

HPGL: High Performance

Geostatistics Library

**version 0.9.6 BSD**

User Guide

2009

Table of contents

[1. Understanding Basics 4](#_Toc240180300)

[1.1. Software Description 4](#_Toc240180301)

[1.2. System Requirements and installation 4](#_Toc240180302)

[1.2.1. Windows 5](#_Toc240180303)

[1.2.2. Ubuntu Linux 5](#_Toc240180304)

[1.2.3. Other OS 5](#_Toc240180305)

[1.3. Used components 5](#_Toc240180306)

[2. Core Features 6](#_Toc240180307)

[2.1. Library import 6](#_Toc240180308)

[2.2. Creating an IJK grid 6](#_Toc240180309)

[2.3. Properties 6](#_Toc240180310)

[2.4. Eclipse Property file format 7](#_Toc240180311)

[2.4.1. Reading Eclipse Property files 7](#_Toc240180312)

[2.4.2. Writing Eclipse Property files 8](#_Toc240180313)

[2.5. GSLIB file format 9](#_Toc240180314)

[2.5.1. Reading GSLIB files 9](#_Toc240180315)

[2.5.2. Writing GSLIB files 9](#_Toc240180316)

[2.6. Covariance (variogram) object 9](#_Toc240180317)

[2.7. Releasing data from memory 10](#_Toc240180318)

[3. Using the Algorithms 11](#_Toc240180319)

[3.1. Simple Kriging 11](#_Toc240180320)

[3.2. Ordinary Kriging 11](#_Toc240180321)

[3.3. Indicator Kriging 12](#_Toc240180322)

[3.4. LVM Kriging (Local Varying Mean) 13](#_Toc240180323)

[3.5. Sequential Indicator Simulation (SIS) 14](#_Toc240180324)

[3.6. Sequential Gaussian Simulation (SGS) 15](#_Toc240180325)

[4. Sub modules 18](#_Toc240180326)

[4.1. geo.routines 18](#_Toc240180327)

[4.1.1. Mean calculation 18](#_Toc240180328)

[4.1.2. VPC (Vertical Proportion Curve) Calculation 18](#_Toc240180329)

[4.1.3. GSLIB files functions 19](#_Toc240180330)

[4.1.4. Расчет «ползущего среднего» 20](#_Toc240180331)

[Contact the Authors 22](#_Toc240180332)

[Modification History 23](#_Toc240180333)

[License 25](#_Toc240180334)

# 1. Understanding Basics

## 1.1. 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 **0.9.6 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.

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 Python Software version 2.5/2.6 and NumPy/SciPy python packages installed (for corresponding Python version).

### 1.2.1. Windows

MS Windows installation require 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!** Redistributable package must be at revision date 7/28/2009 or later (after ATL security update).

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

### 1.2.2. Ubuntu Linux

Ubuntu (32-bit/64-bit):

Install package **hpgl\_X.Y.Z-BSD-[x32/x64].deb (**corresponding to operation system’s architecture). Using HPGL also requires boost libraries to be installed.

### 1.2.3. Other OS

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. Used components

* 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 import **geo** module command:

from geo import \*

HPGL also includes sub module called **geo.routines** with additional properties-related algorithms: VPC (Vertical Proportion Curve) and moving average calculations, GSLIB file format support, etc.

If you want to use this sub module Python script must be started with:

from geo.routines import \*

For detailed information about **geo.routines** sub module, see ch. 4.

## 

## 2.2. Creating an IJK grid

Every HPGL geostatistics 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 HPGL are the Python tuples with NumPy arrays:

property = (array\_prop, array\_informed)

where **array\_prop** – NumPy-array - *float32* type [for continuous property] or *uint8* type [for indicator property] with property data;

**array\_informed** – NumPy-array - *uint8* type, which define **array\_prop** points with value (*array\_informed = 1*), and points without value (*array\_informed = 0*),

Indicator properties also require one additional parameter to be added:

property = (array\_prop, array\_informed, **indicators\_number**)

where **indicators\_number** – number of categorical indicators in **array\_prop**.

***WARNING!*** Numpy arrays must be in FORTRAN data store order. This can be achieved with:

- when creating a new array:

a = array([], **order=’F’**)

- when changing existing non-Fortran order array:

a = require(a, **requirements=’F’**)

If HPGL algorithm input array will be non-Fortran, it’ll be converted automatically, so you need to keep in mind that *all resulted properties will be returned as FORTRAN order arrays*.

More information about FORTRAN order arrays located here: <http://www.ibiblio.org/pub/languages/fortran/ch2-6.html>.

## 2.4. Eclipse Property file format

HPGL supports reading and writing of Eclipse Property files.

Eclipse Property files must be in the following format:

-- comment (will be ignored)

PROPERTY\_NAME

0

1

0

...

/

File data must be in the certain axis order: *i=0…max, j=max…0, k=0…max*.

### 2.4.1. Reading Eclipse Property files

Reading properties from Eclipse Property text files implemented with 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)

Both these commands will create an object (**prop**) which will contain 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**.

The **[indicators]** argument in the **load\_ind\_property** function is the Python tuple with indicator codes, contained in the file.

