HIGH PERFORMANCE GEOSTATISTICS LIBRARY

HPGL

User Guide

HPGL

High
Performance
Geostatistics
Library

version 0.9.9 BSD

User Guide

2010

Contents

1. Understanding the Basics

1.1. Software Description

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

Version **0.9.9 BSD** 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)
- Correlogram 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.

Kriging algorithms supports parallel processing, see 2.7 to learn how to set up the number of threads.

The following data import/export formats are currently supported:

- Eclipse Property text file;
- GSLIB property text file.

HPGL properties are stored as NumPy Arrays (see 2.3 for details).

1.2. System Requirements and Installation

Using HPGL requires a Windows (32-bit) or Linux (32/64-bit) operating system with installed Python version 2.5 or higher, as well as NumPy/SciPy python packages installed (for the corresponding Python version).

1.2.1. Microsoft Windows

MS Windows installation requires the presence of Microsoft Visual C++ 2005 SP1 Redistributable Package (it can be downloaded from `http://www.microsoft.com/downloads/details.aspx?familyid=2051A0C1-C9B5-4B0A-A8F5-770A549FD78C&displaylang=en`).

WARNING! The Redistributable Package must be of revision date 7/28/2009 or later (after the ATL security update).

HPGL installation is performed by running the file **`HPGL-X.Y.Z-BSD-[py2.5/py2.6].win32.exe`** (for the corresponding Python version).

1.2.2. Ubuntu/Debian Linux (.deb-based)

Install package **`hpgl_X.Y.Z-BSD-[x32/x64].deb`** (corresponding to the operation system's architecture). Using HPGL also requires the Boost Libraries to be installed.

1.2.3. Other Systems

So far HPGL has binary packages only for Ubuntu Linux and Windows. However, if you want to compile the project under another Linux system (or to create a package), feel free to contact the authors.

1.3. Components Used

- TNT (Template Numerical Toolkit) – (can be downloaded from http://math.nist.gov/tnt/overview.html);
- Boost Libraries (i.e. **boost::python**).

2. Core Features

2.1. Library Import

Every HPGL Python script must be started with the import **`geo_bsd`** module command:

```
from geo_bsd import *
```

HPGL also includes two sub-modules **`geo_bsd.routines`** with additional property-related algorithms: VPC (Vertical Proportion Curve) and moving average calculations, GSLIB file format support etc., and **`geo_bsd.cvariogram`** for sample variogram calculation.

If you want to use these sub-modules, the Python script must be started with:

```
from geo_bsd.routines import *
from geo_bsd.cvariogram import *
```

For detailed information about sub-modules, see Ch. 4.

2.2. Creating an IJK Grid

Every HPGL geostatistical algorithm requires a Cartesian Grid object. An IJK (Cartesian) grid can be 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. Properties

All HPGL properties must be objects of the two classes: `ContProperty` (for continuous data) or `IndProperty` (for categorical data).

a) Continuous property:

```
cont_property = ContProperty(array_prop, array_mask)
```

where

- **`array_prop`** is a 3D NumPy-array (*float32* type) with property data;

- **array_mask** is a 3D NumPy-array (*uint8* type), which defines **array_prop** points with a value (*array_informed = 1*), and **array_prop** points without value (*array_informed = 0*).

b) Categorical property:

```
ind_property = IndProperty(array_prop, array_mask,
                           indicators_number)
```

where

- **array_prop** is a 3D NumPy-array (*uint8* type) with categorical property data. Categorical indicators must be named from 0 up to *max* (0,1,2,3…);
- **array_mask** is a 3D NumPy-array (*uint8* type), which defines **array_prop** points with a value (*array_informed = 1*), and **array_prop** points without value (*array_informed = 0*).
- **indicators_number** is the number of categorical indicators in **array_prop**.

