

HPGL: High Performance

Geostatistics Library

**version 0.9.5**

User Guide

2009

Table of contents

[1. Understanding Basics 4](#_Toc230771446)

[1.1. System Requirements 4](#_Toc230771447)

[1.2. Software Description 4](#_Toc230771448)

[1.3. Used components 5](#_Toc230771449)

[1.4. Installation 5](#_Toc230771450)

[1.4.1. Windows 5](#_Toc230771451)

[1.4.2. Linux 5](#_Toc230771452)

[2. Core Features 6](#_Toc230771453)

[2.1. Library import 6](#_Toc230771454)

[2.2. Creating an IJK grid 6](#_Toc230771455)

[2.3. Properties 6](#_Toc230771456)

[2.4. Importing data from text files 6](#_Toc230771457)

[2.5. Releasing data from memory 9](#_Toc230771458)

[2.6. Vertical Proportion Curve (VPC) property calculation 9](#_Toc230771459)

[2.7. Working with Properties 10](#_Toc230771460)

[3. Using the Algorithms 11](#_Toc230771461)

[3.1. Simple Kriging 11](#_Toc230771462)

[3.2. Ordinary Kriging 12](#_Toc230771463)

[3.3. Indicator Kriging 13](#_Toc230771464)

[3.4. LVM Kriging (Local Varying Mean) 15](#_Toc230771465)

[3.5. Sequential Indicator Simulation (SIS) (LVM, Corellogram) 16](#_Toc230771466)

[3.6. Sequential Gaussian Simulation (SGS, LVM SGS) 17](#_Toc230771467)

[4. Python Scripts Collection 21](#_Toc230771468)

[Contact the Authors 22](#_Toc230771469)

[Modification History 23](#_Toc230771470)

[License 24](#_Toc230771471)

# 1. Understanding Basics

## 1.1. System Requirements

Using HPGL requires a Windows (32-bit) or Linux (32/64-bit) operating system with Python Software version 2.5 and with NumPy and SciPy python packages installed (you may download the latest Python installer 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 **0.9.5** 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.

The following data import/export formats are currently supported:

- Eclipse property text file;

- GSLIB property text file.

## 1.3. Used components

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

## 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 (32-bit):

Install package **hpgl\_x32\_X.Y.Z\_ubuntu\_V.deb.**

1. Ubuntu (64-bit):

Install package **hpgl\_x64\_X.Y.Z\_ubuntu\_V.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. However, if you want to compile the project under another Linux system (or to create a package), feel free to contact the authors.

# 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. Properties

Since 0.9.5 version, all properties are tuples with NumPy arrays inside, with following structure:

property = (array\_prop, array\_informed)

where array\_prop – NumPy-array (float) with property data;

array\_informed – NumPy-array (uint), which define array\_prop points with value (array\_informed = 1), and points without value (array\_informed = 0).

## 2.4. Importing data from text files

Eclipse and GSLIB text files are natively supported.

1. **Ecpilse file format**

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 Eclipse 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, size)

property\_object = load\_ind\_property(filename, undefined\_value, [indicators], size)

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 [indicators] argument in the load\_ind\_property function are the codes of the indicators contained in the file.

The last argument – size – is a grid size in cells i,j,k for property, in Python tuple:

size = (i,j,k)

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

**Example:**

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)

A property is written to an Eclipse text file by write\_property:

write\_property(prop\_object, filename, prop\_name, undefined\_value, indicator\_values=[])

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 are defined by indicator\_values. If indicator\_values is not set, indicator values in saved property will be 0,1,2,…

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

**Example:**

write\_property(cont\_prop, "CON\_PROP.INC", "PROPCON", -99)

write\_property(i\_prop, "INDP.INC", "PROP\_IND", -99, [0,1])

1. **GSLIB file format**

Importing properties from GSLIB text files (for format description, see http://www.gslib.com/gslib\_help/format.html) is implemented in function load\_gslib\_file:

dict\_gslib = load\_gslib\_file(**filename**)

where dict\_gslib is a Python dictionary with data from file filename (dictionary items are the NumPy-array properties from file).

To access the property named **property\_1** you can use following syntax:

dict\_gslib[“property\_1”]

A property is written to a GSLIB text file by write\_gslib\_file:

write\_gslib\_file(dict\_gslib, **filename, caption**),

where dict\_gslib is a Python dictionary with properties as NumPy arrays items, and **caption** is a first string of file.

