OCTA
ソフトマテリアルのための統合化シミュレータ
GOURMET PYTHONスクリプト
リファレンスマニュアル

OCTAユーザーズグループ
Aug. 08 2007

執筆者

西本正博

プログラム開発者

プログラム開発　西本正博
動作チェック　西尾裕三，西谷栄介

謝辞

本プログラム開発の大半は、経済産業省の出資・補助を受け、新エネルギー・産業技術総合開発機構(NEDO)が(財)化学技術戦略推進機構に委託した、大学連携型産業科学技術研究開発プロジェクト「高機能材料設計プラットフォーム」通称「土井プロジェクト」の下で行われたものである。

本プログラムの改良は、独立行政法人 科学技術振興機構（旧称　特殊法人 科学技術振興事業団）の補助を受け、「多階層的バイオレオシミュレータの研究開発」における「バイオレオシミュレータ用プラットフォーム」機能改良ソフト開発の下で行われたものである。

Copyright ©2000-2007 OCTA Licensing Committee All rights reserved.

目次

第１章 はじめに.. 1
第２章Pythonの基本的な使い方.. 2
2.1 Pythonの起動... 2
2.2 変数.. 2
2.3 タプル、リスト、ディクショナリ 2
2.4 制御構文... 3
2.5 関数とクラス... 4
2.6 モジュール... 4
2.7 参考文献... 5
第３章OCTAにおけるPythonの活用.. 6
3.1 OCTAにおけるPythonの活用とは....................................... 6
3.2 UDF操作用拡張モジュールUDFManager for Python 6
3.3 Python in GOURMETの実行.. 16
3.3.1 GOURMETにおけるUDF Manager for Python の利用法 16
3.3.2 グラフ描画用シート操作関数 17
3.3.3 Python描画関数... 18
3.3.3.1 一般描画関数 19
3.3.3.2 contour関係描画オブジェクト（廃止） 25
3.3.3.3 メッシュフィールド関数 25
3.3.3.4 描画属性 32
3.3.3.5 Cognac描画クラス 34
3.3.3.6 ３次元移動（平行移動・回転移動）アクション用メソッド 39
第４章 ユーティリティモジュール.. 41
4.1 UDFファイル操作用ユーティリティ 41
4.2 構造メッシュおよび非構造メッシュ操作用ユーティリティ 45
4.2.1 ユーティリティモジュール 45
4.2.2 部分領域抽出処理.. 50
4.2.3 NASTRANデータコンバータースクリプト 52
4.2.3.1 NASTRANデータ読み込み 52
4.2.3.2 NASTRANデータ書き込み 52
4.2.3.3 部分領域構成.. 53
4.2.3.4 描画 53
第５章 サンプル.. 55
5.1 UDFManager for Pythonのサンプル 55
5.2 Python in GOURMETのサンプル... 57
5.3 Python描画関数のサンプル.. 57
5.4 色相範囲による描画のサンプル.. 59
5.5 チューブ・リボンの描画のサンプル 62
5.6 UDFデータと描画オブジェクトの関連付けサンプル 65
付録A OCTA2007 リリースにおける新機能・変更点の一覧.................................. 67
A.1 GOURMET 2007 の新機能および変更点...................................... 67
A.1.1 GOURMET2007 の新機能.. 67
A.1.2 その他の変更点 67

A.2 GOURMET 2006 の新機能および変更点... 68
A.2.1 GOURMET2006 の新機能.. 68
A.3 GOURMET2005 の新機能および変更点（履歴） 68
A.3.1 GOURMET2005 の新機能.. 68
A.3.2 GOURMET2005 の変更点.. 70
A.4 GOURMET2003 の新機能および変更点（履歴） 70
A.4.1 GOURMET2003 の新機能.. 70
A.4.2 GOURMET2003 の変更点.. 71
付録B. 解決されたバグ一覧.. 72
B.1 OCTA2006 で解決されたバグ 72
B.2 OCTA2006 で解決されたバグ 72
B.3 OCTA2005 で解決されたバグ 72
B.4 OCTA2003 で解決されたバグ 72

第１章 はじめに

GOURMETはPythonスクリプト言語を用いてＵＤＦに記述されているデータに様々な操作をすることができます。データの値を参照したり編集してファイルにセーブすることはもちろん、データに付随する単位を変更したり、新しいレコードを追加することなどもPythonスクリプトから行うことができます。さらに３次元座標値を指定してGOURMET描画画面に線分・矢印・球や四面体などの単純な図形を描画することや、図形の頂点に与える値によって表示色が変化するコンター図等の描画を行うことができます。
このような操作を行うPythonスクリプトを実行する方法は、以下で述べるように２種類用意されています。

・UDFManger for Python
１つの方法はPythonコマンドウィンドウを普通に開いてからＵＤＦ操作用の拡張ライブラリをインポートして実行することです。この方法はGOURMETのグラフィカルユーザーインターフェースを用いないのでメモリーなどのコンピュータ資源をあまり消費することなく実行できますが、描画に関する操作などGOURMETを操作するコマンドを実行することはできません。
UDFファイルを扱うコマンド群はUDFManagerクラスのメソッドとしてまとめられており、３章第２節で説明します。

・Python in GOURMET
GOURMETのPythonパネルでPythonスクリプトを実行します。
GOURMETのPythonパネルから実行できる命令は、UDFManagerクラスのメソッド、グラフ描画用シート操作関数およびGOURMETで図形を描画するための描画関数があります。
３章第３節でPythonパネルからUDFManagerクラスのメソッドを実行する時の注意点とグラフ描画用シート操作関数及び描画関数について説明します。

まず第２章ではPythonの基本的な使い方を簡単に説明します。
UDFManagerクラス以外のユーティリティモジュールについて第４章で説明します。

第 2 章 Python の基本的な使い方

この章では、Pythonの使い方とスクリプトの書き方について簡単に説明します。更に詳しくPythonについて知りたい方はこの章の最後にあげた参考文献を読んでください。

2.1 Python の起動

Pythonは、Unix、Linux、Win32(Windows9X/NT/2000/XP)などの多くの環境で動作するアプリケーションです。Unix(Linux)マシンでは、pythonとタイプ入力すればPythonインタプリタが起動します。WindowsではDOSプロンプトあるいはスタートメニューからPythonインタプリタを起動することができます。[1]

2.2 変数

Pythonが扱うデータは数値オブジェクト（整数、長整数、浮動小数点、複素数）、文字列オブジェクト、配列オブジェクトやその他のクラスのオブジェクトです。Python変数は変数名とオブジェクトを結びつけます。変数は型宣言なしで使います。
以下に簡単なスクリプト例を示します。

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

文字列はシングルクォート(')、ダブルクォート(") あるいは３重のシングルクォート(''') またはダブルクォート(""") で挟みます。文字列は+(連結) 演算子で結合することができます。数値には、通常の四則演算(+,-,*,/) 、累乗(**)および余り(%)の演算子が使えます。

2.3 タプル、リスト、ディクショナリ

Pythonのオブジェクトには配列のように複数のオブジェクトの集まりを扱えるものが用意されています。タプル(tuple)、リスト(list)およびディクショナリ(dictionary) の3 種類です。タプルは一旦作成すると要素を変更できませんが、リストは変更できます。ディクショナリはハッシュ表であり、キーとして文字列や数値などを与えます。以下に、その使用例と実行結果を示します。

```
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] Pythonが既にインストールされていて実行検索パスにPythonが設定されていると仮定しています。

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

実行結果は以下のようになります。

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

2.4 制御構文

プログラムを作成する上で基本となる制御構文には、条件分岐としてif/else 文、ループ構文としてfor及びwhile文があります。
条件分岐の例として、aとb の大きい方の数をmに代入するプログラムは以下のようになります。

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

ループの例として、100までの素数をリストに入れる例を示します。range()はfor ループ文の中で良く使われる関数であり、range(i,j,k)とすると、iから(j-1)までkずつ増える数のリストを返します。kは省略すると、1とみなされます。またiを省略すると0とみなされます。

