OCTA

Integrated simulation system for soft materials

GOURMET PYTHON SCRIPT

REFERENCE

OCTA User's Group

DEC. 25 2002

Authors of the Manual

The Japan Research Institute, Limited

Program Developers

Software Design	Shinji Shibano, Yuzo Nishio
Software Development	Masahiro Nishimoto
Testing	Yuzo Nishio, Eisuke Nishitani
	The Japan Research Institute, Limited

Acknowledgment

A large part of this work is supported by the national project, which has been entrusted to the Japan Chemical Innovation Institute (JCII) by the New Energy and Industrial Technology Development Organization (NEDO) under METI's Program for the Scientific Technology Development for Industries that Creates New Industries.

Contents

List of Figures

Chapter 1

Preface

In GOURMET, using the Python script language, you can easily operate the data described in a UDF file. With Python script, you can perform not only getting or editing the value, but changing the unit which is attendant to the data, or adding a new analysis step and writing out a file.

In addition, using the drawing functions which can be used from the Python, you can draw simple figures, such as a segment, an arrow, a sphere and a tetrahedron etc, and a little complicated figures, such as a contour and an isosurface on the drawing output screen of GOURMET.

There are two ways of executing the above operations by Python scripts:

- **UDF Manger for Python**

  After starting Python from a command window so that Python may usually be handled, by importing a special extended library, the extended Python commands treating UDF data are available. The command group treating UDF data is summarized as the UDFManager class methods, and is explained in Section 3.2.

- **Python in GOURMET**

  After starting the GOURMET,which is the graphical user interface tool, and reading a UDF formatting data file, the following Python commands can be executed in Python scripting window of GOURMET.

  - •the methods of the UDFManager class.
  - •the drawing functions for drawing figures on the viewer screen of the GOURMET.
  - •the sheet handling functions for graph drawing.

  Special usage to execute the methods of the UDFManager class on the Python scripting window of GOURMET is introduced in Section 3.3. And this section discusses the drawing functions and the sheet handling functions.

Chapter 2 describes the basic usage of Python and Python's terms, that is, the above-mentioned "extended library" and "import" are also explained.

Chapter 2

Basic usage of Python

This chapter describes how to use Python and how to write Python scripts briefly. If you want to know about Python in more detail, please read the books in the bibliography of the last of this chapter.

2.1 Starting of Python

Python runs in many platforms, such as Unix, Linux and Windows, etc. With a Unix (Linux) machine, if you type "python", Python interpreter will start.(Notes [1]) On Windows, Python interpreter can be started as an application from the start menu.

2.2 Variable

There is no type declaration of variables in Python interpreter. The value handled in Python is an object, which are a numerical object (integer, long integer, floating point and complex types), a string object, an array object and the object of other classes. The variable of Python interpreter only binds the name and a object, so there is no type declaration of variables. Easy examples are shown below.

```
msg="Hellow" + "world"
int=10
flt=0.5
ans=int*(1+flt)*2.5
print(msg, ans)
```

A string is enclosed in single quotes ('), double quotes ("), triple single quotes ("') or triple double quotes (""""). Operators + can use to connect 2 strings. For numerical values, the usual arithmetic operations (+, -, *, /), a power operation (**) and a operator (%) can be used. The operator % yields the remainder from the division of the first expression by the second.

2.3 tuple, list, dictionary

Python offers the tuple, the list and the dictionary as the data types, which gather several separate value into one. Once a tuple is created, we cannot change the elements or the number of elements. On the other hand, a list can change the elements, and can delete or add elements. A dictionary is the built-in hash table, so we can add, get or delete a value by the key from a dictionary object. The example of these usage and the executing result are shown below.

```
tuple = ( 'awk' , 'perl' , 'ruby' , 'python' )
list = [ 1, 2.3, [ 'awk',' perl' , 'ruby' ] ]
list.append('python')
dict = { 'awk':1, 'perl':[2,3], 'ruby':('diamond','carbon'), 'python':1.5 }
```

[1] It is assumed that the Python interpreter is already installed in your machine and Python is set in shell's search paths.

```
print(len(tuple), tuple[2])
print(len(list), list[0], list[2][0])
print(len(dict), dict['ruby'])
```

produces as output:

```
4 ruby
4 1 awk
4 ('diamond', 'carbon')
```

2.4 Control statements

Python has the usual primary control syntax, those are "for" or "while" statements as loop statements and if/else statements as conditional branching.

As a simple example of conditional branching, the program which substitutes a larger number between a and b for m is as follows.

```
if a > b:
    m = a
else:
    m = b
```

As an example of a loop, the program which puts the prime numbers up to 100 into a list is shown. The function range() is frequently used in the for loop statement, for example, range(i, j, k) returns the list of numbers which increase from i to (j-1) in k step. If k is omitted, it will be regarded as 1.

```
n=100
found=[]
for trgt in range( 2, n+1 ):
    for dv in range( 2, trgt+1):
        if dv >= trgt:
            found.append(trgt)
            break
        if trgt % dv == 0:
            break

print(found)
```

- **About program blocks**

  In Python, the depth of indentation expresses the block structure of program. For example, the indentation is used to make grouping of the processing statements in the if/else statement or the loop statement. Although each block has any depth of indentation, the indentation of one block needs to have the same depth. In the def or class statement as shown later, the program blocking using the indentation is needed.

2.5 Function and class

Like the following example, a new function can be defined with the def statement.

```
def fact(n):
    if (n<=1): return 1
    return fact(n-1)*n
```

The class, a user defined compound data type with methods, can be defined with the class statement. About defining new classes, you should refer to bibliographies.

2.6 Module

In Python, the classes or functions which were described in an external program file can be used by includeing the file. An external program file is called an extended library or an extended module. An extended module is not only a scripting file written in the Python scripting form, but a compiled dynamic link module written in C/C++. The import statement is used when an external extended module is loaded.

For example, when the above fact(n) function is saved to the file named factpy.py, the file is included as follows and the function becomes usable:

```
import factpy
```

The function can be called as follows:

```
print(factpy.fact(10))
```

If you want to miss the "factpy." of the called function name, the import statement is used as follows:

```
from factpy import *
```

It can be called as follows:

```
print(fact(10))
```

If there is no extended module in the executing directory of Python, it is necessary to set up the directory path which has the extended module to the environment variable PYTHONPATH.

2.7 Bibliography

Guido van Rossum. "Python Tutorial" http://www.python.org, March 22, 2000.

Mark Lutz "Programming Python" O'Reilly & Associates, Inc, October 1996.

David M. Beazley "Python Essential Reference" New Riders Publishing, 1999.

Python Language Home Page
Python can be downloaded from here. http://www.python.org/

Chapter 3

Using Python in OCTA

In OCTA, the operations of the UDF data take the main subject. The class for handling a UDF file easily from Python is offered as an extended module. The operations of the extended modules which can be used on OCTA are explained below.

3.1 About using Python in OCTA

The main data of a UDF file consists of the data name definition part and the data entity part, and the data entity part may have a large number of numeric values or strings. If a data file have more values, it will become more difficult to find the value to need from a big size array, for example. The UDFManager extended module will find out the value from a lot of data, if the data name or the set of the name and the array index are passed to the get() method.

The UDFManager extension module can be included by "import" command from the Python interpreter like an ordinary Python extension module. With executing commands as follows after starting a Python interpreter, you can get the desired value from a UDF data file.

```
from UDFManager import *
udfobj = UDFManager('UDF_file_path')
print(udfobj.get('data_name_and_array_index'))
```

You can use other methods, such as data editing or record movement, and others.