Note: 2D or 1D properties must be created as 3D ones:

```
a = zeros((10,10, 1)) # 2D 10x10 property
a = zeros((10, 1, 1)) # 1D 10 property
```

WARNING! NumPy arrays must use the FORTRAN data storage order. This can be achieved with the following:

- creating a new array:

```
a = array([], order='F')
```

- changing an existing non-Fortran order array:

```
a = require(a, requirements='F')
```

If an HPGL input array will be non-FORTRAN, it will be converted to the FORTRAN type automatically; you need to keep in mind that *all resulted properties will be returned as FORTRAN order arrays*.

More information about FORTRAN order arrays can be addressed here: http://www.ibiblio.org/pub/languages/fortran/ch2-6.html.

2.4. Eclipse Property File Format

HPGL supports reading from and writing to Eclipse property files.

Eclipse property files must be in the following format:

```
-- comment (will be ignored)

PROPERTY_NAME
0
1
0
...
/
```

Values will be read in the order defined in the file.

2.4.1. Reading Eclipse Property files

Reading properties from Eclipse property text files is implemented in two functions:

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

```
prop = load_cont_property(filename, undefined_value, size)

prop = load_ind_property(filename, undefined_value,
[indicators], size)
```

These commands will create an object (**`prop`**) of the corresponding class (**`ContProperty`** or **`IndProperty`**) that will contain a property from the file **`filename`**. Cells with values equal to **`undefined_value`** will be considered empty (*undefined*), and **`array_informed`** for these cells will be set to **0**.

Direct access to data and mask NumPy arrays can be achieved by indexing the property object: `prop[0]` will be a pointer to the data array, and `prop[1]` will be a pointer to the mask array.

The **`[indicators]`** argument in the **`load_ind_property`** function is a Python tuple with indicator codes contained in the file.

The last argument (**`size`**) is a Python tuple with the grid size in cells **`i,j,k`**:

```
size = (i,j,k)
```

WARNING! After importing data from the file, the indicators will be renumbered to 0,1,2… like in **`indicators`**.

<u>Example:</u>

```
size = (50, 50, 100)
cont_property = load_cont_property("d:\CONT.INC", -99, size)
```

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

2.4.2. Writing Eclipse Property Files

Properties can be written to an Eclipse property file using the **write_property()** function:

```
write_property(prop_object, filename, prop_name, undefined_value,
                    indicator_values=[])
```

This command will create a text file with name **filename**, which will contain the property **prop_name** extracted from the object **prop_object**. Empty cells (if any) will be written as **undefined_value**. For indicator properties, the indicator values are defined in **indicator_values**. If **indicator_values** is not defined, the indicators in the saved property will be 0,1,2,...

Example:

```
write_property(cont_prop, "CON_PROP.INC", "PROPCON", -99)
write_property(i_prop, "INDP.INC", "PROP_IND", -99, [0,1])
```

2.5. GSLIB Files

A detailed description of the GSLIB file format can be found at http://www.gslib.com/gslib_help/format.html. All GSLIB-related functions are included in the `geo_bsd.routines` sub-module, so you need to import it before using GSLIB files:

```
from geo_bsd.routines import *
```

2.5.1. Reading from GSLIB Files

Reading properties from a GSLIB file is implemented in the **LoadGslibFile()** function:

```
dict_gslib = LoadGslibFile(filename)
```

where **dict_gslib** is a Python dictionary with data from file **filename** (the dictionary items will be NumPy-array properties from the file).

A property with the name **property_1** can be accessed using the following syntax:

```
dict_gslib[“property_1”]
```

2.5.2. Writing to GSLIB Files

An HPGL property can be written into a GSLIB file by the **SaveGSLIBCubes()** function:

```
SaveGSLIBCubes(dict_gslib, filename, caption, Format = "%d")
```

where **filename** is the GSLIB file name;
dict_gslib is the Python dictionary with the properties in the form of NumPy-arrays;
caption is the caption of the GSLIB file.