```
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
```

・ブロックについて
Python では、ifやforループなどの構文上のブロックをインデント（字下げ）の量によって表現します。各ブロックのインデントの量は任意ですが、１つのブロックのインデントは同じ深さになるようにします。（以下で述べるdef文やclass文においても、定義する文の範囲を示すためにインデントが必要です。）

2.5 関数とクラス

次の例のように、新たな関数はdef文を使って作ることができます。

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

class文を使って次のように新しいクラスを作ることができます。

```
class Linear: #y=a*x+b
  def __init__(self,a,b):
    self.a=a
    self.b=b
  def y(self,x):
    return self.a * x + self.b
  def solv(self,y):
    if self.a == 0.0: return None
    return (y-self.b)/self.a
```

ここで、__init__()はクラスのコンストラクターです。クラスのメソッドはselfを第一引数として指定します。#は以降の一行がコメントであることを示します。
クラスのオブジェクトは以下のように生成します。

```
b=Linear(2.0,3.0)
b.y(4)
b.solv(0)
```

2.6 モジュール

Pythonは、Pythonで書かれたスクリプトファイルやC/C++で書かれたダイナミックリンクライブラリをimport文を使って取り込むことにより、その中のプログラムを実行することができます。Pythonスクリプトファイルおよびpythonに取り込めるダイナミックリンクライブラリを（拡張）モジュールなどと呼びます。

例えば、上記のdef文の例にあるfact(n)関数が、factpy.pyというファイルに保存されている時、

```
import factpy
```

とすると、

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

などと呼び出すことができます。
関数名の重複がない時には、呼び出し関数名の"factpy."部分をなくして関数を呼び出すことができ、

```
form factpy import *
```

とすれば、

```
print fact(10)
```

と利用できるようになります。

拡張モジュールファイルがPythonを実行しているディレクトリにあれば、問題なくPythonはimportできます。もしPythonの実行ディレクトリに拡張ライブラリファイルが無い場合、Pythonインタプリタは拡張ライブラリの所在が分からないためにimport命令がエラーとなります。その時は、環境変数PYTHONPATHに拡張ライブラリのあるディレクトリパスを設定してインタプリタに拡張ライブラリの所在を知らせる必要があります。

2.7 参考文献

Python入門　Mark Lutz著　飯坂剛一監訳　株式会社オライリー・ジャパン
Pythonプログラミング　Mark Lutz著　飯坂剛一監訳　村山敏夫,戸田英子共訳　株式会社オライリー・ジャパン
Pythonデスクトップリファレンス　Mark Lutz著　飯坂剛一監訳　金森玲子訳　株式会社オライリー・ジャパン
Pythonテクニカルリファレンス　言語仕様とライブラリ　D・ビーズリー著、習志野弥治朗訳　ピアソン・エデュケーション
初めてのPython　Mark Lutz,David Ascher共著　紀太章訳　株式会社オライリー・ジャパン
Python Language Home Page
Python の本家ホームページ。Python本体はここからダウンロードできます。
http://www.python.org/

第 3 章 OCTA における Python の活用

OCTAではUDF書式ファイルを操作するということが中心となるのでPythonの拡張機能を利用してUDFファイルを簡単に扱えるようにする機能を提供しています。以下ではOCTA上で利用できるPythonの拡張機能について説明します。

3.1 OCTA における Python の活用とは

UDFデータファイルの主要な部分は定義部とデータ部からなっていて、データ部はほとんど数値の羅列になっています。大きなデータファイルでは、ある数値はどのデータの何番目の配列かということが簡単には分かりません。UDFManager拡張モジュールを使えば簡単に必要なデータにアクセスすることができるようになります。

UDFManager拡張モジュールは通常のPython拡張モジュールと同じようにPythonインタプリタからimport文によって取り込むことができます。Pythonインタプリタが起動していていれば、以下のようにするだけでUDFデータファイルから必要なデータを読み出すことができます。

```
from UDFManager import *
udfobj = UDFManager('UDFファイルのパス文字列')
print udfobj.get('データ名称と配列インデックスを含む文字列')
```

他にもデータ編集やレコード移動等のメソッドが用意されています。

UDFデータを描画するためにはGOURMETシステムが必要です。GOURMETの中でPython拡張モジュールを使えば、GOURMETの描画画面に３次元の高分子モデル等を描画することができます。GOURMETでもUDFManager拡張モジュールと同じデータ操作メソッドを利用することができます。

3.2 UDF 操作用拡張モジュール UDFManager for Python

UDFManagerモジュールは基本的なデータ操作用クラスとして２つのクラスを持ちます。**UDFManager**クラスと**Location**クラスです。UDFManagerクラスはUDFファイルを読み込んでデータを操作します。Locationクラスはデータ名称を簡単に扱えるようにした文字列操作用のクラスです。

（１）UDFManagerクラス

UDFManagerクラスのメソッド解説を以下に述べます。

・**UDFManager(udf_filename)**

コンストラクターです。指定されたＵＤＦファイルを読み込み、インスタンスを生成します。

（例）

```
from UDFManager import *
uobj=UDFManager("/home/octa/test/simple_test_out.udf")
uobj=UDFManager(r"C:¥home¥octa¥test¥simple_test_out.udf")
```

・**totalRecord()**

ＵＤＦファイル内の総レコード数を返します。

・currentRecord()

現在のレコード位置を返します。

・jump(record_no_or_label)

指定されたレコード番号またレコードラベルに移動します。コモンレコードに移動するためには、レコードとして-1を指定します。移動できればそのレコード番号、できなければ-2を返します。

・nextRecord()

次のレコードに移動します。移動できればそのレコード番号、できなければ-2を返します。

・newRecord(label="",record_pos=-999,record_num=1)

指定レコード位置の前に新規レコードを作成し、成功すればレコードラベルを返します。できなければNoneを返します。データ参照および編集対象とする現在レコードは新規レコードに移動します。

挿入レコード数が複数の場合は、２つ目以降のレコードラベルは"'Step ' + レコード番号"が自動的に設定されます。参照および編集対象とする現在レコードは追加レコードの先頭レコードになります。

label：先頭新規レコードのレコードラベル。

record_pos：挿入するレコード位置を指定する。指定したレコードの前に新規レコードが挿入される。負数の場合は、最後に新規レコードを追加する。

record_num：挿入レコード数を指定する。デフォルト値は１。

・eraseRecord(record_pos=-999,record_num=-999)

指定レコード以降のレコードを指定数削除します。現在レコードが削除対象でないときは、現在レコード位置は変化しません。現在レコードが削除される時は、削除後の現在レコードは１つ前のレコードになります。

record_pos：削除開始レコード指定する。負数の場合は、現在レコードが削除開始レコードとなる。

record_num：削除レコード数を指定する。負数の場合は、削除開始レコード以降の全レコードが削除対象となる。

・setRecLabel(label)

カレントレコードのレコードラベルを変更します。

・getRecLabel()

カレントレコードのレコードラベルを返します。コモンレコードの場合は、Noneを返します。

・seek(location)

データ参照位置が指定されたlocation位置に移動します。ここで"location"はデータ名と配列インデックスの一組を意味し、下記のようにインデックスを含むデータ名もしくはデータ名＋インデックスリストで指定することができます。

（例）

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

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

・tell()
現在のデータ参照位置を返します。

・next()
現在のデータ参照位置の最も下位の配列インデックスを１つ進めます。

・prev()
現在のデータ参照位置の最も下位の配列インデックスを１つ戻します。

・type(location)
指定されたlocationのＵＤＦデータ型を文字列で返します。

・size(location)
指定されたlocationが配列であれば、locationのインデックスが空で指定された分の要素数を返します。インデックス指定が空ではない配列データまたはスカラー型データがlocationに指定された時は、データがあるときは１を返しデータが無い時はゼロを返します。locationの指定が間違っているときはNoneを返します。
インデックス指定は右からのみ空で指定することができます。
（例）
uobj.size("field.fraction[0].value[]")
または
uobj.size('field.fraction[].value[]',[0])

・get(location,unit="")
指定されたlocationの値を返します。データの配列インデックスはデータ文字列に含めて指定するかあるいはデータ文字列とインデックスリストに分けて指定できます。locationに構造体データ名を指定した場合は、構造体データに含まれるデータ全てがリストで返されます。
引数unitで単位が指定された時、データ値はその単位に変換されます。ただし、指定できる単位はＵＤＦの定義部で指定した単位と同じ単位次元を持っていなければなりません。
（単位および単位の次元については"ＵＤＦ文法リファレンスマニュアル"を参照。）
unitを指定した場合はlocationに構造体データ名を指定できません。
locationに誤ったデータ名を指定したり、unitに指定した単位で単位変換できない場合などの時はNoneを返します。
（例）
uobj.get("field.fraction[0].value[1]")
または
uobj.get("field.fraction[].value[]",[0,1])
uobj.get("field.fraction[0].value[1]","[m/s]")
または
uobj.get("field.fraction[].value[]",[0,1],"[m/s]")
uobj.get("field.fraction[0].value[]") 等と配列インデックスを省略した場合、すべての配列データがリストにして返されます。配列インデックスは右側からのみ省略できます。

・getArray(location,unit="")

指定されたlocationのＵＤＦ配列データを返します。データが存在しなければ、Noneを返します。locationの配列インデックスが省略されている時は、配列データをリストで返します。

・put(value,location,unit="")

指定されたlocationに値をセットします。unitで単位が指定された時、データ値はその単位で入力されたものとみなされます。ただし、指定できる単位はＵＤＦの定義部で指定した単位と同じ単位次元を持たなければなりません。（単位および単位の次元については、"ＵＤＦ文法リファレンスマニュアル"を参照。）

unitを指定した場合は、locationに構造体データ名を指定できません。構造体及び配列データをセットする時、valueに指定するデータ記述形式は、getメソッドで取り出した時のデータ記述形式と同様にします。

（例）

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

・insert(number,location)

指定されたlocationのインデックス位置に空のデータをnumber個挿入します。空データとは、数値の場合はゼロ、文字列の場合は長さゼロの文字列です。

・erase(number, location)

指定されたlocation位置からnumber個のデータを削除します。削除に成功した時は削除した数を返し、失敗した時はNoneを返します。スカラーデータやスカラー構造体データを削除する時には引数numberにゼロより大きい数値を指定しておきます。

・getDefine(name='')

引数nameに指定されたデータ名の属性データ名をリストで返します。nameを指定しない時は最上位のデータ定義名リストを返します。

（例）