When the UDFManager extension module is used in GOURMET, a 3-dimensional graphical model can be drawn on the view screen of GOURMET. The same data manipulation methods of the UDFManager extension module can be used in GOURMET, too.

3.2 Extended module for UDF data operations *the UDFManager for Python*

The UDFManager module has several classes. The UDFManager class can read, write and operate a UDF data. The Location class is an auxiliary class for the string operations which allowed to handle a UDF variable name easily. Other classes are used for drawing on GOURMET.

Before using the methods of the UDFManager class, it is necessary to execute the import command for reading the extended module.

The extended module consists of UDFManager.py and UdfManagerPython.dll(so). UdfManagerPython.dll (so) is imported in UDFManager.py.

(1) UDFManager class

The methods of the UDFManager class are listed below.

- **UDFManager(udffilename)**

  The UDF file specified to udffilename is opened and a UDFManager object is created.

  ```
  (Example)
  from UDFManager import *
  uobj=UDFManager("/home/octa/test/simple_test_out.udf")  # on UNIX(Linux)
  uobj=UDFManager("D:/octa/test/simple_test_out.udf")     # on Window
  ```

- **totalRecord()**

  The total record count in a UDF file is returned.

- **currentRecord()**

  The current-record position is returned.

- **jump(record_no_or_label)**

  The target record is moved to the record position specified with the integer number or the label string. In specifying with the integer value, -1 is specified to move to the common data part. And to move to some record part, the integer value which counted from the first record at zero is specified. If the record movement is successful, the current record position number is returned, and if it cannot do, returns -2.

- **nextRecord()**

  The target record moves to the next record. If the record movement is successful, the current record position number is returned, and if it cannot do, -2 is returned.

- **newRecord(label="")**

  A new record with the name given with the label argument is appended at the last record and the name of the record is returned. The target record moves to the new record.

- **setRecLabel(label)**

  The label of a current record is changed to the given label.

- **getRecLabel()**

  The label of a current record is returned. If a current record is the common data part, None is returned.

- **seek(location)**

  The reference position of data moves to the location specified by the 'location' argument. The term location means the pair of a data name and an array index.

  ```
  (Example)
  uobj.seek("field.fraction[0].value[1]")

  The following is equivalent to the above:

  uobj.seek('field.fraction[].value[]',[0,1])
  ```

- **tell()**

  The current location is returned.

- **next()**

  The rightmost index included in the location of an array is increased by one.

- **prev()**

The rightmost index included in the location of an array is decreased by one.

- **type(location)**

  The UDF data type of the current location is returned as a string.

- **size(location)**

  If the specified location is an array, the number of elements is returned. When a scalar variable has a value, 1 is returned, and if it has no value, 0 is returned. When the specified location is wrong, None is returned.

  ```
  (Example)
  uobj.size("field.fraction[0].value[]")

  The following is equivalent to the above:

  uobj.size('field.fraction[].value[]',[0])
  ```

- **get(location,unit="")**

  Returns a value at the specified location. When the specified location is compound data type, all the values included in the compound data are stored to a list and returned. When the unit is specified, the values are converted into the unit. Here, the unit which can be specified must have the same unit dimension as the unit specified in the definition part of a UDF file. (Please refer to "UDF Syntax Reference" about the dimension of a unit.)

  A unit cannot be specified to the location which indicates a compound data. This method returns None, in the following case, the wrong data name is specified to the location, or the unit conversion cannot be done with the specified unit.

  ```
  (Example)
  uobj.get("field.fraction[0].value[1]")
  The following is equivalent to the above:
  uobj.get('field.fraction[].value[]',[0,1])

  uobj.get("field.fraction[0].value[1]","[m/s]")
  or:
  uobj.get('field.fraction[].value[]',[0,1],"[m/s]")
  ```

  When an array index is omitted, such as uobj.get("field.fraction[0].value[]"), all the array data is put in a list and returned. An array index is omissible only from right-hand side of a location.

- **getArray(location,unit="")**

  This method is identical in functionality to the get method. The value of the specified location is returned. None is returned if data does not exist. When an array index is omitted, all the array data is put in a list and returned.

- **put(value,location,unit="")**

  The value is set to the location. When a unit is specified, it is considered that the data has the value in the unit. The unit which can be specified must have the same unit dimension as the unit specified by the definition part of a UDF file.

  A unit cannot be specified to the location which indicates a compound data. If the value is appropriately arranged into a list, they can be set to a compound data or an array at once.

  ```
  (Example)
  uobj.put(2.34,"field.fraction[0].value[1]")
  uobj.put([1.1,2.3,3.4],'field.fraction[].value[]',[0])
  ```

- **insert(number,location)**

The number of empty data is inserted before the specified location.

- **erase(number, location)**

  The number of data is deleted after the specified location. When the deletion is successful, a zero is returned, and None is returned when it fails. To delete scalar data or scalar compound data, you have to specify a larger number than zero to the argument 'number'.

- **write([filename,record,mode])**

  Data is written to a file.

```
All the present data is overwritten to the same file, as follows:
uobj.write()

It writes into another file:
uobj.write("/home/octa/test/simple_test_work.udf")

Only the current record is written into another file:
uobj.write("/home/octa/test/simple_test_work.udf",currentRecord)

The current record data is appended to the tail of another file:
 (If there is no file, it will newly created.)
uobj.write("/home/octa/test/simple_test_work.udf",currentRecord,append)
```

  record=allRecord/currentRecord : All records or a current record is specified.
  mode=overwrite/append : Overwriting or appending is specified.

- **getIDList(class_name)**

  All the ID type values used for the objects of the class given by class_name are returned as a list.

```
(Example)
idlist = uobj.getIDList("Vertex")
```

- **getKeyList(class_name)**

  All the KEY type values used for the objects of the class given by class_name are returned as a list.

```
(Example)
keylist = uobj.getKeyList("AtomType")
```

- **getLocation(class_name,key_or_index)**

  A location of the data which has the same KEY/ID value as the specified value by key_or_index in the objects of the class given by class_name is returned.

```
(Example)
loc = uobj.getLocation("AtomType",keylist[0])
```

- **About operations for header informations**

  getEngineType()

  getEngineVersion()

  getIOType()

  getProjectName()

  getComment()

  getAction()

These methods return the UDF header informations.

setEngineType(engine)

setEngineVersion(version)

setIOType(io)

setProjectName(project)

setComment(comment)

setAction(comment)

You can modify the UDF header informations by these methods.

- **About operations for a unit**

unitConvert(to_unit,from_unit,value=1.0)

The given value is converted into to_unit from from_unit, and returned. Not only a numeric value but a list of numeric values can be given to the value argument.
(Example) getUnitValue('[erg]','[J]',[1.1,2.1])

unit(variable_name,set_unit_name='')

The unit assigned to a UDF data variable is returned or changed. Without set_unit_name, if the variable specified by variable_name has a unit, the unit name will be returned. With set_unit_name, the unit of the variable is changed to the given unit name. The units before and after changing must have the same unit dimensions.

getUnit(unit_name)

A unit definition described in the unit definition part of a UDF file is returned. A unit name is specified by unit_name, which is enclosed in square brackets, for example, getUnit("[sigma]").

setUnit(unit_name,unit_expression)

A unit definition is newly created or changed. The full expression or the coefficient part of a unit definition is specified at unit_expression.
(Example)
setUnit('[sigma]','1.2[nm]')
If [sigma] has been defined as '1.1 [nm]', you can change 1.1 to 2.1, like this:
setUnit('[sigma]',2.1)

- **About the operations of a unit system**

allUnitSystem()

All the name of the unit system which can be used for unit conversion is returned as a list of strings.

getUnitSystem()

The current unit system name is returned.

setUnitSystem(unit_system)

A unit system is changed.

readUnitSystemFile(unit_system_path)

A new unit system definition file is read. (Please refer "UDF Syntax Reference" about a unit system definition file.)

