HIGH PERFORMANCE GEOSTATISTICS LIBRARY

HPGL

User Guide

HPGL: High Perfomance Geostatistics Library

версия 0.9.3

Руководство пользователя

2009

Оглавление

1. Основные сведения 4
1.1. Требования к системе 4
1.2. Описание 4
1.3. Используемые компоненты 5
1.4. Установка 5
1.4.1. Windows 5
1.4.2. Linux 5
2. Основные функции 6
2.1. Импорт библиотек 6
2.2. Создание IJK сетки 6
2.3. Загрузка свойств из файлов данных 6
2.4. Сохранение свойств в файлы данных 7
2.5. Удаление данных из памяти 7
2.6. Расчет куба Vertical Proportion Curve (VPC) 8
2.7. Функции работы с кубами свойств 9
3. Использование алгоритмов 12
3.1. Simple Kriging 12
3.2. Ordinary Kriging 13
3.3. Indicator Kriging 14
3.4. LVM Kriging (Local Varying Mean) 16
3.5. Sequential Indicator Simulation (SIS) (VPC, Corellogram) 17
3.6. Sequential Gaussian Simulation (SGS, SGS LVM) 18
4. Набор скриптов Python 21
Авторы - Контакты 22
Список изменений 23

1. Основные сведения

1.1. Требования к системе

Для работы HPGL потребуется операционная система Windows (32 бита) или Linux (32/64 бита) с установленным Python версии не ниже 2.5 (скачать последнюю версию Python можно по адресу http://www.python.org/download/).

1.2. Описание

HPGL является C++/Python библиотекой, в которой реализованы геостатистические алгоритмы. Использование алгоритмов осуществляется командами на языке Python, благодаря чему возможно создание необходимых сценариев геологического моделирования.

В версии **0.9.3** реализованы следующие алгоритмы:

- 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)*

  * в наборе скриптов на Python

Распространение свойств производится в IJK пространстве, поэтому все параметры (например, радиусы вариограмм и эллипсоида) задаются в ячейках сетки.

Поддерживается текстовый формат загрузки/выгрузки данных, в будущем планируется обеспечить совместимость в python numpy array.

1.3. Используемые компоненты

В библиотеке используются следующие свободно распространяемые компоненты:

- модифицированная версия библиотеки GsTL (оригинал доступен по адресу https://sourceforge.net/projects/gstl, модифицированная версия – в репозитории проекта HPGL).

- TNT (Template Numerical Toolkit) – библиотека для решения систем линейных уравнений (доступна по адресу http://math.nist.gov/tnt/overview.html)

1.4. Установка

1.4.1. Windows

Используйте инсталляционный файл HPGL-X.Y.Z.win32.exe.

В операционных системах Windows также потребуется установить Microsoft Visual C++ 2005 SP1 Redistributable Package (можно скачать по адресу http://www.microsoft.com/downloads/details.aspx?familyid=200B2FD9-AE1A-4A14-984D-389C36F85647&displaylang=en)

1.4.2. Linux

Ubuntu 8.10 (32-bit):

Установите пакет **`hpgl_x32_X.Y.Z_ubuntu_8.10.deb.`**

Ubuntu 8.10 (64-bit):

Установите пакет **`hpgl_x64_X.Y.Z_ubuntu_8.10.deb.`**

Все необходимые дополнительные пакеты (например библиотеки boost) будут установлены по зависимостям из репозитория.

К сожалению, пока нет сборок бинарных пакетов для других дистрибутивов, но если вы захотите собрать HPGL под другой дистрибутив, можете писать все вопросы в рассылку (см. Авторы – Контакты в конце документа). Инструкции по компиляции скоро будут на сайте проекта (http://hpgl.sourceforge.com)

2. Основные функции

2.1. Импорт библиотек

В начале каждого Python-скрипта, который будет использовать HPGL, необходимо прописать строки импорта библиотечных функций:

```
from geo import *
```

2.2. Создание IJK сетки

Создание IJK сетки производится при помощи функции `SugarboxGrid`:

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

В результате выполнения команды будет создан объект сетки `grid` с размерностями `I, J, K`.

Пример:

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

2.3. Загрузка свойств из файлов данных

Поддерживаются текстовые файлы с данными следующего формата:

```
-- комментарий (игнорируется)

PROPERTY_NAME
-- данные в порядке k,j,i (по слоям)
0
1
0
...
/
```

Файл формата данных должен содержать последовательно записанные значения из сетки, в порядке k,j,i (т.е. вертикальными слоями по k). Загрузка свойства из файла производится при помощи двух функций:

- `load_ind_property` для индикаторных величин;
- `load_cont_property` для непрерывных величин.

```
property_object = load_cont_property(filename,
undefined_value)
```

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

В результате выполнения команды будет создан объект `property_object`, в который будет загружено свойство из файла `filename`. Ячейки, значение в которых равно `undefined_value`, будут считаться пустыми.