```
print getDefine()
print getDefine('Simulation')
print getDefine('Simulation.Deformations[]')
```

（実行結果）

```
['GraphSheet[]', 'Restart', 'MyComponent', 'MySample', 'Deformation', 'Simulation',
'StepInfo', 'Stress', 'ComponentData', 'StepData', 'Vector3D', 'LinkConfig',
'ChainConfig', 'SystemConfig', 'Unit_Parameter']

['Deformations[]']

['FlowType', 'DeformationType', 'Strain', 'StrainRate', 'dt', 'MaxTimeStep',
'IntervalStep']
```

・queryDefine(name,key='')

ＵＤＦ定義の付加的情報を返します。

name：データ名を与えます。
key ：付加的情報のキーワードを与えます。指定できるキーワードは以下のとおり、
''　　　：データ定義が存在する場合１、無い場合０を返します。
'class'　：class 付の構造体およびその属性データの場合１、その他の場合０を返します。
'help'　：ヘルプ説明文を返します。
'select'：select 型の場合、選択項目のリストを返します。
select 型でない場合、空リスト([])を返します。

keyに与えたキーワードあるいはudfpathに与えたデータ名が存在しない場合はNoneを返します。データ名が存在していれば、ヘルプ文など文字列データの情報がＵＤＦ定義に設定されていない場合でも、空文字('')を返します。
（例）
```
print queryDefine('Simulation_Conditions','help')
print queryDefine('BoundaryCondition','class')
print queryDefine('Interactions.Pair_Interaction[].Potential_Type','select')
```

・isGlobalDef(name)
name に与えたデータが globaldef として定義されているかどうかを返します。
戻り値：globaldef の時=1, NOT=0

・file()
・udfFile()
現在対象としているUDFファイルのパスを返します。
file()は他のモジュールの関数と混同される可能性があるので、udfFile()の使用を推奨します。

・udfFilename()
現在対象としている UDF ファイルのファイル名部分を返します。

・udfDirectory()
現在対象としているUDFファイルがあるディレクトリパスを返します。

・write([filename,record=allRecord,mode=overwrite,define=0])
データをUDFファイルに書き込みます。
record = allRecord(0)/currentRecord(1)/initialRecord(-1)
:全てのレコードか、現在のレコードかを指定します。
mode = overwrite(0)/append(1)：上書きか、追加かを指定します。
（注１）record = allRecordの時、modeの設定は無視されます。
define=0/1 ：定義部分を書き出すかどうかを指定します。
（注２）record = allRecordの時、defineの設定は無視されます。
（注３）define=1の時、modeの設定は無視され上書きのみになります。
recordおよびmode にデフォルト以外の値を使う時はファイル名に現在対象としている以外の別ファイル名を指定する必要があります。
引数の与え方により、以下のような使い方ができます。
（例１）現在のファイルを更新する。
```
uobj.write()
```
（例２）別ファイルに書き出す。

uobj.write("/home/octa/test/simple_test_work.udf")
（例３）別ファイルに現レコードのみ書き出す。（コモンデータを書き出す場合は、カレントレコードを-1にする）
uobj.write("/home/octa/test/simple_test_work.udf",currentRecord)
（例４）別ファイルの末尾に現レコードを追加する。（ファイルが無ければ新規作成する）
uobj.write("/home/octa/test/simple_test_work.udf",currentRecord,append)
（例５）別ファイルにデータ定義および現レコードを書き出します：
uobj.write("/octa/test/work.udf",record=currentRecord,define=1)

・getIDList(class_name)

class_nameクラスの指定に使われたID型データをリストで返します。
（例）
idlist = uobj.getIDList("Vertex")

・getKeyList(class_name)

class name クラスの指定に使われたKEYデータをリストで返します。
（例）
keylist = uobj.getKeyList("AtomType")

・getLocation(class_name,key_or_index)

指定されたクラスで、KEY/ID値を持つlocationを返します。
（例）
loc = uobj.getLocation("AtomType",keylist[0])
返されるlocationはclass_name型のデータ名です。

・ヘッダー情報関連

getEngineType()
getEngineVersion()
getIOType()
getProjectName()
getComment()
getAction()

それぞれのＵＤＦヘッダー情報を文字列で返します。

setEngineType(engine)
setEngineVersion(version)
setIOType(io)
setProjectName(project)
setComment(comment)
setAction(files)

それぞれのＵＤＦヘッダー情報を文字列で設定します。

・単位関連

unitConvert(to_unit,from_unit,value=1.0)

valueの値をfrom_unitからto_unitに変換した値を返します。valueは数値または数値のリストで与えます。
（例）getUnitValue('[erg]','[J]',[1.1,2.1])

ただし、単位[erg]及び[J]はＵＤＦ単位定義部で定義されている必要があります。
例えば、
[J]=[kg*m^2/s^2]
[erg]=[g*cm^2/s^2]

unit(name,set_unit_name='')
ＵＤＦデータに指定された単位を取得または変更します。ＵＤＦデータに単位が指定されている場合はその単位名を返します。set_unit_nameはUDFデータに設定する、あるいは変更する単位名で、ＵＤＦ定義部で指定した単位と同じ単位の次元を持つ単位にのみ変更できます。

unitList(name)
nameで与えられたＵＤＦデータに指定された単位と交換可能な登録済単位のリストを返します。返される単位はＵＤＦデータに指定された単位と単位次元が同じです。
（例）print unitList("Simulation_Conditions.Dynamics_Conditions.Max_Force")
print unitList("Interactions.Pair_Interaction[0].Cutoff")
（結果例）['N', 'dyn', 'sigma*mass/tau^2']
['nm', 'um', 'mm', 'cm', 'm', 'Angstrom', 'sigma']

getUnit(unit_name)
単位定義部に記述された単位定義式を返します。unit_nameには単位名の文字列を[]付きで指定します。（"[sigma]"等）

setUnit(unit_name,unit_expression)
単位定義を新規作成または変更します。
unit_expressionには単位定義式または単位定義式の係数を指定します。
（例）
setUnit('[sigma]','1.2[nm]')
既に[sigma]が、'1.1[nm]'等と定義されていて、係数部の1.1を2.1に変更するだけならば以下のようにします。
setUnit('[sigma]',2.1)

・単位系関連

allUnitSystem()
現在使用可能な単位系を文字列リストで返します。

getUnitSystem()
現在指定されている単位系名を返します。

setUnitSystem(unit_system)
単位系を変更します。

readUnitSystemFile(unit_system_path)
新たな単位系定義ファイルを読み込みます。（単位系定義ファイルについては"ＵＤＦ文法リファレンスマニュアル"参照。）

・データ取得レコードの切り替えおよび取得

common_start()
common_end()
record_start()
record_end()

データ取得系関数 (size(),get(),getArray(),getIDList(),getKeyList(),getLocation()) によって取り出されるデータは、まず現在のレコードのデータを探索してデータが有ればそれを返し、無ければコモンデータ部の値を返します。現在のレコードとコモンデータ部両方にデータがある時にコモンデータ部のデータのみを取り出したい時、common_start()で取り出す対象のレコードをコモン部だけにすることができます。この状態はcommon_end()が呼ばれるまで続きます。
逆に本当にレコード部のみにデータがあるのか知りたい時などには、record_start()を使って対象をレコードのみにすることができます。

getReferRecord()
setReferRecord(rec_ref)

データ参照対象のレコードが、コモン（イニシャル）か現在レコードかを取得(getReferRecord)および設定(setReferRecord)します。
戻り値：コモン（イニシャル）のデータを参照 = 0,
　　　　現在レコードのデータを参照 = 1,
　　　　現在レコードにデータがあれば現在レコードを参照し、無ければコモンを参照=2.
rec_ref:上記戻り値に示した数値を指定します。

・バイナリUDF関連

setBinaryMode()
UDFオブジェクトにバイナリモードを設定します.
テキストUDFから生成されたオブジェクトに対しては、setBinaryMode()を呼び出した以降に追加されたレコードがバイナリになります。

isBinaryMode()
テキストUDFオブジェクトではゼロ、バイナリUDFオブジェクトでは１を返します。

（２）Locationクラス

UDFManagerクラスのメソッドでUDFデータ名を取り扱う時に、データ名の文字列操作が煩雑になる場合があります。Locationクラスを用いればデータ名の文字列操作を簡単に行うことができます。
Locationオブジェクトを引数として渡せるUDFManagerメソッドには以下のものがあります。
　put, get, getArray, insert, erase, seek

Locationクラスのメソッドを以下に示します。

・Location(path_and_index)

コンストラクターです。下記のように使用します。