- **file()**

The path of a current UDF file is returned.

- **About the change of the record part and the common part**

  common_start()

  common_end()

  The methods for getting values, such as size(), get(), getArray(), getIDList(), getKeyList() and getLocation(), search for the current record first, and if there is value, the value will be returned. But if there is no value in the current record, the common data part will be searched for next. When a variable has its values in both a current record and the common data part and you want to get only values in the common data part, the target to search can be made only into the common data part by common_start() method. If common_end() is used, the target will return to the original record.

(2) Location class

In treating data with the methods of the UDFManager class, the string operation of a data name may become complicated. Using Location class, string operations of a data name can be easier.

The following UDFManager methods can take a Location object as arguments directly:
put, get, getArray, insert, erase, seek.

The methods of Location class is shown here.

The following terms are used about "location". For example, in the expression of a location "A.B[0].C[1]", the top location of "A.B[0].C[1]" is "A", the parent location of "A.B[0].C[1]" is "A.B[0]", "A.B[0].C[1]" is the child location of "A.B[0]", the path of "A.B[0].C[1]" is "A.B[].C[]", and the index of "A.B[0].C[1]" is [0,1].

- **Location(path_and_index)**

  The instance of the Location class is generated:
  loc = Location('a[].b[]',[0,0])
  or
  loc = Location('a[0].b[0]')

- **str()**

  The data location string which may contain the array index is returned.

- **root()**

  The target location is changed to the top of the current location which is a compound data type.

- **seek(location_index_args)**

  A given location is set as the target of processing.

- **up()**

  A target location is moved to the parent of the current location which is a (nested) compound data type.

- **down(location_index_args)**

  A target location is moved to the child of the current location which is a (nested) compound data type.

- **prev()**

  The rightmost index of the current location is decreased by one.

- **next()**

  The rightmost index of the current location is increased by one.

- **getPath()**

  The path of the current location is returned.

- **getIndex()**

  The index list of the current location is returned.

Examples of Location class

```
>>> from UDFManager import *
>>> loc=Location('a[].b[]',[0,0])
>>> print(loc.str())
a[0].b[0]
>>> loc.next()
a[0].b[1]
>>> loc.up()
a[0]
>>> loc.down('c[0]')
a[0].c[0]
>>> loc.getPath()
'a[].c[]'
>>> loc.getIndex()
[0, 0]
```

3.3 Operations of Python in GOURMET

Python can be executed in the Python scripting window of GOURMET. Therefore, the UDFManager class can be operated like the usual Python command window. About a UDF file opened currently in GOURMET, the methods of the UDFManager class can be used in the special way. In the following, those special operations and the usage of the sheet handling function for graph drawing are described.

3.3.1 How to use of the UDFManager for Python in GOURMET

First, there is no need of executing the "import" command for importing a UDFManager extened module. Because the import command is executed automatically at the time of GOURMET starting. Furthermore, for a UDF file opened currently in GOURMET, the UDFManager object of the name _udf_ is generated automatically. In executing the methods of the UDFManager class to a UDF file opened currently in GOURMET, it is not necessary to describe a UDFManager object _udf_, that is, the methods can be called just like functional forms.

There are the following other extensions and changes in operating the UDFManager class in GOURMET.

(1) $ extension

Instead of passing a location as the arguments to the get(), getArray() and put() methods of the UDFManager class, attaching $ to the top of a location can be substituted for those methods. For example:

data = getArray("a[].b",[1])

the above can be written as follows,

data=$a[1].b

Where, since the UDF variable with $ extension is not a variable of Python interpreter, the convenient slice operator for a list variable in Python cannot be used for it.

(2) Other available methods in GOURMET

focusData(datapath)

The data name specified by the datapath is displayed on the editor screen of GOURMET.

The sheet handling methods (functions) and the drawing methods (functions) are also available only in GOURMET.

(3) Change for functional usage

When a method is used like a function, there are some methods which need to change its name called as functions. Since some method names may overlap the functions of the Python interpreter, it is necessary to make a method name and a function name different only in the following cases.

Instead of the type() method, udftype() is used as a function.

3.3.2 The sheet handling functions for graph drawing

GOURMET has the features which easily displays a graph using Gnuplot. The sheet for graph drawing has the table form structure, and by the sheet handling functions of the UDFManager class, values can be set to the sheet or the value of the sheet can be got or edited.

If some UDF file is opened in GOURMET, the sheet for graph drawing named "GraphSheet[]" is automatically displayed on the data view. About "GraphSheet[]", using the sheet handling functions shown below, new variables can be added as a column, and values can be set to them.

- **Defining columns**

  createSheetCol(col,name,size=0,type='float')

  col : This is a column number starting at 0. When the specified column number has already existed, the column is overwritten. When more numbers than the number of the present columns are specified, the column is added at the last.

  name : This specifies the label of a column to add.

  size : This specifies the size of rows. Even if this is zero, the rows can be added later.

  type : This specifies the data type of the column. A number type is good at the default 'float'. The 'string' is used only for string data.

- **Delete columns**

  deleteSheetCol(col_or_name)

  col_or_name : The label string or the column number of a column to delete is specified.

- **Insertion of rows**

  insertSheetRow(index,number)

  index : The row number of the row position which inserts rows is specified, the row number starts at zero. The number rows are inserted just before this row position for all columns.

  number : This specifies the number of the rows to insert.

- **Delete of rows**

  deleteSheetRow(index,number)

  index : This specifies the row number of the start row to delete. The number rows are deleted for all columns.

- **Values set for each column**

  setSheetCol(col_or_name,location_index_or_data_list)

  col_or_name : A label string or a column number of a column is specified.

  location_index_or_data_list : A list of values or a data name is specified.

- **Value set for each cell**

  setSheetData(col_or_name,row,location_index_or_data)

col_or_name : A label string or a column number is specified.

row : A row number is specified.

location_index_or_data_list : A value set to a cell is passed. A value enclosed square brackets or a UDF variable name is specified.

- **The number of columns and the number of rows are returned, respectively.**

  getSheetColSize()

  getSheetRowSize()

- **Getting values**

  value_list = getSheetCol(col_or_name)

  All the values of the specified column are returned as a list.

  value = getSheetData(col_or_name,row)

  Returns the value of a cell specified with the column and row.

- **The examples of the sheet handling functions:**

  In the following example, two columns are created and the values of the variable gr are substituted for the GraphSheet[].

```
gr=[[1,2],[3,4],[5,6],[7,8],[9,10],[11,12],[13,14]]
n = len(gr)
createSheetCol(0,'r')
createSheetCol(1,'gab')
for i in range(0,n):
    rsize=getSheetRowSize()
    if rsize <= i:
        insertSheetRow(rsize,1)
    setSheetData(0,i,[gr[i][0]])
    setSheetData(1,i,[gr[i][1]])
$GraphSheet[].r = [1,2,3,4]
sdata = $GraphSheet[].r
```

3.3.3 Drawing functions

Using the drawing functions shown below in GOURMET, you can draw on the 3D object window of the viewer screen of GOURMET.