A detailed description of Python dictionaries can be found in Python documentation, for example, here:
http://docs.python.org/tutorial/datastructures.html#dictionaries

2.5.3. Writing to GSLIB Files (C++)

There is another GSLIB property write function called `write_gslib_property()` present in HPGL. The parameters of this function are identical to those of the `write_property()` function for Eclipse property files:

```
write_gslib_property(prop_object, filename, prop_name,
        undefined_value, indicator_values=[])
```

This command will create a file named **filename** which will contain the property named **prop_name** extracted from the object **prop_object**. Empty cells (if any) will be written as **undefined_value**. For indicator property, indicator values are defined by **indicator_values**. If **indicator_values** is not defined, the indicators in the saved property will be written as 0,1,2,...

This function is much faster than `SaveGSLIBCubes()`, but it can be used to store only one property at a time. Multiple properties defined as dictionaries can be stored using the `SaveGSLIBCubes()` function.

Example:

```
write_gslib_property(cont_prop, "CON_PROP.INC", "PROPCON", -99)
write_gslib_property(i_prop, "INDP.INC", "PROP_IND", -99, [0,1])
```

2.6. Covariance (Variogram) Object

All HPGL geostatistical algorithms use a unified type of the covariance (variogram) function. A covariance (variogram) object must be created as `CovarianceModel`:

```
cov = CovarianceModel(
      type = 0,
      ranges=(0,0,0),
      angles=(0,0,0),
      sill=1.0,
      nugget=0.0)
```

where

`type` is the variogram type:

0 – spherical, **1** – exponential, **2** – Gaussian;

`ranges` are the variogram ellipsoid ranges (0^o, 90^o, vertical);

`angles` are the variogram ellipsoid angles;

`sill` is the sill value of the variogram;

`nugget` is the nugget-effect value.

Covariance model objects can be used in all HPGL geostatistical algorithms.

2.7. Threading Parallel Algorithms

The number of threads for parallel algorithms (so far, only for Kriging) can be set/modified with the `set_thread_num()` function:

```
set_thread_num(th_num)
```

where `th_num` is the number of threads to be allocated.

Note: A 'rule of thumb' for threading is to set the number corresponding to the number of CPUs (or cores) operating on the system.

To get the current number of threads, calle the function `get_thread_num`:

```
current_th_num = get_thread_num()
```

2.8. Releasing Data from Memory

When a property is no longer needed, it should be deleted to free system memory. This is done by the **`del()`** command defined as

```
del(prop_object)
```

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
        cov_model,          # covariance (variogram) object (see 2.6)
        mean=None           # mean value
                                # if None, it will be calculated automatically
                                # from the initial data
)
```

Example:

```
size = (55, 52, 100)
grid = SugarboxGrid(55, 52, 100)
prop = load_cont_property("HARD_DATA.INC", -99, size )
cov_krig = CovarianceModel(type=1, ranges=(10,10,10), sill=1)

prop_result = simple_kriging(prop, grid,
          radiuses = (20, 20, 20),
          max_neighbours = 12,
          cov_model = cov_krig,
          mean = 1.6)

write_property(prop_result, "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
        cov_model,          # covariance (variogram) object (see 2.6)
)
```

Example:

```
size = (55, 52, 100)
grid = SugarboxGrid(55, 52, 100)
prop = load_cont_property("HARD_DATA.INC", -99, size )
cov_krig = CovarianceModel(type=1, ranges=(10,10,10), sill=1)

prop_result = ordinary_kriging(prop, grid,
     radiuses = (20, 20, 20),
     max_neighbours = 12,
     cov_model = cov_krig)

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

3.3. Indicator Kriging

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

```
data =  [

            # Variogram parameters for 0 indicator:

        {
            "cov_model": cov0            # covariance (variogram) object (see 2.6)
            "radiuses": (SR1, SR2, SR3),         # search ellipsoid radiuses

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

            # Variogram parameters for 1 indicator:

        {
            "cov_model": cov1            # covariance (variogram) object (see 2.6)
            "radiuses": (SR1, SR2, SR3),         # search ellipsoid radiuses

            "max_neighbours": neigh_count,     # maximum interpolation points
        }
]
```

A variogram is required for each indicator variable.