```
loc = Location('a[].b[]',[0,0])
loc = Location('a[0].b[0]')
```

・str()
配列添え字を含むデータ名文字列を返します。

・root()
構造体の最上位データ名に移ります。

・seek(location_index_args)
指定されたデータ名に移ります。

・up()
１つ上の構造体データ名に移動します。

・down(location_index_args)
下の属性パスに移動します。

・prev()
最も下位の配列インデックスを１つ減らします。

・next()
最も下位の配列インデックスを１つ増やします。

・getPath()
配列インデックスを含まないデータ名を返します。

・getIndex()
配列インデックスをリストで返します。

Locationクラスの使用例
```
>>> 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）関数

UDFManager拡張モジュールに定義されている関数について説明します。

・バイナリコンバータ

convertUDF(infiles, outfile = None, option = 'binary', binaryRecord = 0, force = 0)

テキストUDFファイルとバイナリUDFファイルの相互変換を行います。

infiles：コンバート対象の入力ファイル名または入力ファイル名のリスト

outfile：出力先ファイル名

出力先ファイル名が指定された時、コンバート対象ファイルは１つになります。出力先ファイル名が指定されていない時、出力先ファイル名として入力ファイル名の拡張子部分を変更したファイル名を使います。バイナリUDFファイル名の拡張子は".bdf"、テキストUDFファイルの拡張子は".udf"となります。

option：'binary'('b')または'text'('t')

それぞれtext->binaryまたはbinary->textの変換方向を指定します。

binaryRecord：option='binary'の時、バイナリ化を開始するレコード番号（コモン=-1）を指定します。

force：0（デフォルト）の時、出力先ファイルが存在すれば変換を行いません。1の時は出力先ファイルが存在しても上書きします。

3.3 Python in GOURMET の実行

GOURMETのPythonパネルから上記UDFManagerクラスを使用する時の注意点およびグラフ描画用シート操作関数について説明します。

3.3.1 GOURMET における UDF Manager for Python の利用法

GOURMETのPythonパネルからUDFManagerのメソッドを実行する時には拡張モジュールをインポートするコマンドが自動的に実行されます。従ってインポートコマンドを実行する必要はありません。また、GOURMETでオープンされているＵＤＦファイルに対しては、_udf_という名称のUDFManagerオブジェクトが自動的に生成されます。さらにGOURMETでオープン済みのＵＤＦファイルに対してUDFManagerメソッドを実行する時には、UDFManagerオブジェクトを指定せずに関数を呼び出すような形式でコマンドを発行すれば_udf_オブジェクトに対するメソッドに変換されます。
GOURMETのUDFManagerには他に以下の拡張機能と変更点があります。

（１）＄拡張機能
UDFManagerのget(),getArray()及びput()メソッドについて、データ名を"location"引数で渡す代わりに、UDFデータ名の先頭に$を付けることによってメソッドを代用することができます。
例えば、
```
data = getArray('a[].b',[1])
```
とする代わりに以下のように記述します。
```
data = $a[1].b
```

ただし、$拡張されたUDF変数はPythonインタプリタが管理する変数ではないので、配列の[]部分にPythonのリストで通常使える便利なスライス演算子を使用することはできません。
size()関数では、size("a[].b") 或いはsize($a[].b) の両方の記述ができます。
後述の描画関数では座標を与える引数に$付きのUDFデータ変数を指定することができます。

（２）メソッド名の変更個所
Pythonパネルからオープン済みのUDFファイルに対してUDFManagerのメソッドを実行する場合、オブジェクトを指定せずに関数形式でメッドを記述できます。その時、関数名がPythonインタプリタ関数と重複してしまう場合が有るので、以下の場合のみ関数名を変えて使用します。
type()の代わりに、udftype()を使用します。

（３）GOURMETのみで有効な関数
シート操作関数及び描画関数はGOURMETのPythonパネルからのみ実行することができます。その他にGOURMET上でしか使用できない関数は以下のとおりです。

focusData(datapath)
datapathに指定したデータをGOURMETのEditor画面に表示します。

openUDF(path)
GOURMET に UDF ファイルを読み込みます。
GOURMET の File メニューから UDF ファイルを開く操作を行うことと同様の動作を Python スクリプトから行います。
（例） openUDF(r'D:¥OCTA2002¥PF_ENGINE_2002¥COGNAC3¥blend¥out¥blend_out.udf')