The followings explaine the arguments of the drawing functions introduced here.

coordinateN(N is integer) is either a list type data with 3-dimensional coordinate value such as [x, y, z], or a location string with $ at the head, the location indicates the compound data of (x, y, z) values.

coordinate_list is a list type data with 3-dimensional coordinate value such as [x, y, z].

attribute_id is an integer value or a list of floating values, which specifies the display attributes for drawing.

3.3.3.1 Introduction of basic drawing functions

- **Drawing a line**

  line(coordinate1,coordinate2,[r,g,b,t])
  line(coordinate1,coordinate2,attribute_id)

- **Drawing a point**

  point(coordinate,[r,g,b,t])
  point(coordinate,attribute_id)

- **Polygon**

  polygon(coordinate_list,[r,g,b,t])
  polygon(coordinate_list,attribute_id)

- **Polyline**

  polyline(coordinate_list,[r,g,b,t])
  polyline(coordinate_list,attribute_id)

- **Disk**

  disk(coordinate1,[r,g,b,t,radius,vx,vy,vz])
  disk(coordinate1,attribute_id)

- **Drawing a ellipse with one center point**

  ellipse1(coordinate1,[r,g,b,t,a,b,vx,vy,vz])
  ellipse1(coordinate1,attribute_id)

- **Drawing a ellipse with two focus points**

  ellipse2(coordinate1,coordinate2,[r,g,b,t,a,vx,vy,vz])
  ellipse2(coordinate1,coordinate2,attribute_id)

- **Cylinder**

  cylinder(coordinate1,coordinate2,[r,g,b,t,radius])
  cylinder(coordinate1,coordinate2,attribute_id)

- **Sphere**

  sphere(coordinate1,[r,g,b,t,radius])
  sphere(coordinate1,attribute_id)

- **Drawing a ellipsoid with one center point**

  ellipsoid1(coordinate1,[r,g,b,t,a,b,c,vx,vy,vz])
  ellipsoid1(coordinate1,attribute_id)

- **Drawing a ellipsoid with two focus points**

  ellipsoid2(coordinate1,coordinate2,[r,g,b,t,a])
  ellipsoid2(coordinate1,coordinate2,attribute_id)

- **Tetrahedron**

  tetra(coordinate1,coordinate2,coordinate3,coordinate4,[r,g,b,t])
  tetra(coordinate1,coordinate2,coordinate3,coordinate4,attribute_id)

- **Cube**

  cube(coordinate1,length,[r,g,b,t])
  cube(coordinate1,length,attribute_id)

  coordinate1 is a central coordinate and length is a length of one side.

- **Arrow which has an arrowhead at coordinate2**

  arrow(coordinate1,coordinate2,[r,g,b,t,radius,height])
  arrow(coordinate1,coordinate2,[r,g,b,t,radius,height,magnification])
  arrow(coordinate1,coordinate2,attribute_id)

- **Text**

  text(coordinate,contents,[r,g,b,t,size])
  text(coordinate,contents,attribute_id)

- **Other drawing operations**

  clearDraw()

All the graphic objects before calling this are erased.

appendDraw()

Within one Python execution, the drawings after calling this are overwritten.

clearDrawMode()

It enters the mode in which drawings erase for each Python execution. This mode is the default.

appendDrawMode()

It enters the mode in which all drawings are overwritten.

3.3.3.2 About drawing contour

- **Drawing contour cross section and isosurface**

  contour(element_type,node_number,node_coordinate_list,node_value_list)

  This makes the element for creating the piece of a contour.

Element	element_type	node_number
Triangle	"triangle"	3
Quadrangle	"quad"	4
Tetrahedron	"tetra"	4
Hexahedron	"hexa"	8

Table 3.1: Contour element type

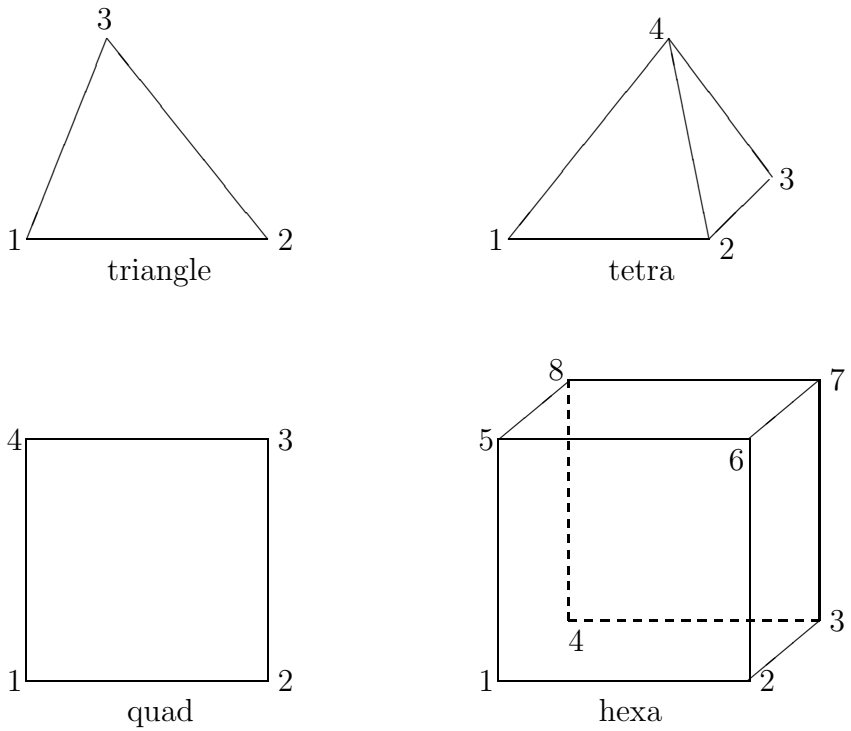

Figure 3.1: Node order of a contour element

The value in Table 3.1 is used for specifying element_type and node_number. The node order of a element is shown in Figure 3.1 .

The example of a contour function is as follows.

contour("triangle", 3, [[0,0,0],[1,0,0],[0,1,0]],[1.0,2.0,3.0])

- **Specification of a isosurface**

  clevel(min_value,max_value,attribute_id)

  A isosurface is created from the points with the value between min_value and max_value in the elements specified with the contour function.

- **Cancel of isosurfaces**

  delete_clevel()

  All specifications by the clevel() function called before this are canceled.

- **Specification of the maximum and the minimum value which are drawn on a contour cross section.**

  crange(min_value,max_value)

  The value of each node specified with the contour function is set the value below min_value to min_value and the value more than max_value to max_value.

  Note:

  1. When two or more crange() are called, the last call is used.
  2. In min_value >max_value, it is ignored.

- **Specification of a cutting plane**

  cplane(point_on_plane,normal_vector)

  This specifies the cutting plane on which a contour is drawn. The plane given here cuts the solid elements specified by the contour() function or generated from the below-mentioned meshfield.

  Note: There is no need of specifying a cutting plane for 2-dimensional flat elements, such as triangles or quadangles, but specifying color is needed using the ccolor function described later.

- **Cancel of cutting planes**

  delete_cplane()

  All cutting planes specified by the cplane() function called before this are canceled.