Последним параметров функции `load_ind_property` передаются коды индикаторов, находящихся в файле.

Примечание: После загрузки индикаторного свойства из файла, все индикаторы преобразуются к виду 0,1,2,… в порядке, указанном в `indicators`.

Пример:

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

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

2.4. Сохранение свойств в файлы данных

Сохранение свойства в файл производится при помощи функции `write_property`:

```
write_property(prop_object, filename, prop_name, undefined_value,
                                [indicators])
```

В результате выполнения команды будет создан файл `filename`, в который будет записано свойство из объекта `prop_object` под именем `prop_name`. На место пустых ячеек будет записано `undefined_value`. Индикаторы будут записаны с индексами, указанными в `indicators` (если он не задан, то по порядку: 0, 1, 2…)

Тип свойства при записи (индикаторный/непрерывный) сохраняется автоматически.

Пример:

```
write_property(my_cont_prop, "CON_PROP.INC", "PROP_CON", -99)
write_property(my_ind_prop, "PR.INC", "PROP_IND", -99, [0,1])
```

2.5. Удаление данных из памяти

Когда свойство уже не нужно, его можно удалить, освободив оперативную память. Делается это командой `del`:

```
del(prop_object)
```

Пример :

```
del(my_prop)
```

2.6. Расчет куба Vertical Proportion Curve (VPC)

Куб VPC – это куб, содержащий среднее значение свойства по каждому из вертикальных слоев. Он может быть использован в качестве источника средних значений в алгоритмах с изменяющимся средним (LVM SIS, LVM Corellogram SIS, LVM Kriging, и т.д.).

Расчет куба VPC по загруженному свойству `prop` выполняется следующим образом:

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

где `grid` – это сетка, на которой выполняется расчет;

`marginal_probs` – средние значения для каждого индикатора (это значение будет поставлено в те места, где по вертикали слой оказался пустым).

Нужно иметь в виду, что внутри куба `vpc_cube` содержатся кубы средних для каждого из индикаторов, если индикаторов два — это будут соответственно `vpc_cube[0]` и `vpc_cube[1]`.

Для сохранения куба VPC для любого из индикаторов, в файл, необходимо использовать функцию `write_mean_data`.

Куб, созданный при помощи `calc_vpc` можно использовать для любого из алгоритмов с изменяющимся средним, передавая его в качестве параметра `mean_data`.

Пример :

```
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. Функции работы с кубами свойств

Каждый куб свойств полученный в результате выполнения алгоритма, является объектом, и у него есть ряд методов.

1) **Создание пустого свойства заданного типа и размера**

Существует две возможность создать куб свойств – создать его с нуля (тогда он будет заполнен undefined-значениями) или осуществить копирование уже существующего.

Создание пустого свойства осуществляется функциями:

- для *непрерывного* свойства:

```
hpgl.cont_property_array(size)
```

- для *индикаторного* свойства:

```
hpgl.byte_property_array(size, [indicators])
```

Для копирования уже существующего свойства необходимо использовать функцию clone():

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

Пример:

```
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()
```

Примечание: Если вам необходимо проверить, к какому типу (непрерывному или индикаторному) принадлежит какое-либо свойство, это можно сделать следующим образом:

```
if type(property) is hpgl.cont_property_array:
     # действия, если property – непрерывное свойство
else:
     # действия, если property – индикаторное свойство
```

ОБРАТИТЕ ВНИМАНИЕ, что конструкция вида:

```
property_1 = property_2
```

не создает копию свойства `property_2`, а лишь делает `property_1` **указателем** на `property_2`! Для создания полноценной копии нужно использовать функцию `clone()`.

2) Работа со значениями свойства через get_at/set_at

Обращение к определенному значению свойства идет по индексу, от 0 до N-1, где N – это размер свойства. Индексирование идет в том порядке, в каком свойство было записано в файл данных, поэтому эти функции подходят в основном для изменения или получения значений в точках **независимо от их координат в сетке**. Плюс данного метода в том, что он очень быстрый и не занимает дополнительной памяти.

Все точки свойства делятся на два типа: известные (informed) и неизвестные (not informed) – т.е. те, в которых значение еще не смоделировано. Проверка на известность осуществляется функцией `is_informed()`:

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

Если значение точки index известно, то `result`будет равен `True`, если нет, то `False`.

Для выполнения операций, в которых необходимо знать точные координаты изменяемых точек (a la работа с трехмерными массивами MATLAB), нужно воспользоваться функцией преобразования свойства в numpy-массив (описано ниже) – в этом случае с ним можно будет обращаться как с обыкновенным трехмерным массивом.

Для получения значения по индексу `index` из куба свойств `property` существует функция `get_at()`. **Обратите внимание**, что перед использованием функции get_at всегда необходимо проверять, является ли значение известным:

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

Для изменения значения по индексу `index` на `value` из куба свойств `property` существует функция `set_at()`:

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

После выполнения этой функции точка **index** будет считаться известной (informed).

Пример:

```
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) **Работа со значениями свойства через NumPy-массив**

Будет в следующем релизе.

3. Использование алгоритмов

3.1. Simple Kriging

Вызов алгоритма Simple Kriging осуществляется при помощи функции `simple_kriging`:

```
def simple_kriging(
        prop,               # свойство с изначальными данными (hard data)
        grid,               # сетка, на которой производится SK
        radiuses,           # радиусы эллипсоида поиска.
        max_neighbours,     # максимальное число точек участвующих в интерполяции
        covariance_type,    # тип ковариации.
                            # может принимать значения
                            # covariance.spherical, covariance.exponential
                            # и covariance.gaussian
        ranges,             # радиусы вариограммы
        sill,               # пороговое значение вариограммы
        nugget=None, # величина nugget-эффекта
        angles=None,        # углы поворота вариограммы
        mean=None           # среднее значение
                                # если не задано, вычисляется автоматически по
                                # изначальным данным
)
```

Пример:

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

3.2. Ordinary Kriging

Вызов алгоритма Ordinary Kriging осуществляется при помощи функции `ordinary_kriging`:

```
def ordinary_kriging(
    prop,               # свойство с изначальными данными (hard data)
    grid,               # сетка, на которой производится OK
    radiuses,           # радиусы эллипсоида поиска
    max_neighbours,     # максимальное число точек участвующее в интерполяции
    covariance_type,    # тип ковариации.
                        # может принимать значения
                        # covariance.spherical, covariance.exponential
                        # и covariance.gaussian
    ranges,             # радиусы вариограммы
    sill,               # пороговове значение вариограммы
    nugget=None, # величина nugget-эффекта
    angles=None,        # углы поворота вариограммы
)
```

Пример:

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

3.3. Indicator Kriging

Перед вызовом функции `indicator_kriging`, выполняющей алгоритм, необходимо заполнить структуру, содержащую параметры моделирования.

Ниже приведен пример заполнения этой структуры параметров:

```
ik_data = [

        # Параметры вариограммы 1-го индикатора

        {
            "cov_type": cov_type,           # тип вариограммы:
                                            # 0 - сферическая
                                            # 1 - экспоненциальная
                                            # 2 - гауссовая

            "ranges": (R1, R2, R3),         # радиусы вариограммы

            'sill': sill,           # Sill — пороговое значение вариограммы

            "radiuses": (SR1, SR2, SR3),    # радиусы эллипсоида поиска

        "max_neighbours": neigh_count,  # максимальное количество соседей
                                            # для интерполяции

        "marginal_prob": marg_prob,     # априорная вероятность индикатора

            "value": 0                      # значение индикатора
        },

        # Параметры вариограммы 2-го индикатора

        {
            "cov_type": cov_type,           # тип вариограммы:
                                            # 0 - сферическая
                                            # 1 - экспоненциальная
                                            # 2 - гауссовая

            "ranges": (R1, R2, R3),         # радиусы вариограммы

            'sill': sill,           # Sill — пороговое значение вариограммы

            "radiuses": (SR1, SR2, SR3),    # радиусы эллипсоида поиска

        "max_neighbours": neigh_count,  # максимальное количество соседей
                                            # для интерполяции

        "marginal_prob": marg_prob,     # априорная вероятность индикатора

            "value": 1                      # значение индикатора
```

```
            }
]
```

Для каждого существующего индикатора задается своя вариограмма.

Примечание: При использовании только двух индикаторов автоматически выполняется Median IK.