（4）複数の異なるUDFファイルを$$で指定する（$$拡張）機能

GOURMETでオープンされているＵＤＦファイルに対するUDFManagerオブジェクト（_udf_）に対しては、上記$拡張機能を使ってget()及びput()メソッドを簡単に利用することができます。
一方ユーザーがUDFManagerクラスから作成したUDFManagerオブジェクトに対しては、$$を使って以下のようにget()及びput()メソッドを簡単に利用することができます。

```
udf1 = UDFManager(r"D:¥OCTA2005¥blend¥blend_de01_in.udf")
print $$udf1.Molecular_Attributes.Atom_Type[0].Name
$$udf1.Molecular_Attributes.Atom_Type[0].Name = 'atom_C'
```

3.3.2 グラフ描画用シート操作関数

GOURMETには表形式データをgnuplotで簡易にグラフ表示する機能があります。グラフ描画用シートはＵＤＦに定義されたデータ構造以外に、ユーザーが自由に操作することができる表形式データ構造を提供します。
GOURMETでUDFファイルをオープンすると、グラフ描画用シート"GraphSheet[]"がツリーの先頭に表示されます。このグラフ描画用シートに以下に示すグラフ描画用シート操作関数を使ってデータ定義を追加したり、データ値をセットすることができます。

・シートの列を定義します
createSheetCol(col,name,size=0,type='float')
- col ：列番号（0,1,2,...）(既にある列を指定すると上書きします。現在の列数より多い数字が指定された時は最後列に追加します。)
- name：列名称（データ名称）
- size：行数（ゼロでも可。後で行を追加することができます。）
- type：列データタイプ（数値の場合はデフォルトの'float'で良い。文字列の場合のみ'string'とする。）

・シートの列の削除
deleteSheetCol(col_or_name)
- col_or_name：列名称か列番号

・行の挿入
insertSheetRow(index,number)
- index：行番号（ゼロから）。この行番号の直前に全列に対して挿入されます。最後尾に追加する時は現在の行数を指定します。
- number：挿入する行数。

・行の削除
deleteSheetRow(index,number)
- index：行番号（ゼロから）。この行番号からnumber行削除されます。

・列単位でシートにデータをセット

setSheetCol(col_or_name,location_index_or_data_list)
　col_or_name：列名称か列番号
　location_index_or_data_list：ＵＤＦデータ名またはデータのリスト列。

・セル単位でシートにデータをセット
setSheetData(col_or_name,row,location_index_or_data)
　col_or_name：列名称か列番号
　row：行番号（0,1,2,...）
　location_index_or_data：ＵＤＦデータ名または、[ ]で囲ったデータ値。（データ名称と区別するために１つのデータでも[ ]で囲みます。）

・列数および行数の取得
getSheetColSize()
getSheetRowSize()