- **Gradating contour**

  When the colors corresponding to the maximum and minimum values are given respectively, the colors displayed on a contour change gradually between the color of minimum and the color of maximum. Those colors correspond to the values given to the nodes of elements.

  ccolor([minR,minG,minB,minT,maxR,maxG,maxB,maxT])

  Here minR, minG, minB and minT are the RGB values and transparency for the minimum value, which values are between 0 and 1. The maxR, maxG, maxB and maxT are the RGB values and transparency for the maximum value.

  The above calling can be described as follows using the display attribute number to the ccolor() functions.

  ccolor(attribute_id)

  Note: When two or more ccolor() are called, the set value of the last call is used.

- **Discrete contour(color specification corresponding to the value)**

  dcolor(value_list,attribute_id_list)

  or

  dcolor(value_list,color_list)

  This specifies colors corresponding to each values.

  (sample 1) dcolor([0,1,2],[1,2,3]) # with attribute ID

  (sample 2) dcolor([0,1,2],[[1,0,0,1],[0,1,0,1],[0,0,1,1]]) # with RGB

  Note: When two or more dcolor() are called, the set value of the last call is used.

3.3.3.3 Meshfield function

The contour() function described above creates one element (triangle, quad, tetra and hexa). On the other hand, the meshfield function creates many elements at one time from a mesh form efficiently.

(1) The meshfield function generates mesh objects.

(1-1) Regular mesh(structured mesh)

mesh_obj_in_python = meshfield(mesh_type, coord_list_list, div_list)

mesh_type : This specifies the type of a mesh. A type name uses the following, and 4 characters of the beginning are used.

"regular", "rectangular", "sphere"(2-dimensional polar coordinate is used), "cylinder"

coord_list_list : Coordinate data of a mesh is given in the following form for each mesh type.

With "regular", the maximum and minimum coordinates is set as a Python's list.
[[xmin, xmax],[ymin, ymax],[zmin, zmax]]
With "sphere", the minimum and maximum of (r, θ) are specified, respectively.
[[rmin, rmax],[thetamin, thetamax]]
With "cylinder", the minimum value and maximum of (r, θ, z) are specified, respectively.
[[rmin, rmax],[thetamin, thetamax],[zmin, zmax]]
With "rectangular", the coordinates of a starting point, dividing points and an end point are given as Python's lists.
[[xmeshPosition0, xmeshPosition1, ...],[ymeshPosition0,...], ...]

div_list : The numbers of partitions are specified in the following form for each mesh type.

With "regular" or "rectangular", the number of partitions of (x, y, z) is specified, respectively.
[x_npart, y_npart, z_npart]
With "sphere", the number of partitions of (r, θ) is specified, respectively.
[r_npart, theta_npart]
With "cylinder", the number of partitions of (r, θ, z) is specified, respectively.
[r_npart, theta_npart, z_npart]

(1-2) Unstructured mesh

mesh_obj = mesh(element_type, vertex_coord_list, element_vertex_list)

element_type : This specifies the type of a element.

A type name uses the following, and 4 characters of the beginning are used.

"triangle" : triangle element
"quad" : quadrangle element
"tetra" : tetrahedron element
"hexa" : hexahedron element

vertex_coord_list : This argument specifies the indexes and the coordinates of nodes (vertexes) as a list.

Coordinates can be given as a list of coordinate values or a location of UDF data.

[[1, [x1, y1, z1]],[2, [x2, y2, z2]],[3, [x3, y3, z3]],...,[N, [xN, yN, zN]]]

When the location (here for example, "position") of the compound data which has coordinates (x, y, z) as it's children is given, the coordinate values (x y, z) in the compound data can be got in a meshfield object.

[[1, mesh.data.vertex[0].position],[2, mesh.data.vertex[1].position],

[3, mesh.data.vertex[2].position],.....,[N, mesh.data.vertex[N-1].position]]

The indices are integer and those order is free (no need of sorting in ascending order or descending order).

element_vertex_list : In order to show the connective order of the nodes which forms an element, the indexes of the nodes are passed as a list.

With "triangle", for example:

[[1, 2, 3],[5, 6, 7],[9, 10, 11],.....,[2, ,3, N]]

With "quad" or "tetra", for example:

[[1, 2, 3, 4],[5, 6, 7, 8],[9, 10, 11, 12],.....,[1, 2, 3, N]]

(2) Values are set to each node.

(2-1) The index and the value (scalar or vector) of each node are set by the set() method.

mesh_obj_in_python.set(index_list, data_value)

(2-1-1)In the regular mesh(structured mesh)

A value is set by a grid index. A grid index is the set of numbers which enumerated the dividing points in the meshed grid from the origin.

- **For scalar value**

  A value 1.2 is set to the node of the dividing point at the 0th of X direction, the 1st of Y direction and the 2nd of Z direction:

  meshf.set([0,1,2],1.2)

- **For vector value**

  meshf.set([0,1,2],[1.2,3.5,0.8])

(2-1-2)In the unstructured mesh

A value is set with a node index.

- **For scalar value**

  meshf.set(1,2.2)

- **For vector value**

  meshf.set(1,[1.2,3.5,0.8])

(2-2) The list of indexes and the list of values (scalar or vector) of each node are set by the field() method.

mesh_obj.field(index_list_list, data_value_list)

index_list_list : For a regular mesh, grid indexes are specified, like this:

[[0,0,1], [0,0,2], [0,0,3],...,[0,0,Nz-1],..., [Nx-1,Ny-1,Nz-1]]

For an unstructured mesh, the list of node indexes is specified, like this:

[1,2,3,4,..., N]

data_value_list : For scalar values, the list of values is specified as follows:

[1.2, 2.0, ..., 5.0]

For vector values, the list of lists as a vector value is specified as follows:

[[1.2,3.5,0.8],[1.0,3.0,0.5],....,[2.2,1.5,1.8]]

Or you may give the list of data locations directly.

[field.volume_fraction.value[0], field.volume_fraction.value[1],

field.volume_fraction.value[2],.....,field.volume_fraction.value[NxNyNz-1]]

Note: The index_list_list is omissible, such as index_list_list=None. That leads to the same result as the node indexes given in creating a meshfield object is specified in ascending order to the index_list_list.

(3) How to get the value of each node

The scalar value of each node is got by the get() method, and the vector value is got by the getv() method.

(3-1)In the regular mesh(structured mesh)

- **For scalar value**

  mesh_obj.get([i,j,k])

- **For vector value**

  mesh_obj.getv([i,j,k])

  The getv() returns a vector value by Python list form.

The indexes list [i, j, k] shows the dividing point of the XYZ direction.

(3-2)In the unstructured mesh

Values are got with the node index as follows.

- **For scalar value**

  mesh_obj.get(id)

- **For vector value**

  mesh_obj.getv(id)

  The getv() returns a vector value by Python list form.

(4) The contour drawing methods of the meshfield

The meshfield class has the methods corresponding to the functions introduced about the contour drawings in Section 3.3.3.2.

- **Specifying an isosurface**

  mesh_obj.clevel(min_value,max_value,attribute_id)

- **Cancel of isosurfaces**

  mesh_obj.delete_clevel()

- **Specifying the maximum and the minimum value which are drawn on a contour cross section.**

  mesh_obj.crange(min_value,max_value)

- **Specifying a cutting plane on which a contour is drawn**

  mesh_obj.cplane(point_on_plane,normal_vector)

  mesh_obj.cplane(point_on_plane, normal_vector, attribute_id)

- **Cancel of cutting planes**

  mesh_obj.delete_cplane()

- **Color specification for minimum and maximum values**

  mesh_obj.ccolor(attribute_id)

- **Color specification corresponding to the value**

  mesh_obj.dcolor(value_list,attribute_id_list)

  or

  mesh_obj.dcolor(value_list,color_list)

- **Specifying which legend to display**

  mesh_obj.legend()

  As for the legend which shows correspondence of values and colors in a contour, only one is displayed on the right end on the view screen of GOURMET. In the following case, two or more legendes are created.