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

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

        data,           # property with initial values (hard data)

    marginal_probs      # Python tuple with marginal probabilities for each indicator

)
```

Example:

```
size = (55, 52, 100)
grid = SugarboxGrid(55, 52, 100)
prop = load_ind_property("HARDDATA.INC", -99, [0,1], size)

cov1 = CovarianceModel(type=1, ranges=(10,10,10), sill=1)

data =  [ {
                "cov_model": cov1,
                "radiuses": (20, 20, 20),
                "max_neighbours": 12,
          },
          {
                "cov_model": cov1,
                "radiuses": (20, 20, 20),
                "max_neighbours": 12,
          }]

ik_result = indicator_kriging(prop, grid, data, (0.8, 0.2))
write_property(ik_result, "RESIK.INC", "PROP_IK", -99, [0,1])
```

3.4. LVM Kriging (Local Varying Mean)

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

```
def lvm_kriging

(

    prop,               # initial property values (hard data)

    grid,               # the grid in which lvm kriging is performed
```

mean_data, # property with LVM values (must be *float32* NumPy array)

radiuses, # search ellipsoid radiuses

max_neighbours, # maximum interpolation points

cov_model # covariance (variogram) object (see 2.6)

)

Example:

```
grid = SugarboxGrid(55, 52, 100)
size = (55, 52, 100)
mean_data = load_cont_property("cube_local_means.inc", size)[0]

cov1 = CovarianceModel(type=1, ranges=(10,10,10), sill=1)

lvm_prop = load_cont_property("LVM.INC", -99, size)

prop_lvm = lvm_kriging(lvm_prop, grid, mean_data,
          radiuses = (20, 20, 20),
          max_neighbours = 12,
          cov_model = cov1)

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

del(mean_data)
del(prop_lvm)
```

3.5. Sequential Indicator Simulation (SIS)

The SIS parameters structure is identical to Indicator Kriging described above.

The algorithm is executed by the **sis_simulation** function:

def sis_simulation(

prop, # initial property data (hard data)

grid, # grid on which SIS is performed

data, # algorithm parameters structure

seed, # random seed (a stochastic realization number)

marginal_probs, # if Python tuple with marginal probabilities
for each indicator, SIS will be performed;
if Python tuple with NumPy-arrays (probabilities cubes)
SIS LVM will be performed.

use_correlogram = **True,**

```
                              # Type of LVM SIS (only if mean data defined as
                              # probabilities cubes)
                              # True — use Correlogram SIS
                              # False — use Classic LVM SIS

    mask = None,             # modeling region -
                              # in case not all points need to be simulated

                              # mask must be an uint8 NumPy array with
                              # 1 (ones) for points to be simulated, and 0 (zeros)
                              # for the ones to leave out

                              # if mask = None, all points will be simulated
)
```

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

Example:

```
size = (55, 52, 100)
grid = SugarboxGrid(55, 52, 100)
sis_prop = load_ind_property("HARD.INC", -99, [0,1], size)

cov1 = CovarianceModel(type=1, ranges=(10,10,10), sill=1)


sis_data =  [  {
                "cov_model": cov1,
                "radiuses": (20, 20, 20),
                "max_neighbours": 12,
           },
           {
                "cov_model": cov1,
                "radiuses": (20, 20, 20),
                "max_neighbours": 12,
           }]

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

write_property(sis_result, "RESSIS.INC", "P_SIS", -99, [0,1])
```

3.6. Sequential Gaussian Simulation (SGS)

SGS is implemented in the function `sgs_simulation`:

```
def sgs_simulation(
    prop,                  # initial property data (hard data)
    grid,                  # grid on which SGS is performed

    radiuses,              # search ellipsoid radiuses


    max_neighbours,        # maximum interpolation points


    cov_model,             # covariance (variogram) object (see 2.6)
    seed,                  # random seed (a stochastic realization number)

    kriging_type = "sk",   # Kriging type
                           # sk – Simple Kriging
                           # ok – Ordinary Kriging
                           # ignored for SGS LVM

    mean = None,           # modeling property mean value
                           # if number, SGS will be performed
                           # if float32 NumPy array – SGS LVM will be performed

    use_harddata = True,
                   # if False, initial property data will be ignored, and unconditional
                   # SGS with histogram from cdf_data will be performed

    cdf_data = None,
                   # CdfData class object, which defines CDF used for modeling
                   # can be created:
                   # 1. by calc_cdf(prop), function, where prop is a NumPy-ndarray
                   # 2. as CdfData(values, probs), where values are property cdf values
                   # and probs are the corresponding cumulative probabilities

    mask = None,          # modeling region -
                          # in case not all points need to be simulated

                          # mask must be an uint8 NumPy array with
                          # 1 (ones) for points to be simulated, and 0 (zeros)
                          # for the ones to leave out

                          # if mask = None, all points will be simulated
)
```

Example:

```
size = (55, 52, 100)
grid = SugarboxGrid(55, 52, 100)
prop = load_cont_property("SGS_HARD_DATA.INC", -99, size)
```

```
cov1 = CovarianceModel(type=1, ranges=(10,10,10), sill=1)

sgs_result = sgs_simulation(prop, grid,
      radiuses = (20,20,20),
           max_neighbours = 12,
           cov_model = cov1,
           seed=3439275)

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

Example(LVM):

```
grid = SugarboxGrid(55, 52, 100)
size = (55, 52, 100)

prop = load_cont_property("HARD_DATA.INC", -99, size )
mean_data = load_cont_property("MEAN.INC", -99, size )[0]

sgs_lvm_result = sgs_result = sgs_simulation(prop, grid,
      radiuses = (20,20,20),
           max_neighbours = 12,
           cov_model = cov1,
           seed=3439275,
           mean = mean_data)

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

del(sgs_lvm)
```

4. Sub-Modules

4.1. geo_bsd.routines

The `geo_bsd.routines` sub-module has many additional functions to work with HPGL properties.

4.1.1. Mean calculation

a) `CalcMean` – returns the mean value for the NumPy-array `Cube` calculated on the defined (`Mask = 1`) cells:

```
mean = CalcMean(Cube, Mask)
```

b) **`CalcMarginalProbsIndicator`** – returns a NumPy-array with proportions (marginal probabilities) of indicators in the array **`Cube`**, for each indicator in **`Indicators`**, calculated on the defined (`Mask = 1`) cells:

```
MProbs = CalcMarginalProbsIndicator(Cube, Mask, Indicators)
```

4.1.2. VPC (Vertical Proportion Curve) Calculation

a) **`CalcVPC`** – returns a NumPy-array with VPC (*Vertical Proportion Curve*) – mean values of vertical slices for the NumPy-array `Cube`, calculated on the defined (`Mask = 1`) cells:

```
VPC = CalcVPC(Cube, Mask, MarginalMean)
```

`MarginalMean` must be the mean value for the property defined in **`Cube`**. This value will be set in VPC for slices without defined (`Mask = 1`) cells.

b) **`CalcVPCsIndicator`** – returns a Python list with NumPy-arrays VPC (*Vertical Proportion Curve*) – means of vertical slices for the NumPy-array `Cube` for each indicator defined in `Indicators`, calculated on the defined (`Mask = 1`) cells:

```
Result = CalcVPCsIndicator(Cube, Mask, Indicators,
                           MarginalProbs)
```

`MarginalProbs` must be the means (marginal probabilities) for each of the indicators. These values will be set in VPC for slices without defined (`Mask = 1`) cells.

c) **`CubeFromVPC`** – creates a 3D NumPy-array of shape **`NX`**, **`NY`**, `len(`**`VPC`**`)`, filled with **`VPC`** values for each of the vertical slices.

`VPC_Cube = CubeFromVPC(VPC, NX, NY)`

VPC_Cube array can be used as mean data for continuous Local Varying Mean algorithms (SGS LVM, LVM Kriging). This function must be used in couple with **CalcVPC**.

d) **CubesFromVPCs** – creates a Python list with 3D NumPy-arrays shaped as **NX, NY,** len(**VPC**), filled with mean values for each of the vertical slices.