・値取り出し
value_list = getSheetCol(col_or_name)
　１列のデータをリストで返します。

value = getSheetData(col_or_name,row)
　値を１つ取り出します。

・シート操作関数の使用例
```
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 Python 描画関数

GOURMETのPython入力エリアに記述できる描画関数について説明します。
以下で説明する描画関数の引数について、coordinateN（N は整数）は３次元座標[x, y, z]値を表すPythonリスト、または（x, y, z）の３成分をデータとしてもつUDF構造体変数名の先頭に$を付けたlocationを示します。
coordinate_listはPythonリストとしての３次元座標[x, y, z]形式データを示します。

3.3.3.1 一般描画関数

基本的な描画対象の描画を行う関数を以下に示します。引数にattribute_id あるいは[r,g,b,t]等とリストで与えられているのは描画対象の色や大きさなどを指定するための描画属性値です。描画属性については、3.3.3.4節で説明します。

●線分
line(coordinate1,coordinate2,[r,g,b,t])
line(coordinate1,coordinate2,attribute_id)
　coordinate1,coordinate2：[x1,y1,z1],[x2,y2,z2]とリストに包んで線分の両端座標を指定します。
　r,g,b,t：色を指定します。r,g,b,t は、それぞれ赤(Red),緑(Green),青(Blue),不透明度(Alpha)の値を0.0～1.0の範囲で指定します。

●点
point(coordinate,[r,g,b,t])
point(coordinate,[r,g,b,t,size])
point(coordinate, attribute_id)
　coordinate：[x,y,z]とリストに包んで点の座標を指定します。
　r,g,b,t：色を指定します。
　size：点のドット数を指定します。この指定がない時はデフォルト値３を使用します。

●ポリゴン（多面体）
polygon(coordinate list,[r,g,b,t])
polygon(coordinate list, attribute_id)
　coordinate list：[[x1,y1,z1],[x2,y2,z2],...]のように座標のリストを指定します。
　r,g,b,t：色を指定します。

●ポリライン
polyline(coordinate list,[r,g,b,t])
polyline(coordinate list, attribute_id)
　coordinate list：[[x1,y1,z1],[x2,y2,z2],...]のように座標のリストを指定します。
　r,g,b,t：色を指定します。

●円
disk(coordinate1,[r,g,b,t,radius,vx,vy,vz])
disk(coordinate1, attribute_id)
　coordinate1：円の中心座標を指定します。
　r,g,b,t：色を指定します。
　radius：円の半径を指定します。
　vx,vy,vz：面の法線ベクトルを指定します。

●楕円：中点による指定
ellipse1(coordinate1,[r,g,b,t,a,b,ax,ay,az,vx,vy,vz])
ellipse1(coordinate1, attribute_id)
　coordinate1：楕円の中心座標を指定します。
　r,g,b,t：色を指定します。

a,b：楕円の長軸および短軸の長さを指定します。
ax,ay,az：長軸方向のベクトルを指定します。
vx,vy,vz：面の方向を指定します。
(vx,vy,vz)のうち(ax,ay,az)に垂直な成分を面の法線ベクトルとします。

●楕円：焦点による指定
ellipse2(coordinate1,coordinate2,[r,g,b,t,a,vx,vy,vz])
ellipse2(coordinate1,coordinate2, attribute_id)
coordinate1,coordinate2：楕円の焦点 F1,F2 の座標を指定します。
r,g,b,t：色を指定します。
a：焦点F1,F2からの距離の和が2aとなる楕円を描きます。
vx,vy,vz：面の方向を指定します。焦点F1,F2を通る直線と垂直な成分を法線ベクトルとします。

●円錐
cone(coordinate1,coordinate2,[r,g,b,a,radius])
coordinate1を底円の中心とし、coordinate2を頂点とする円錐を描きます。
radius：円の半径を指定します。

●円柱
cylinder(coordinate1,coordinate2,[r,g,b,t,radius])
cylinder(coordinate1,coordinate2, attribute_id)
coordinate1,coordinate2：円柱の両端円の中心座標を指定します。
r,g,b,t：色を指定します。
radius：円柱の半径を指定します。

●球
sphere(coordinate1,[r,g,b,t,radius])
sphere(coordinate1, attribute_id)
coordinate1：球の中心点座標を指定します。
r,g,b,t：色を指定します。
radius：球の半径を指定します。

●楕円体：中点による指定。
ellipsoid1(coordinate1,[r,g,b,t,a,b,c,ax,ay,az,cx,cy,cz])
ellipsoid1(coordinate1, attribute_id)
coordinate1：楕円体の中心点座標を指定します。
r,g,b,t：色を指定します。
a,b,c：主軸の長さ（円でいえば直径）を、2a、2b、2c とします。
ax,ay,az：主軸 a の方向を指定するベクトルです。
cx,cy,cz：中心点を通り、ベクトル（ax,ay,az）（cx,cy,cz）に垂直なベクトルを主軸 b の方向ベクトルとします。
主軸cの方向ベクトルは主軸aおよびbに垂直なベクトルです。
（注）ベクトル（ax,ay,az）（cx,cy,cz）の与え方により、楕円体が描画出来ない場合があります。

●回転楕円体：焦点による指定
ellipsoid2(coordinate1,coordinate2,[r,g,b,t,a])

ellipsoid2(coordinate1,coordinate2,attribute_id)
　coordinate1,coordinate2：楕円の焦点座標 F1,F2 を指定します。
　r,g,b,t：色を指定します。
　a：楕円体上の点は焦点F1,F2からの距離の和が2aとなる点の集合です。
　（注）定数aの与え方により、回転楕円体が描画出来ない場合があります。

●四面体
tetra(coordinate1,coordinate2,coordinate3,coordinate4,[r,g,b,t])
tetra(coordinate1,coordinate2,coordinate3,coordinate4,attribute_id)

●立方体
cube(coordinate1,length,[r,g,b,t])
cube(coordinate1,length,attribute_id)
　coordinate1は中心座標、lengthは１辺の長さです。

●矢印（矢はcoordinate2 の方に付きます）
arrow(coordinate1,coordinate2,[r,g,b,t,radius,height])
arrow(coordinate1,coordinate2,[r,g,b,t,radius,height,magnification])
arrow(coordinate1,coordinate2,attribute_id)
　coordinate1からcoordinate2に向かう矢印を描きます。
　radius,height：矢じりの半径および高さを指定します。
　magnification：長さ倍率を指定します。指定が無い場合は1.0です。

●テキスト
text(coordinate,content,[r,g,b,t,size])
text(coordinate,content,attribute_id)
　常に画面に対して正面を向くテキストを描画します。
　coordinate：テキストの左下座標を指定します。
　content：テキスト文字列を指定します
　r,g,b,t：色を指定します
　size：テキストのドット数を指定します

●チューブ・リボン
tube(vertexlist,radius=1.0,smooth="fit",arrow=None,hidden=[0,0])
　チューブ（円断面）を描画します。
　vertexlist=[[[x,y,z],[r,g,b,a]], ...]:頂点データのリスト（タプル）です。
　　１頂点データを[[x,y,z],[r,g,b,a]]として、頂点数分リスト(タプル)にします。
　　[r,g,b,a]を省略すると直前に指定した色を指定したことになります。
　radius:断面の半径を指定します。
　smooth:フィッティング方法を指定します。"curve"または"fit"のいずれかを指定できます。
　　"curve"：頂点を通らない３次スプライン曲線を描く
　　"fit"：頂点を通る３次スプライン補間曲線を描く
　arrow=(lengthRatio,widthRatio):終端の矢印の大きさを指定します。
　　lengthRatio：最後の区間に対する矢印の割合（ゼロ～１）
　　widthRatio：半径に対する矢じり幅（１～２．５）
　hidden:異なるチューブ・リボンオブジェクトを滑らかにつなぐために、隠し頂点を両端に設定することができます。始端および終端の隠し頂点の数をリストまたはタプルで与えます。

チューブ・リボンオブジェクトを滑らかにつなぐためには、始端および終端にそれぞれ隠し頂点が１つ必要です。設定できる最大数は、それぞれ２５５です。また、つなぐチューブ・リボンオブジェクトのフィッティング方法は全て同じにしなければなりません。

ribbon(vertexlist,section=("rectangle",width,height),smooth="fit",
arrow=None,hidden=[0,0])

リボン（楕円または矩形断面）を描画します。

vertexlist=[[[x,y,z],[nx,ny,nz],[r,g,b,a]], ...]:頂点データのリスト(タプル)です。

１頂点データを[[x,y,z],[nx,ny,nz],[r,g,b,a]]として、頂点数分リスト（タプル）にします。

[nx,ny,nz]または[r,g,b,a]を省略すると直前に指定した法線ベクトルまたは色を指定したことになります。

section=(sectioType,width,height):リボン断面の情報を指定します。

sectionType:断面形状を指定します

"round"：楕円断面

"rectangle"：矩形断面

width,height:断面の幅および高さを指定します。（頂点に指定する法線ベクトル方向を高さとします）

smooth:フィッティング方法を指定します。"curve"または"fit"のいずれかを指定できます。

"curve"：頂点を通らない３次スプライン曲線を描く

"fit"：頂点を通る３次スプライン補間曲線を描く

arrow = (lengthRatio, widthRatio):終端の矢印の大きさを指定します。

lengthRatio：最後の区間に対する矢印の割合（ゼロ～１）

widthRatio：半径に対する矢じり幅（１～２．５）

hidden:異なるチューブ・リボンオブジェクトを滑らかにつなぐために、隠し頂点を両端に設定することができます。始端および終端の隠し頂点の数をリストまたはタプルで与えます。（チューブ・リボンの描画のサンプル参照）

●イメージファイル(jpeg)

imageRectangle(imageFile,ratio=1.0,origin=[0,0,0],
widthVector=[1,0,0],heightVector=[0,1,0])

イメージファイルを元の縦横サイズで矩形に表示します。

imageFile：イメージファイル名（JPEG）

ratio：1.0の場合、１ドットあたり長さ１で表示します。

origin：イメージを表示する左下座標です。

widthVector：横方向のベクトルを与えます。

heightVector：縦方向のベクトルを与えます。

imageBackground(imageFile,spread=1,ratio=1.0)

ビューのバックグラウンドにイメージを表示します。

imageFile：イメージファイル名（JPEG）

spread:イメージを貼り付ける方法を指定します。

MAX_ALL(1):イメージ全体をビュー内で最大表示します。

MAX_CUT(2):イメージ（の一部）をカットしてビュー内で最大表示します。

RATIO(3):MAX_ALLと同じですが、表示する倍率(ratio)を受け付けます。

ratioは、ゼロ～１．０の範囲です。

STRETCH(4):イメージ全体をビューに合わせて縦横に伸縮して表示します。

●描画操作

clearDraw()

このメソッドが呼ばれる以前の描画を消去します。

appendDraw()

１回のPython実行内で、このメソッド以降の一連の描画を上書き描画します。appendDraw()の前に何らかの描画関数が呼ばれた時は、その時点で描画が消去されます。

clearDrawMode()

一連の描画操作毎に描画を消去するモードになります。（デフォルト）

appendDrawMode()

常に上書き描画するモードにします。

●任意の描画対象とUDFデータを関連付ける機能（ピッキング対象の拡大）

setDrawRelation(udfpath_with_index)
resetDrawRelation()

ＵＤＦデータと描画オブジェクトを関連付けて、ピッキング時にそのデータおよび親データに対するアクションが実行できるようにします。（メッシュフィールドによる描画およびイメージファイルの描画はピッキング対象になりません。）

udfpath_with_index：描画対象と関連付けるＵＤＦデータ名。配列データの場合は必ずインデックスを付けたデータ名を渡す必要があります。

setDrawRelation ～ resetDrawRelation の間に呼ばれた全ての描画対象に同じデータが関連付けられます。

●照明操作

照明光の方向

lightDirection(direction)

ビュー(View)に後述の視点操作を行っていない標準座標(Standard)の状態で照明光の方向をベクトルで指定します。照明光の方向は視点が移動してもビューから見た方向は変わりません。
照明方向の指定をしていない時は、常に照明光の方向は正面から光があたっている状態になります。正面から光があたっている状態は、照明方向の指定をしていれば下記のように指定したことになります。
lightDirection([0,0,-1])

左上から照明があたっているようにするには、
lightDirection([1,-1,-1])
を指定します。

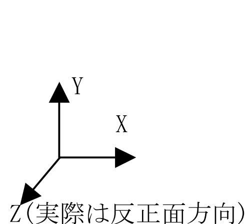


図 3.3.3.1
視点変更のない標準状態座標

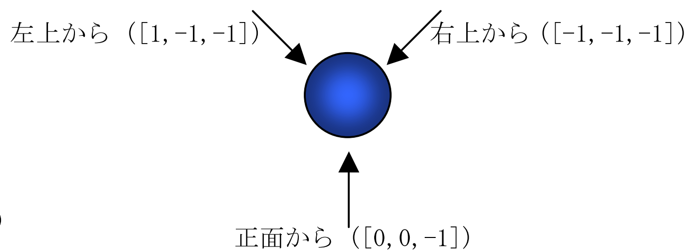


図 3.3.3.2
照明の方向指定例

照明の各種強度

lightParameter(ambient=-1,diffuse=-1,specular=-1)

環境光(ambient)、拡散光(diffuse)、反射光(specular)の強さをそれぞれ指定します。
引数には 0.0 ～ 1.0 の実数を指定します。この範囲外の値を指定した場合は無視されます。
下記のように指定した場合、拡散光および反射光はデフォルト値(=-1)が使用されるので無視され、環境光のみが 0.5 になります。
lightParameter(ambient=0.5)

●視点操作
viewDefaultDirection(eyes,head)

標準(Standard)状態[2]の視点を変更します。eyesには視線変更後に正面となる３次元ベクトルをリストで与えます。headには視線変更後に上方となる３次元ベクトルをリストで与えます。ベクトルを３次元未満で与えた時には、自動的にゼロを追加して３次元のベクトルになります。
例えば、viewDefaultDirection([0,-1,0],[0,0,-1]) を実行すると座標の表示は下図のようになります。

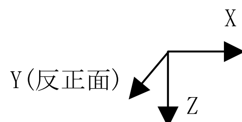
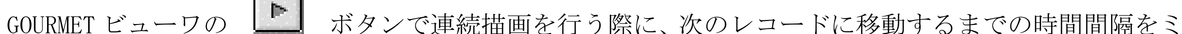


図 3.3.3.3　視点変更後の座標

viewDirection(eyes,head)

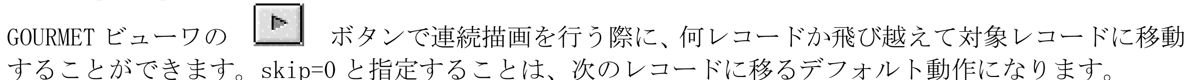
一時的な視点の変更を行います。″一時的″とは viewDirection()が使用されている Python スクリプトを実行した直後のみ視点が変更された状態になるということです。
eyes および head は viewDefaultDirection()と同様に与えます。

●アニメーション（連続描画）操作
drawInterval(millisecond)
GOURMET ビューワの  ボタンで連続描画を行う際に、次のレコードに移動するまでの時間間隔をミ

[2] GOURMETでは[Python: Run]ボタンにより描画スクリプトを実行するとビューの視点は標準(Standard)状態になります。

リ秒単位で与えます。
drawInterval(500)とすると、次のレコードに移動するまで 0.5 秒待ちます。

drawSkip(skip)


GOURMET ビューワの ボタンで連続描画を行う際に、何レコードか飛び越えて対象レコードに移動することができます。skip=0 と指定することは、次のレコードに移るデフォルト動作になります。

UDF ファイルを読み込んでいない状態での Python 実行

GIURMET で UDF ファイルを開かずに Python スクリプトが実行できます。UDF ファイルが開かれていないので、UDF ファイルのデータを操作するメソッドは正常に機能しませんが、描画関連の関数などは実行することができます。
GOURMET で UDF ファイルを開かなくても実行できる UDFManager クラスのメソッドには以下のものがあります。

・openUDF
・全ての一般描画関数
・描画操作（clearDraw()、appendDraw()など）
・contour 作成関数（contour）
・meshfield 関数
・照明操作関数
・視点操作関数
・グラフ描画用シート操作関数

3.3.3.2 contour 関係描画オブジェクト（廃止）

メッシュの要素(triangle, quad,tetra,hexa) を１つづつ設定していく関数contour()は、メッシュ描画の効率がよくありませんでした。そのためcontour()関数はGOURMET3.7.0で廃止しました。同様の機能を持ち、より効率的にメッシュ描画を行えるメッシュフィールド関数をお使いください。

3.3.3.3 メッシュフィールド関数

メッシュフィールド関数は一度にメッシュの要素(triangle,quad,tetra,hexa) を設定することができ、コンター、等値面およびメッシュ要素を効率的に描画します。

（１）meshfield関数によるメッシュオブジェクト生成

構造格子(STRUCTURED MESH) と非構造格子(UNSTRUCTURED MESH) でメッシュの指定方法が異なります。

（１－１）構造格子(STRUCTURED MESH)の場合