  (1) Two or more meshfield objects are generated.

  (2) The function ccolor() or dcolor() is called two or more times.

  (3) To a meshfield object, the ccolor() or dcolor() method is called and more ccolor() or dcolor() function is called.

  The legend() method specifies one legend which you want to display from two or more legendes.

  The priority of a legend display is as follows.

  (1) Specification by the legend() method

  (2) The legend generated from the function dcolor()

  (3) The legend generated from the function ccolor()

  (4) The legend of the meshfield which latest called the draw() method.

  Note : When two or more legend() are called, the legend of the meshfield which latest called the draw() method is displayed.

(5) Specifying the drawing conditions and the contour generation

field_obj_in_python.draw(.....)

Executing this method generates a contour or an isosurface. The following drawing conditions are specified as arguments.

1. The number of skips in the case of skipping nodes of a mesh can be given.

   skip = n

2. If drawing the frame lines of a mesh, the drawing attribute of a line is specified. The drawing attribute number or the property list of a line is specified as follows. (The drawing attribute is introduced in the following subsection.)

   frame = line_attribute_id

   or

   frame = [red, green, blue, transparency]

3. If displaying a vector value by an arrow, the drawing attribute of an arrow is specified.

   arrow = arrow_attribute_id

   or

   arrow = [red, green, blue, transparency, arrowhead_radius, arrowhead_height]

3.3.3.4 Drawing attribute

The display attribute, such as color or size of objects to draw, is shown by the drawing attribute values specified to a drawing function. The drawing attribute is specified by a integer value as a drawing attribute number or a drawing property list.

The drawing attribute numbers get related with the drawing property lists by a drawing attribute file. If a drawing attribute file is not specified, the default values are used for drawings.

The format of a drawing attribute file is described below.

object keyword
ID property values
ID property values
...

The drawing object keywords are shown in following (1), and the drawing property list is shown in (2).

(1) Keywords
The keyword corresponding to the drawing object type is used.

LineAttr : Line(segment)
PointAttr : Point
PolygonAttr : Polygon
PolylineAttr : Polyline
DiskAttr : Disk
Ellipse1Attr : Ellipse(specifying a center)
Ellipse2Attr : Ellipse(specifying two focuses)
CylinderAttr : Cylinder
SphereAttr : Sphere
Ellipsoid1Attr : Ellipsoid(specifying a center)
Ellipsoid2Attr : Ellipsoid(specifying two focuses)
TetraAttr : Tetrahedron
CubeAttr : Cube
ArrowAttr : Arrow line
TextAttr : Text
CContourAttr : Gradating contour
DContourAttr : Discrete contour
ClevelAttr : Isosurface

(2) Properties list
* The values of a color (RGB, transparency) use the value between 0 and 1.

LineAttr : ID, color (RGB, transparency)

PointAttr : ID, color (RGB, transparency)

PolygonAttr : ID, color (RGB, transparency)

PolylineAttr : ID, color (RGB, transparency)

DiskAttr : ID, color (RGB, transparency), radius, normal vector (XYZ)

Ellipse1Attr : ID, color (RGB, transparency), length of axes(a,b), direction vector of axis a(XYZ), normal vector (XYZ)

Ellipse2Attr : ID, color (RGB, transparency), length of the major axis(a), normal vector (XYZ)

CylinderAttr : ID, color (RGB, transparency), radius

SphereAttr : ID, color (RGB, transparency), radius

Ellipsoid1Attr : ID, color (RGB, transparency), length of axes(a,b,c), direction vector of axis a(XYZ), direction vector of axis c(XYZ)

Ellipsoid2Attr : ID, color (RGB, transparency), length of the major axis(a)

TetraAttr : ID, color (RGB, transparency)

CubeAttr : ID, color (RGB, transparency)

ArrowAttr : ID, color (RGB, transparency), arrowhead radius, arrowhead height, (scale factor of length (note))

TextAttr : ID, color (RGB, transparency), PointSize

CContourAttr : ID, color for minimun value (RGB, transparency), color for maxium value (RGB, transparency)

DContourAttr : ID, color (RGB, transparency)

ClevelAttr : ID, color (RGB, transparency)

The attribute numbers, ID, should have different integer values in the same drawing object type keyword area. If the same ID value are in one drawing object type area, the first description in a attribute file is used.

(Note) the "length scale factor" is optional. It is 1, if it is not specified. It is not described in a drawing attribute file.

The rule of a drawing attribute file is as follows:

1. Values are separated by one or more whitespaces.
2. A comment line is started with #.
3. If there are the same ID value in one drawing object type area, the first attribute value is used.
4. Default values is used when there is no drawing attribute file or the specified ID is not found.

The example of a drawing attribute file is shown below.

```
### Shape Drawing Attribute ###
LineAttr
# ID R G B TRANS
1 1.0 0.0 0.0 1.0
2 0.0 1.0 0.0 1.0
3 0.0 0.0 1.0 1.0
4 1.0 1.0 0.0 1.0
5 1.0 0.0 1.0 1.0
6 0.0 1.0 1.0 1.0
7 0.0 0.0 0.0 1.0
########################
PointAttr
# ID R G B TRANS
1 1.0 0.0 0.0 1.0
2 0.0 1.0 0.0 1.0
3 0.0 0.0 1.0 1.0
4 1.0 1.0 0.0 1.0
5 1.0 0.0 1.0 1.0
6 0.0 1.0 1.0 1.0
7 0.0 0.0 0.0 1.0
```

About how to pass the drawing properties list directly to a drawing function as arguments, the numeric values except the attribute ID of the properties list are wrapped into a list and are passed to a drawing function.

The examples are shown below:

```
line([1,1,1],[2,2,2],[1,0,0,1])
arrow([0,0,0],[0,0,1],[1,0,0,1,0.1,0.1])
arrow([0,0,0],[1,0,0],[1,0,0,1,0.1,0.1,2]) # With an optional length scale factor.
# For the draw method of meshfiled.
meshf.draw(frame=[0,1,1,0.6],arrow=[1,0,0,1,0.1,0.1,1.5])
```

3.3.3.5 Cognac drawing class

The Cognac drawing class is a Python extension class for rapid drawing the UDF data of COGNAC in GOURMET. COGNAC is a general purpose coarse-grained molecular dynamics program.

If a UDF data has the data structure shown below, the Cognac drawing class can be used for the UDF data file.

Available data structure

In order to use the Cognac drawing class, molecule data, atom data and bond data need to have the following relations.

```
molecule array--+--(1)The name of a molecule (string type)
                |--(2)atom array--+--(3)The type of an atom (string or integer type)
                |                 +--(4)The coordinate of an atom--+-- X (float or double type)
                |                                                  |-- Y (float or double type)
                |                                                  +-- Z (float or double type)
                +--(5)bond array--+--(6)The type of a bond(string or integer type)
                                  |--(7)The array index of an atom attached to
                                  |     the starting point of the bond (integer type)
                                  +--(8)The array index of an atom attached to
                                        the end point of the bond (integer type)
```

The UDF data name corresponding to (1) - (8) of the above figure is passed to the constructor of the COGNAC drawing class.