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

size = (i,j,k)

**WARNING!** After loading data from the file, indicators will be 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)

### 2.4.2. Writing Eclipse Property files

Property can be written to the Eclipse Property file using **write\_property** function:

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

This command will create the file **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 property, indicator values are defined by **indicator\_values**. If **indicator\_values** is not set, indicators in 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 file format

Detailed description of GSLIB file format can be found at <http://www.gslib.com/gslib_help/format.html>. All GSLIB-related functions are included in sub module geo.routines, so you need to import it before using them:

from geo.routines import \*

### 2.5.1. Reading GSLIB files

Reading properties from GSLIB file implemented in **LoadGslibFile** function:

dict\_gslib = LoadGslibFile(**filename**)

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

Property named **property\_1** can be accessed using the following syntax:

dict\_gslib[“property\_1”]

### 2.5.2. Writing GSLIB files

HPGL property can be written into a GSLIB file by **SaveGSLIBCubes** function:

SaveGSLIBCubes(dict\_gslib, **filename**, caption, Format = "%d")

where **filename** – GSLIB file name;

**dict\_gslib** – Python dictionary with properties as Numpy-arrays;

**caption** – caption of the GSLIB file.

Detailed description of Python dictionaries can be found in Python documentation (for example, here –

[http://docs.python.org/tutorial/datastructures.html#dictionaries](http://docs.python.org/tutorial/datastructures.html%23dictionaries)).

## 2.6. Covariance (variogram) object

All HPGL geostatistical algorithms use unified type of covariance (variogram) function. 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 – variogram type:

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

ranges – variogram ellipsoid ranges (0⁰, 90⁰, vertical);

angles – variogram ellipsoid angles;

sill – sill value of the variogram;

nugget – “nugget”-effect value.

Object cov can be used in all HPGL geostatistical algorithms.

## 2.7. 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)

# 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)

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 the Indicator Kriging's 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)

Algorithm SGS is implemented in 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,**

# used only if **use\_harddata = True**

#numpy-array (float32), which is used as histogram (CDF) source

# when unconditional SGS takes place

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.routines

Sub module geo.routines has many additional functions to work with HPGL properties.

### 4.1.1. Mean calculation

1. CalcMean – returns mean value for Numpy-array Cube calculated on defined (Mask = 1) cells:

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

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

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

### 4.1.2. VPC (Vertical Proportion Curve) Calculation

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

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

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

1. CalcVPCsIndicator - returns Python list with Numpy-arrays VPC (*Vertical Proportion Curve*) – means from vertical slices of Numpy-array Cube – for each of indicators, defined in Indicators, calculated on defined (Mask = 1) cells:

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

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

с) **CubeFromVPC** – creates 3D Numpy-array with shape **NX**, **NY**, len(**VPC**), filled with **VPС** 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 with **CalcVPC**.

с) **CubesFromVPCs** – creates 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 with CalcVPCsIndicator.

### 4.1.3. GSLIB files functions

Reading and writing functions from this sub module described in ch. 2.5. Some additional functions, useful to work with GSLIB files describes below.

1. Cubes2PointSet – converts dictionary with GSLIB properties in GSLIB PointSet format:

**PointSets** = Cubes2PointSet(**CubesDictionary**, **Mask**)

where:

**-**  **CubesDictionary –** dictionary with GSLIB properties**;**

**- Mask** – defined **(Mask = 1)** / undefined **(Mask = 0)** cells mask array.

1. **Cube2PointSet** – converts defined **(Mask = 1)** cells of Numpy-array **Cube** into GSLIB Point Set:

**PointSet** = Cube2PointSet(**Cube**, **Mask**)

**с) PointSet2Cube** – converts GSLIB Point Set into HPGL property:

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

where:

- **Cube** – Numpy-array for converted points;

- **Mask** – Numpy-array which defines defined **(Mask = 1)** and undefined **(Mask = 0)** cells for **Cube**;

- **X** – X-coordinates for Point Set’s points;

- **Y** – Y-coordinates for Point Set’s points;

- **Z** – Z-coordinates for Point Set’s points;

- **Property** – Numpy-array with Point Set property values.

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

e) SaveGSLIBPointSet – saves GSLIB Point Set (PointSet) into GSLIB file (FileName) with caption (Caption):

SaveGSLIBPointSet(**PointSet**, **FileName**, **Caption**)

### 4.1.4. Moving average calculation

Moving average function returns numpy-array, which can be used for Local Varying Mean algorithms (SIS LVM, SGS LVM, LVM Kriging).

If you want to calculate moving average array MACubeon defined (Mask = 1) cells of numpy-array Cube, you can use MovingAverage3D function:

**MACube** = MovingAverage3D((**Cube**, **Mask**), **Radiuses**, **undefined**\_value, **MaskCalcFunction**)

where:

- Radiuses –Python tuple with radiuses for moving average calculation;

- undefined\_value – this value will be set in MACube cells, where not enough points for moving average calculation was;

- MaskCalcFunction – pointer on function, which construct moving average template:

- GetCubicalMask – for cubical moving average template;

- GetEllipseMask – for 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)

# 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 0.9.6** *- 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 indicators properties
* **SGS:** bug fixed for **cdf\_data** case
* Random path bug fixed (was wrong for small grids with 100 or less cells)
* Project compilation scheme changed
* Packages for Python 2.5 & 2.6 (Windows + Linux) now builds simultaneously
* FORTRAN order in arrays now optional (arrays will be converted to FORTRAN order automatically inside algorithms).
* New GSLIB files read/write and VPC calculation functions – very fast now.
* Sill > Nugget check added

**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.
* boost::python now statically linked

**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.