```
mesh_obj_in_python = meshfield( mesh_type, coord_list_list, div_list )
```

mesh_type：メッシュのタイプを指定します（最初の４文字が有効です）
"regular","rectangular","sphere"（２次元極座標）,"cylinder"

coord_list_list：下記の形式でメッシュの座標データを与えます。
"regular"の場合、座標値の最大最小をリストで設定します。
[[xmin, xmax],[ymin, ymax],[zmin, zmax]]
"sphere"の場合、（ｒ、θ）の最小値と最大値をそれぞれ与えます。
[[rmin, rmax],[thetamin, thetamax]]
"cylinder"の場合、（ｒ、θ、ｚ）の最小値と最大値をそれぞれ与えます。
[[rmin, rmax],[thetamin, thetamax],[zmin, zmax]]
"rectangular"の場合、分割点の座標をリストで与えます。
[[xmeshPosition0, xmeshPosition1, ...],[ymeshPosition0,...], ...]

div_list：分割区分数をリストで設定します。
"regular","rectangular"の場合
[x_ndiv, y_ndiv, z_ndiv]
"sphere"の場合
[r_ndiv, theta_ndiv]
"cylinder"の場合
[r_ndiv, theta_ndiv, z_ndiv]

（１－２）非構造格子(UNSTRUCTURED MESH)の場合

mesh_obj = meshfield(element_type, vertex_coord_list, element_vertex_list)

element_type：要素タイプを指定します（最初の４文字が有効です）
三角形："triangle"
四角形："quad"
四面体："tetra"
六面体："hexa"
２次元三角形要素・四角形要素混在："2d"
３次元四面体要素・六面体要素混在："3d"
要素形状が混在している時は、要素の節点数から要素形状を決定します。

vertex_coord_list：節点(Vertex)のIDと節点座標のリストを渡す。
座標として、座標値の[x,y,z]リストまたは座標値の構造体データを持つ"location"を指定することができます。
[[1,[x1,y1,z1]],[2,[x2,y2,z2]],[3,[x3,y3,z3]],...,[N,[xN,yN,zN]]]

次のようにx,y,z 座標を下位属性に持つ構造体データ名（ここでは例えばposition）を記述するだけで、構造体内のx,y,z 座標データを指定できます。
[[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]]
IDは整数で並びは自由。IDの重複はエラーになります。

element_vertex_list： 要素を構成する節点の連結関係を示すために、対応する節点の節点ID をリストで渡します。
"triangle"の場合
[[1,2,3],[5,6,7],[9,10,11],.....,[2,3,N]]
"quad","tetra"の場合
[[1,2,3,4],[5,6,7,8],[9,10,11,12],.....,[1,2,3,N]]

（２）meshオブジェクトに対して各ノードに値をセットする

（２－１）setメソッドにより各ノードにスカラー値またはベクトル値をセットします

```
mesh_obj_in_python.set( index_list, data_value)
```

index_list:節点のインデックスのリストです。二次元の場合は、[Ix, Iy]、３次元の場合は[Ix,Iy,Iz]の形式です。
data_value:節点にセットするスカラー値またはベクトル値です。

（２－１－１）構造格子(STRUCTURED MESH)の場合

格子の配列番号に対して値をセットします。（格子の配列番号はゼロから始まるＸＹＺ方向の分割点を示すインデックスです)

・スカラー値のセット
```
meshf.set([0,0,0],1.2)
```
・ベクトル値のセット
```
meshf.set([0,0,0],[1.2,3.5,0.8])
```

（２－１－２）非構造格子(UNSTRUCTURED MESH)の場合

節点ID に対して値をセットします。
・スカラー値のセット
```
meshf.set(1,2.2)
```
・ベクトル値のセット
```
meshf.set(1,[1.2,3.5,0.8])
```

（２－２）fieldメソッドにより各節点のインデックスのリストと値のリスト(スカラー値またはベクトル値)をセットします