`VPC_Cubes = CubesFromVPCs(VPCs, NX, NY)`

VPC_Cubes can be used as mean data for indicator algorithms with Local Varying Mean (SIS LVM). This function must be used in couple with `CalcVPCsIndicator`.

4.1.3. GSLIB File Routines

The file reading and writing functions from this sub-module are described in 2.5. Some additional functions which may come in useful to work with GSLIB files are described below.

a) **Cubes2PointSet** – converts a dictionary with GSLIB properties into the GSLIB PointSet format:

`PointSets = Cubes2PointSet(CubesDictionary, Mask)`

where:
- **CubesDictionary** is the dictionary with GSLIB properties;
- **Mask** defined **(Mask = 1)** / undefined **(Mask = 0)** is the cell mask array.

b) **Cube2PointSet** – converts defined **(Mask = 1)** cells of the NumPy-array **Cube** into a GSLIB PointSet:

`PointSet = Cube2PointSet(Cube, Mask)`

c) **PointSet2Cube** – converts a GSLIB PointSet into an HPGL property:

`Cube, Mask = PointSet2Cube(X, Y, Z, Property, Cube)`

where:
- **Cube** is the NumPy-array for converted points;

- **Mask** is the NumPy-array which defines the defined **(Mask = 1)** and undefined **(Mask = 0)** cells for **Cube**;
- **X** are the X-coordinates for the PointSet's points;
- **Y** are the Y-coordinates for the PointSet's points;
- **Z** are the Z-coordinates for the PointSet's points;
- **Property** is the NumPy-array with the PointSet property values.

Note: **Cube** must be initialized with the corresponding shape. After execution, it will be filled with Point Set values.

d) SaveGSLIBPointSet – saves a GSLIB PointSet (**PointSet**) as a GSLIB file (**FileName**) with a caption (**Caption**):

```
SaveGSLIBPointSet(PointSet, FileName, Caption)
```

4.1.4. Moving Average Calculation

The Moving Average function returns a NumPy-array which can be used in Local Varying Mean algorithms (SIS LVM, SGS LVM, LVM Kriging).

To calculate a moving average array `MACube` on the defined `(Mask = 1)` cells of the NumPy-array `Cube`, you should use the `MovingAverage3D` function:

```
MACube = MovingAverage3D((Cube, Mask), Radiuses, undefined_value,
                             MaskCalcFunction)
```

where:

- `Radiuses` is a Python tuple with radiuses for moving average calculation;
- `undefined_value` – this value will be set in `MACube` cells with insufficient points for moving average calculation;
- `MaskCalcFunction` is a pointer to a function that creates a moving average template:
    - `GetCubicalMask` – for a cubical moving average template;
    - `GetEllipseMask` – for an ellipsoid moving average template;

Example:

```
size_prop = [166, 141, 20]
undef = -99

prop = load_cont_property("DATA.INC", undef, size_prop)

Radiuses = (10, 10, 10)
```

```
MACube = MovingAverage3DP(prop, Radiuses, undef, GetCubicalMask)
```

4.2. geo_bsd.cvariogram

The `geo_bsd.cvariogram` sub-module contains some sample variogram calculation functions.

To calculate a sample variogram, you must first set up the variogram parameters by creating a `VariogramSearchTemplate` object:

```
var_templ_obj = VariogramSearchTemplate(
                    lag_width,
                    lag_separation,
                    tol_distance,
                    num_lags,
                    first_lag_distance,
                    ellipsoid)
```

where:

- `lag_width` is the variogram lag width;
- `lag_separation` is the distance between lags centers;
- `tol_distance` is the search cone height;
- `num_lags` is the number of lags;
- `first_lag_distance` is the distance between the cone node and the first lag center;
- `ellipsoid` is the ellipsoid which defines the search cube parameters; it must be an `Ellipsoid` class object (see below).