For advanced use of dictionaries, please refer to Python manual.

If you want to use data from GSLIB files in algorithms, you will need to convert it in properties format described in **2.3**. This can be achieved by using get\_gslib\_property function:

get\_gslib\_property(prop\_dict, prop\_name, undefined\_value)

where prop\_dict – python dictionary with loaded data from GSLIB file;

prop\_name – name of the property, which you want to convert;

undefined\_value – value of undefined points in property prop\_name.

**Example:**

gslib\_prop = load\_gslib\_file("test\_data/samples.gslib")

egslib\_prop = get\_gslib\_property(gslib\_prop, "porosity", -99)

## 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)

## 2.6. Vertical Proportion Curve (VPC) property calculation

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

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

vpc\_cube = calc\_ind\_vpc(prop, marginal\_probs)

where 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].

VPC cubes are NumPy arrays.

An object created by calc\_vpc can be used in all algorithms with a varying mean option as a mean\_data parameter.

**Example:**

size = (55, 52, 1)

grid = SugarboxGrid(55, 52, 1)

prop = load\_ind\_property("IK\_HARD\_DATA.INC", -99, [0,1], size )

vpc\_cube = calc\_ind\_vpc( prop, [0.8, 0.2] )

## 2.7. Working with Properties

Since 0.9.5 release, HPGL properties are the NumPy arrays, so you can work with them like with any other NumPy arrays (see NumPy/SciPy documentation on the site).

# 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:**

size = (55, 52, 1)

grid = SugarboxGrid(55, 52, 1)

prop = load\_cont\_property("HARD\_DATA.INC", -99, size )

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:**

size = (55, 52, 1)

grid = SugarboxGrid(55, 52, 1)

prop = load\_cont\_property("HARD\_DATA.INC", -99, size )

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 parameters 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 parameters 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:**

size = (55, 52, 1)

grid = SugarboxGrid(55, 52, 1)

ik\_prop = load\_ind\_property("HARD\_DATA.INC", -99, [0,1], size)

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, "RES\_IK.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

(

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

)

**Example:**

grid = SugarboxGrid(55, 52, 1)

size = (55, 52, 1)

mean\_data = load\_cont\_property("cube\_local\_means.inc", size)

lvm\_prop = load\_ind\_property("LVM.INC", -99, [0,1], size)

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 described 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\_harddata": **True,**

# if **False,** non-conditional simulation

# without hard data will be performed

use\_corellogram = False

# Correlogram LVM SIS (only with mean data)

# True — use Correlogram SIS

# False — use Classic LVM SIS

)

**Example:**

size = (55, 52, 1)

grid = SugarboxGrid(55, 52, 1)

sis\_prop = load\_ind\_property("HARD\_DATA.INC", -99, [0,1], size)

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, "RES\_SIS.INC", "P\_SIS", -99, [0,1])

## 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:

# itmay 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

"cdf\_data": **None,**

# Property for CDF transform

"use\_harddata": **True,**

# if **False,** non-conditional simulation

# without hard data will be used

# "mean\_data": **mean\_data\_obj**

# lvm values property

"mask": **mask,**

# property defining modeling regions

# mask must be indicator property with values

# 0 (points which are not to be modeled) and

# 1 (points to be modeled)

}

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 if 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.

**Example:**

size = (55, 52, 1)

grid = SugarboxGrid(55, 52, 1)

prop = load\_cont\_property("SGS\_HARD\_DATA.INC", -99, size)

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)

## Example(LVM):

grid = SugarboxGrid(55, 52, 1)

size = (55, 52, 1)

prop = load\_cont\_property("HARD\_DATA.INC", -99, size )

mean\_data = load\_cont\_property("MEAN.INC", -99, size )

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.

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 collection can be downloaded from HPGL project site (http://hpgl.sourceforge.net).

# 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 0.9.5** *- 22/05/2009*

* Properties now are NumPy arrays compatible.
* GSLIB files support added.
* Non-conditional simulation support added
* Almost all algorithms (except the Ordinary Kriging) now use Cholesky decomposition solver, performance is improved up to 2 times.

**HPGL 0.9.4** *- 12/05/2009*

* GsTL is not used anymore.
* Library switched to BSD License.
* Nugget and anisotropy variograms added.
* New algorithms structure.
* Modeling regions support 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.

Copyright (c) 2009, HPGL Team

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

\* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

\* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

\* Neither the name of the HPGL nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.