In fact, these data do not need to be gathered in one compound data, for example, "atom array" and "bond array" may be under another compound data. The relation among a molecule, an atom and a bond should just be shown by the indexes of those arrays.

For example, a molecule at index [0] has a bond at index [0,0] and atoms at indexes [0,0] and [0,1], that is, the indexes of atoms of the ends of a bond at [0,0] are [0,0] and [0,1].

The following UDF data names are specified in COGNAC.

```
(1)molecule_type_path='Set_of_Molecules.molecule[].Mol_Name'
(2)atom_data_path='Set_of_Molecules.molecule[].atom[]'
(3)atom_type_path='Set_of_Molecules.molecule[].atom[].Atom_Type_Name'
(4)atom_position_path='Structure.Position.mol[].atom[]'
(5)bond_data_path='Set_of_Molecules.molecule[].bond[]'
(6)bond_type_path='Set_of_Molecules.molecule[].bond[].Potential_Name'
(7)bond_atom1_id_path='Set_of_Molecules.molecule[].bond[].atom1'
(8)bond_atom2_id_path='Set_of_Molecules.molecule[].bond[].atom2'
```

The following sample is shown one molecule which consists of two atom, and the atom's indexes are [0,0] and [0,1].

```
\begin{def}
Set_of_Molecules:{
    molecule[]:{
        Mol_Name:string
        atom[]:{
            Atom_Name:string
```

```
        }
        bond[]:{
            Potential_Name:string
            atom1:int
            atom2:int
        }
    }
}
Structure:{
    Position:{
        mol[]:{
            atom[]:{
                x:float, y:float, z:float // coordinate
            }
        }
    }
}
\end{def}
\begin{data}
Set_of_Molecules:{
   [
       {
           "molecule1",
           [ {"atom1"},{"atom2"} ]
           [ { "bond1", 0, 1 } ]
       }
   ]
}
Structure:{
    {
        [
            {
                [ {0,0,0}, {1,1,1} ] // coordinate
            }
        ]
    }
}
\end{data}
```

Classes and methods

Cognac drawing class consists of an atom drawing class, a bond drawing class and a class for operating coordinates of atoms.

Atom drawing class

- **Object creation**
  An atom drawing object is created by the createAtom() method of the UDFManager class.
  atom=createAtom(molecule_type_path,atom_type_path,atom_data_path,atom_position_path)

  molecule_type_path : A UDF data path of a molecule name is specified. This is required for the classification by color for each molecule.(optional)

  atom_type_path : A UDF data path of an atom name is specified. This is required for the classification by color for each atom.(optional)

  atom_data_path : A UDF data path of atom compound data is specified.

  atom_position_path : A UDF data path of the position-coordinate of an atom is specified. This is indispensable.

- **Drawing method**
  atom.draw(index=[],attr=-1,type='line',moltypedic=0,atomtypedic=0,coord=0)

  index : The array indexes of atoms to draw are specified. Specifying [] (an empty list) draws all atoms, and [i] draws the atoms belonging to the i-th molecule, and [i, j] draws one atom at [i,j].

attr : This specifies drawing colors. This is either a drawing attribute inumber, a drawing property list or following keyword: "atom","molname,"mol". When a drawing attribute number or a drawing property list is specified, an atom is drawn with the drawing attribute. When "atom" is given, an atom is drawn with a different color for a different atom name, and if atomtypedic is then given, an atom will be drawn with the color specified by the dictionary object. In the case of "mol", even if a molecule name is the same as other molecules, the atoms of a different molecule are drawn with a different color. In "molname", an atom is drawn using a different color for a different molecule name or the attribute of moltypedic.

type : Drawing shape type is specified. One of the following words can be used : "point","line","ball","ball-stick","stick","rod","volume".

The "point" draws a point, and the "ball" and "ball-stick" draws a sphere as an atom.

As for "line", a point is drawn only in a monoatomic molecule, and a sphere is drawn for "stick", "rod" or "volume" in the same case.

moltypedic : This is optional.
The drawing attributes for molecule names (specified by the molecule_type_path) are given by the Python dictionary form.

atomtypedic : Optional. The drawing attributes for atom names (specified by the atom_type_path) is given by the Python dictionary form, for example:
{"atom1":1,"atom2":2}.

coord : Optional. An object of the CognacCoord class is specified. It is used to change the coordinates of atoms at the time of drawing. See **Coordinate mapping**.

- **Methods for getting a coordinate**

1. The coordinate of an atom is returned by a list
   atom.getCoord(index)
2. The average of all coordinates or the average of the coordinates of atoms in a molecule is returned by a list.
   atom.averageCoord(index=())
   Average of all coordinates : xyz = atom.averageCoord()
   Average of the coordinates of atoms in a molecule : xyz = atom.averageCoord([3])

Bond drawing class

- **Object creation**

A bond drawing object is created by the createBond() method of the UDFManager class.
bond=createBond(molecule_type_path,atom_type_path,atom_data_path,atom_position_path, bond_type_path,bond_atom1_id_path,bond_atom2_id_path,bond_data_path)

atom_type_path,atom_data_path,atom_position_path : These are the same as the atom.

bond_type_path : A UDF data path of a bond name is specified. This is required for the classification by color for each bond name.

bond_atom1_id_path,bond_atom2_id_path : UDF data paths of the index data of the atom attached to the starting and the end of a bond is specified respectively.

bond_data_path : A UDF data path of a bond compound data is specified.

- **Drawing method**

bond.draw(index=[],attr=-1,type='line', moltypedic=0,atomtypedic=0,bondtypedic=0,coord=0,maxlen=0.0)

index : The array indexes of bonds to draw are specified. Specifying [] draws all bonds, and [i] draws the bonds belonging to the i-th molecule, and [i, j] draws one bond at index [i,j].

attr : This specifies the drawing color. This is eather a drawing attribute number, a drawing property list or following keyword: "atom","bond","molname","mol". When a drawing attribute number or a drawing property list is specified, bonds are drawn with the specified drawing attribute. When "atom" is given, bonds are drawn with a different color for a different atom name, and if atomtypedic is then given, a bond will be drawn with the color specified by the dictionary object. When "bond" is given, bonds are drawn with a different color for a different bond name, and if there is bondtypedic, it is similar to the atomtypedic. In the case of "mol", even if a molecule name is the same as other molecules, the bonds of the different molecule are drawn with the different color. In "molname", bonds are drawn using a different color for each molecule name or the attribute of moltypedic.

type : Drawing shape type is specified. One of the following words can be specified to this: "line"(line),"ball-stick","stick","rod"(rod),"volume"(ellipsoid)

The "line" draws a segment, and the "ball-stick", "stick" and "rod" draws a cylinder.

When "volume", the ellipsoid which uses the ends of a bond as its focus points is drawn.

bondtypedic : Optional. The drawing attributes for bond names (specified by the atom_type_path) is specified in the Python dictionary form.

coord : Optional. An object of the CognacCoord class is given.

maxlen : Optional. The bonds are not drawn, if the bond length is more than the maxlen and the maxlen is larger than 0.0.

Coordinate mapping

To draw an atom at the different position from the original coordinate, the index of an atom to change a position and the coordinate after change are stored in this mapping object.