An `Ellipsoid` class object can be created as shown below:

```
ellipsoid_obj = Ellipsoid(R1, R2, R3, azimuth, dip, rotation)
```

where:

- `R1, R2, R3` are the ellipsoid radiuses (x,y,z);
- `azimuth, dip, rotation` are the corresponding rotation angles.

To calculate a sample variogram using the parameters defined in the `VariogramSearchTemplate` object, you can use the following functions:

1. To calculate a sample variogram on an HPGL property:

```
(lags_borders, variogram) = CalcVariograms(templ, hard_data,
                                percent=100)
```

2. To calculate a sample variogram on a GSLIB PointSet:

```
(lags_borders, variogram) = CalcVariogramsFromPointSet(templ,
                                point_set)
```

where:

- `lags_borders` are the lag borders for sample variogram values (X);
- `variogram` are the sample variogram values (Y);
- `templ` is the `VariogramSearchTemplate` object;
- `hard_data` is the HPGL property;
- `percent` is the part of the dataset (in percent), on which the sample variogram will be calculated (points will be selected by a random process); this can be used to speed up calculation on large datasets.

Example:

```
lag_width = 1
lag_separation = 1
tol_distance = 1
num_lags = 50
first_lag_distance = 0
r1, r2, r3 = 1, 1, 1
a1, a2, a3 = 0, 0, 0

prop_shape = (166, 141, 20)
prop = load_cont_property('fixed/BIG.INC', -99, prop_shape)

lags, variograms = cv.CalcVariograms(
    cv.VariogramSearchTemplate(
        lag_width,
        lag_separation,
        tol_distance,
        num_lags,
        first_lag_distance,
        cv.Ellipsoid(
            r1, r2, r3,
            a1, a2, a3)),
    prop)
```

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 0.9.9 - *18/02/2010*

- Now HPGL use CLAPACK solvers instead of internal ones, which means great performance boost on large scale linear equation solving problems.

HPGL 0.9.7 Xmas Edition - *31/12/2009*

- Main module name changed from **geo** to **geo_bsd**
- **cvariogram** module introduced for sample variogram calculation
- **CdfData** class introduced for CDF definition in SGS algorithms
- **ContProperty** and **IndProperty** classes for properties introduced (instead of a Python tuple)
- **boost::python** deprecated & replaced by **CTypes** for C-bindings (Python version >= 2.5 supported)
- **CovarianceModel** class introduced as the generic covariance model for all algorithms
- Project refactored to incorporate new building systems for Windows and Linux
- **.deb** packages now packed in the 'true' Debian way

HPGL 0.9.6 - *14/09/2009*

- Added sub-module **geo.routines**
- Module **geo** refactored (many changes in algorithms interfaces)
- **SGS LVM**: algorithm changed, now LVM-means preserved correctly
- **IK/SIS**: Median-algorithms now used by default for 2-indicator properties

- **SGS:** bug fixed for the **`cdf_data`** case

- Random path bug fixed (used to be incorrect for small grids of 100 or less cells)

- Project compilation scheme changed

- Packages for Python 2.5 & 2.6 (Windows + Linux) are now built simultaneously

- FORTRAN order in arrays now optional (arrays will be converted to FORTRAN order automatically inside algorithms)

- New GSLIB file read/write and VPC calculation functions – very fast now

- Sill > Nugget check added

HPGL 0.9.5 - *22/05/2009*

- Properties are now NumPy-array compatible

- GSLIB file support added

- Non-conditional Simulation support added

- Almost all algorithms (except Ordinary Kriging) now use a Cholesky decomposition solver, performance improved up to twice as fast

- **boost::python** now statically linked

HPGL 0.9.4 - *12/05/2009*

- GSTL deprecated
- Library now covered by the BSD License
- Nugget and anisotropy variograms added
- New algorithm structure
- Modeling regions in simulation algorithms

HPGL 0.9.3 - *06/04/2009*

- First open release

License

HPGL is distributed under terms of BSD license.
Full text of BSD license is presented below.