
grid = SugarboxGrid(55, 52, 1)
sis_prop = load_ind_property("SIS_HARD_DATA.INC", -99, [0,1])

sis_data =  [  {
               "cov_type": 0,
               "ranges": (10, 10, 10),
               'sill': 0.4,
               "radiuses": (10, 10, 10),
               "max_neighbours": 12,
               "marginal_prob": 0.5,
               "value": 0
          },
          {
               "cov_type": 0,
               "ranges": (10, 10, 10),
               "sill": 0.4,
               "radiuses": (10, 10, 10),
               "max_neighbours": 12,
               "marginal_prob": 0.5,
               "value": 1
          }]

sis_result = sis_simulation(sis_prop, grid, sis_data,
seed=3241347,  use_corellogram = False)

write_property(sis_result, "RESULT_SIS.INC", "PROP_SIS", -99)
```

3.6. Sequential Gaussian Simulation (SGS, LVM SGS)

First, create a parameters structure:

sgs_params = {

"radiuses": (**SR1**, **SR2**, **SR3**),

search ellipsoid radiuses (max, med, min)

"max_neighbours": **max_neigh**,

maximum interpolation points

"covariance_type": **cov_model**,

variogram type:

it may take different values:
covariance.spherical, covariance.exponential
and covariance.gaussian

"ranges": (**R1**, **R2**, **R3**),

variogram ranges (max, med, min)

"sill": **sill**,

variogram sill value

"kriging_type": **krig_type**,

kriging type

“sk” - for Simple Kriging (default)

“ok” - for Ordinary Kriging

"mean": **mean,**

mean for Simple Kriging

"mean_data": **mean_data_obj**

lvm values property

}

Next, perform the algorithm:

```
sgs_result = sgs_simulation(property_obj, grid_obj, seed, mask,
                              **sgs_params)
```

The algorithm function takes the following arguments:

- `property_obj` — initial property data (hard_data);
- `grid_obj` — the simulation grid;
- `seed` — a stochastic realization seed value;
- `sgs_params` — SGS parameters structure
- `mask` – the region for modeling (in case not all points need to be simulated). The mask must be an indicator-type property, with 1 (ones) for points to be simulated, and 0 (zeros) for the ones to leave out. If **False**, all points will be simulated.

<u>Example:</u>

```
grid = SugarboxGrid(55, 52, 1)
prop = load_cont_property("SGS_HARD_DATA.INC", -99)

sgs_params = {
"radiuses": (20, 20, 20),
"max_neighbours": 12,
"covariance_type": covariance.exponential,
"ranges": (10, 10, 10),
"sill": 0.4,
"kriging_type": "sk",
"mean": 0.3}

sgs_result = sgs_simulation(prop_con, grid, seed=3439275,
**sgs_params)

write_property(sgs_result, "RSGS.INC", "PROP_SGS", -99)
```

<u>Example(LVM):</u>

```
grid = SugarboxGrid(55, 52, 1)

prop = load_cont_property("SGS_HARD_DATA.INC", -99)
mean_data = load_mean_data("SGS_MEAN_DATA.INC")

lvm_sgs_params = {
     "radiuses": (20, 20, 20),
     "max_neighbours": 12,
     "covariance_type": covariance.exponential,
     "ranges": (10, 10, 10),
     "sill": 0.4,
     "mean_data": mean_data}
```

```
    sgs_lvm = sgs_simulation(prop, grid, seed=3439275,
**lvm_sgs_params)

    write_property(sgs_lvm, "SGS_LVM_RESULT.INC", "SGS_LVM", -99)

    del(sgs_lvm)
```

4. Python Scripts Collection

The Python Script Collection provides examples, some additional algorithms, property operation examples, etc.

To work with the *array* data type, you will need to install *NumPy* & *SciPy* Python libraries (they can be downloaded from http://www.scipy.org/Download or installed from the repository on Linux systems).

For the visualization of maps, histograms, plots, etc., you will need to install *matplotlib* (which can be downloaded from http://sourceforge.net/projects/matplotlib or installed from the repository on Linux systems).

Scripts list is included in `scripts_list.txt` file within the collection.

Contact the Authors:

Vladimir Savichev

Andrey Bezrukov

Artur Mukharlyamov

Konstantin Barsky

Dina Nasibullina

Feel free to ask questions at: hpgl-support-eng@lists.sourceforge.net.

Modification History

HPGL 0.9.3 - *06/04/2009*

First open release.