- **Object creation**
  A coordinate mapping object is created by the CognacCoord() method of the UDFManager class.
  coordmap=CognacCoord()
  The mapping object is created. None is returned if it fails.
- **Storing of an index and a coordinate**
  coordmap.set(index,coord)

  index : The array index used as the key for search, in [i, j] or (i, j) form.

  coord : The 3-dimensional coordinate value of the drawing position after change, in [x, y, z] or (x, y, z) form.
  The dimension of index is returned if it succeeds, and None is returned if it fails.
- **The number of coordinates stored in a mapping object is returned.**
  nsize = coordmap.size() nsize = len(coordmap)
- **A coordinate is got from an index**
  coord = coordmap.get(index)

Chapter 4

Samples

This chapter introduces some samples of the UDFManager class.

4.1 Example of the UDFManager for Python

The following simple UDF data is red and edited using the methods of the UDFManager class.

```
sample.udf
\begin{def}
     class Point:{
        x:float
        y:float
        z:float
    }
    class Vertex:{
        id:ID
        position:Point
    }
    class Face: {
        id : ID
        vertex[] :<Vertex,ID>
    }
    parameter: {
        vertex[]:Vertex
        face[]:Face
value[]:float
    }
\end{def}
\begin{data}
parameter:{
    [ { 1,{0,0,0} }, { 2,{1,1,1} }, { 3, { 2,1,2} }, { 4,{3,1,2} }, { 5,{4,3,2} } ]
    [ { 10, [ 1,2,3 ] }, { 11, [ 2,3,4 ] }, { 12,[ 3,4,5 ] } ]
    [ 1,2,3,4,5,6,7,8 ]
}
\end{data}
\begin{record}{"Step 1"}
\begin{data}
parameter:{
    [ { 1,{0,1,0} }, { 2,{1,2,1} }, { 3, { 2,1,0} } ]
    [ { 10, [ 1,2,3 ] } ]
    [ 1,2,3,4,5,6,7,8 ]
}
\end{data}
\end{record}
```

The execution and the response of the example of commands are shown below:

```
 1: >>> from UDFManager import *
 2: >>> ii=UDFManager("testdata/sample/sample.udf")
 3: >>> print(ii.totalRecord())
 4: 1
 5: >>> print(ii.currentRecord())
 6: -1
 7: >>> ii.jump(0)
 8: 0
 9: >>> print(ii.get("parameter.vertex[].position.x",[1]))
10: 1.0
11: >>> print(ii.get("parameter.vertex[2].position.x"))
12: 2.0
13: >>> ii.put( 3, "parameter.vertex[2].position.x")
14: 1
15: >>> print(ii.get("parameter.vertex[2].position.x"))
16: 3.0
17: >>> print(ii.size("parameter.vertex[].position"))
18: 3
19: >>> vtx = ii.get("parameter.vertex[2]")
20: >>> print(vtx)
21: [3, [3.0, 1.0, 0.0]]
22: >>> ii.insert( 2, "parameter.vertex[3]")
23: 2
24: >>> vtx[0] = 4
25: >>> ii.put(vtx,"parameter.vertex[]",[3])
26: 2
27: >>> vtx[0] = 5
28: >>> vtx[1][2]=9
29: >>> ii.put(vtx,"parameter.vertex[]",[4])
30: 2
31: >>> print(ii.get("parameter.vertex[].position"))
32: [[0.0, 1.0, 0.0], [1.0, 2.0, 1.0], [3.0, 1.0, 0.0], [3.0, 1.0, 0.0],
    [3.0, 1.0,9.0]]
33: >>> print(ii.get("parameter.vertex[]"))
34: [[1, [0.0, 1.0, 0.0]], [2, [1.0, 2.0, 1.0]], [3, [3.0, 1.0, 0.0]],
    [4, [3.0, 1.0, 0.0]], [5, [3.0, 1.0, 9.0]]]
35: >>> ii.write("testdata/sample/new_sample.udf")
36: 1
37: >>> ii=0 # destruction
```

4.2 Example of Python in GOURMET

The **UDFManager for Python** formed script in the last section can be rewritten in the simplified form which can be executed on the Python scripting window of GOURMET, like this:

```
print(totalRecord())
print(currentRecord())
jump(0)
print($parameter.vertex[1].position.x)
print($parameter.vertex[2].position.x)
$parameter.vertex[2].position.x = 3
print($parameter.vertex[2].position.x)
print(size("parameter.vertex[].position"))
vtx = $parameter.vertex[2]
print(vtx)
insert( 2, "parameter.vertex[3]")
vtx[0] = 4
$parameter.vertex[3] = vtx
vtx[0] = 5
vtx[1][2]=9
$parameter.vertex[4] = vtx
```

```
print($parameter.vertex[].position)
print($parameter.vertex[])
write("testdata/sample/new_sample.udf")
```

Before executing a script, the target UDF file needs to be opened with operations on GOURMET. Here is the result of running the script:

```
1
0
1.0
2.0
3.0
3
[3, [3.0, 1.0, 0.0]]
[[0.0, 1.0, 0.0], [1.0, 2.0, 1.0], [3.0, 1.0, 0.0], [3.0, 1.0, 0.0], [3.0, 1.0, 9.0]]
[[1, [0.0, 1.0, 0.0]], [2, [1.0, 2.0, 1.0]], [3, [3.0, 1.0, 0.0]], [4, [3.0, 1.0, 0.0]],
 [5, [3.0, 1.0, 9.0]]]
```

4.3 Example of drawing functions

How to pass coordinates and drawing attributes as arguments to a drawing function is shown below.

```
line([1,0,0],[1,1,1],[0,1,0,1])
line($parameter.vertex[0].position,$parameter.vertex[1].position,[1,0,0,1])
line($parameter.vertex[0].position,$parameter.vertex[1].position,2)
sphere($parameter.vertex[0].position.x,$parameter.vertex[0].position.y,
       $parameter.vertex[0].position.z,[1,0,0,1,0.5])
```

Example of meshfield : 1. Regular mesh(structured mesh)

```
meshf=meshfield("regular",[[10,20],[0,20],[0,30]],[10,20,30])
nn=1
for i in range(0,11):
    for j in range(0,21):
        nn=nn+1
        for k in range(0,31):
            meshf.set([i,j,k],nn)
cplane([10,0,0],[1,0,0])
cplane([10,0,0],[0,1,0])
cplane([10,0,0],[0,0,1])
cplane([20,20,30],[1,0,0])
cplane([20,20,30],[0,1,0])
cplane([20,20,30],[0,0,1])
ccolor([1,0,0,1,0,0,1,1])
meshf.draw(frame=2,skip=1) # a contour is generated here.
```

Example of meshfield : 2. sphere

```
meshf=meshfield('sphere',[[0,30],[0,3.1415*2]],[4,12])
idxlist=[]
vallist=[]
for i in range(0,5):
    for k in range(0,13):
        idxlist.append([i,k])
        vallist.append(i+k+1)
```

```
meshf.field(idxlist,vallist)
meshf.ccolor([1,0,0,1.0,0,0,1,1.0])
meshf.draw()
```

Example of meshfield : 3. Unstructured mesh

```
mlist=[]
mid=Location('parameter.vertex[0].id')
mpos=Location('parameter.vertex[0].position')
msiz=size('parameter.vertex[]')
for i in range(0,msiz):
    zlist=[get(mid),mpos.str()]
    mlist.append(zlist)
    mid.next()
    mpos.next()
elelist=get('parameter.face[].vertex[]')
mval=get('parameter.value[]')
### MESHFIELD ###
mf=meshfield("triangle",mlist,elelist)
mf.field(None,mval[0:msiz])
mf.ccolor(color=1)
mf.draw()
```