Когда структура параметров заполнена, можно вызывать на исполнение алгоритм, используя функцию `indicator_kriging`:

```
def indicator_kriging
(

            ik_prop,            # список с параметрами алгоритма
            grid,               # сетка на которой производится кригинг
            ik_data,      # свойства с исходными данными (hard data)
)
```

Пример:

```
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)

Кригинг с изменяющимся средним (LVM) производится при помощи функции `lvm_kriging`:

```
def lvm_kriging
(
    lvm_prop,           # свойства с исходными данными (hard data)
    grid,               # сетка, на которой производится LVM Kriging
    mean_data,          # куб, содержащий средние значения свойства для LVM
    radiuses,           # радиусы эллипсоида поиска
    max_neighbours,     # максимальное число точек участвующее в интерполяции
    covariance_type,    # тип ковариации.
                        # может принимать значения
                        # covariance.spherical, covariance.exponential
                        # и covariance.gaussian
    ranges,             # радиусы вариограммы
    sill,               # пороговове значение вариограммы
    nugget=None, # величина nugget-эффекта
    angles=None,        # углы поворота вариограммы
)
```

Куб, используемый в качестве параметра mean_data, загружается при помощи функции `load_mean_data`:

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

где `filename` — путь к файлу со кубом средних значений;

`mean_data_obj` — куб средних значений, который будет использова в алгоритме.

Пример :

```
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.inc", "lvm_kriging", -99)

    del(mean_data)
    del(prop_lvm)
```

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

Параметры моделирования для алгоритма SIS задаются аналогично IK, заполнением той же структуры.

Вызов функции:

```
def sis_simulation(
    ik_prop,            # список с параметрами алгоритма

    grid,               # сетка на которой производится кригинг

    ik_data,            # свойства с исходными данными (hard data)

    seed,               # Seed — номер стохастической реализации

    mask = False,       # задает регион для моделирования (если нужно
                            # моделировать не все точки, а только часть)

                            # задается в виде индикаторного куба, где на месте точек,
                            # подлежащих моделированию стоит 1, а на месте
                            # не-моделируемых точек – 0.

                            # если не задано, моделируются все неизвестные точки

    mean_data = None,   # куб с вероятностями для  LVM SIS

    use_corellogram = True

                        # использование кореллограмного SIS или LVM SIS

                        # True — использовать кореллограмный SIS
                        # False — использовать LVM SIS
)
```

Пример :

```
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, SGS LVM)

Вызов алгоритма SGS производится следующим образом.
Сначала нужно заполнить структуру с параметрами моделирования:

```
sgs_params = {
        "radiuses": (SR1, SR2, SR3),
            # радиусы эллипсоида поиска  (max, med, min)
        "max_neighbours": max_neigh,
            # максимальное количество соседних точек, используемых при
            # интерполяции
        "covariance_type": cov_model,
            # тип вариограммы, может принимать значения:
            # covariance.spherical — сферическая модель
            # covariance.exponential — экспоненциальная модель
            # covariance.gaussian — гауссовая модель
        "ranges": (R1, R2, R3),
            # радиусы вариограмм (max, med, min)
        "sill": sill,
            # пороговое значение вариограммы
        "kriging_type": krig_type,
            # тип кригинга, может принимать значения:
            # “sk” - для Simple Kriging
            # “ok” - для Ordinary Kriging
```

"mean": **mean**
среднее значение свойства (при использовании Simple Kriging)
"mean_data": **mean_data_obj**
# среднее значение свойства, заданное в виде объекта свойтсв
# (куба с данными)
}

После этого производится вызов алгоритма при помощи функции `sgs_simulation`:

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

Параметры функции:

- `property_obj` — свойство с изначальными данными (hard_data);
- `grid_obj` — сетка, на которой производится моделирование;
- `seed` — номер стохастической реализации;
- `sgs_params` — заполненные параметры моделирования SGS
- `mask` - задает регион для моделирования (если нужно моделировать не все точки, а только часть). Задается в виде индикаторного куба, где на месте точек, подлежащих моделированию, стоит 1, а на месте не-моделируемых точек – 0. Если не задано, моделируются все неизвестные точки.

Пример:

```
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=12434,
**sgs_params)

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

Пример (LVM):

```
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. Набор скриптов Python

Набор скриптов на языке Python создан для того, чтобы показать примеры использования алгоритм и решения различных задач с использованием HPGL.

Для работы скриптов понадобится установить дополнение для Python, добавляющее работу с многомерными массивами — NumPy и SciPy (http://www.scipy.org/Download).

Для вывода на экран графиков (гистограмм и карт) необходимо также поставить модуль matplotlib (http://sourceforge.net/projects/matplotlib).

Перечень скриптов включен в файл `scripts_list.txt` с набором скриптов.

Авторы - Контакты

Савичев Владимир
Безруков Андрей
Мухарлямов Артур
Барский Константин
Насибуллина Дина

По всем вопросам обращайтесь по адресу:
hpgl-support-rus@lists.sourceforge.net.

Список изменений

HPGL 0.9.3 - *06/04/2009*

Первая открытая версия