```
field_obj = mesh_obj.field( index_list_list, data_value_list )
```

index_list_list:構造格子(STRUCTURED MESH)の場合、ゼロから始まるＸＹＺ方向の分割点を示すインデックスを与えます。
[[0,0,1],[0,0,2],[0,0,3],...,[0,0,Nz-1],...,[Nx-1,Ny-1,Nz-1]]

非構造格子(UNSTRUCTURED MESH)の場合、節点IDを与えます。
[1,2,3,4,...,Nx]

data_value_list:設定する値をリストで与えます。
スカラー値の場合[1.2, 2.0, ..., 5.0]
ベクトル値の場合[[1.2,3.5,0.8],[1.0,3.0,0.5],....,[2.2,1.5,1.8]]
"location"によって値を指定する場合は次のようになります。
ベクトル値としてx,y,zを下位データに持つ構造体名を指定すれば、構造体内のx,y,z成分値を設定できます。
[field.volume_fraction.value[0], field.volume_fraction.value[1], field.volume_fraction.value[2],.....,field.volume_fraction.value[NxNyNz-1]]
（注）非構造格子でindex_list_listを省略した場合（index_list_list=None）、節点IDの昇順に対応した順序でdata_value_listが設定されます。

（２－３）等値面をコンター表示するためのスカラー値をセットする

mesh_obj_in_python.setSubValue(index_list, value)

setメソッドやfieldメソッドで指定されたスカラーから生成される等値面を他のスカラー値で色分け表示することができます。色分け表示するためのスカラー値を各節点にセットします。

index_list:節点のインデックスのリストです。[Ix,Iy,Iz]の形式です。
value:節点にセットするスカラー値です。

```
meshf.setSubValue([0,0,0],1.2)
```

（３）meshオブジェクトの各ノード値の取得

各ノードのスカラー値はgetメソッド、ベクトル値はgetvメソッドにより取得することができます。

（３－１）構造格子(STRUCTURED MESH)の場合

・スカラー場
mesh_obj.get([i,j,k])

・ベクトル場
mesh_obj.getv([i,j,k])
getv() は戻り値としてベクトル値をPythonリスト形式で返します。
[i,j,k] はゼロから始まるＸＹＺ方向の分割点を示すインデックスです。

（３－２）非構造格子(UNSTRUCTURED MESH)の場合

節点ID を指定します。

・スカラー場
　mesh_obj.get(id)

・ベクトル場
　mesh_obj.getv(id)
　　getv()は戻り値のベクトル値をPythonリスト形式で返します。

（３－３）等値面をコンター表示するために設定したスカラー値を返します

mesh_obj_in_python.getSubValue(index_list)

setSubValue(...)でセットした値を取り出します。

（4）メッシュフィールドの描画メソッド

メッシュフィールドクラスを使ってコンターや等値面を描画するために利用するメソッドについて以下に説明します。

・等値面の指定（注[3]）
mesh_obj.isovalue(value, attribute_id)
　等値面を生成するしきい値および等値面の色を指定します。

　等値面の色を指定する方法は以下のとおりです：
　[1] 単一色を与える。（attribute_id に等値面色番号または[R,G,B,A]色リストを与えます）
　[2] 最小値色および最大値色を与えて、２色でグラデーション表示する。
　[3] 最小値色に対応するRGBA値および最大値までの色相範囲を与える。

　[2]および[3]のように、等値面をさらにコンター表示するためには、上記（２－３）で説明した「等値面をコンター表示するためのスカラー値」を設定しておく必要があります。

　等値面をコンター表示する色の指定方法は以下のとおりです：
　[2] 最小値色および最大値色の指定方法
　　次のように、最小値に対応する色のRGBAおよび最大値に対応する色のRGBAをリストで指定します。
　　[minR,minG,minB,minA, maxR,maxG,maxB,maxA]
　[3] 最小値色に対応するRGBA値および最大値までの色相範囲で指定
　　次のように、最小値に対応する色のRGBAおよびHSVカラーモデルにおける色相（HUE）に対応

[3] 等値面描画時の注意
　等値面を生成する処理において、等値面を生成すべきメッシュ頂点間の値の差が、全頂点の値の差（最大値－最小値）の５０万分の１程度になると面を正確に生成できなくなります。
　例えば、全頂点の値の最小値および最大値が、それぞれゼロおよび１である時、等値面を生成するメッシュ付近にある頂点の値の差が 0.000002 程度であると面を正確に生成できなくなります。それを避けるために､極端に大きい値や小さな値をメッシュ頂点に割り当てないようにするなどの前処理が必要になることがあります。

する値をリストで指定します。
[minR,minG,minB,minA, hue_angle]
minR,minG,minBおよびminAの範囲は、0～1.0です。hue_angleの範囲は、-360～360です。
色相範囲指定による描画方法のサンプルは第4章にあります。

旧バージョンでの同様のメソッドも利用できます。
mesh_obj.clevel(min_value,max_value,attribute_id)
min_valueとmax_valueの間にある値を持つ点をつないだ等値曲面を描きます。

・等値面レベルの設定解除
mesh_obj.delete_isovalue()
以前に呼ばれたisovalue()関数の指定を全てキャンセルします。

旧バージョンでの同様のメソッドも利用できます。
mesh_obj.delete_clevel()

・最大値、最小値の指定
mesh_obj.crange(min_value,max_value)
格子点上の値がmin_value以下の値はmin_valueに、max_value以上の値はmax_valueにします。
min_value < max_valueの場合は無視されます。
（複数設定されている場合最後の指定が有効となります。）

・断面の指定
mesh_obj.cplane(point_on_plane,normal_vector,[minR,minG,minB,minA,maxR,maxG,maxB,maxA])
mesh_obj.cplane(point_on_plane,normal_vector,[minR,minG,minB,minA,hue_angle])
mesh_obj.cplane(point_on_plane,normal_vector,[attribute_id])
mesh_obj.cplane(point_on_plane,normal_vector)
meshfieldで指定した３次元要素の等高断面図を描くための平面を指定します。
複数の断面に一度に表示色を指定する時は、本メソッドでは色指定部分を省略して、後述の
ccolor(...)またはdcolor(...)メソッドを利用することができます。
不連続色コンターを指定するためには、dcolor(valueList, colorList)を使用します。

・断面の設定解除
mesh_obj.delete_cplane()
以前に呼ばれたcplane()関数の指定を全てキャンセルします。

・連続色コンターの表示色（最小値・最大値に対する表示色）指定
mesh_obj.ccolor([minR,minG,minB,minA,maxR,maxG,maxB,maxA])
mesh_obj.ccolor([minR,minG,minB,minA,hue_angle])
mesh_obj.ccolor(attribute_id)
meshfieldで指定した２次元要素または３次元要素の等高図を描くための最小値および最大値
表示色を指定します。
（複数設定されている場合最後の指定が有効となります。）

連続色の指定方法には以下の方法があります。
[1] 最小値色および最大値色を与えて、２色でグラデーション表示する。

［2］最小値色に対応するRGBA値および最大値までの色相範囲を与える。

・不連続色コンターの表示色指定
mesh_obj.dcolor(valueList, colorList)
２次元要素または３次元要素の不連続色コンターの色情報を以下のように指定します。
valueList:不連続値のリストです。[value1,value2,...]の形式です。
colorList:不連続値に対応する色のリストです。[value1_color,value2_color,...]の形式です。
（例１）dcolor([0,1,2],[1,2,3]) # 属性ＩＤで指定
（例２）dcolor([0,1,2],[[1,0,0,1],[0,1,0,1],[0,0,1,1]]) # RGB で指定
（複数設定されている場合最後の指定が有効となります。）

（５）描画条件の設定および描画オブジェクト生成

（５－１）コンター、等値面および矢印の描画オブジェクト生成

meshfield.draw(frame = -9999, arrow = -9999, subdivision = 2, iso_side_surface = 1)

メッシュフィールドによる描画プロセスを完了するメソッドです。このメソッドが呼び出されないとメッシュフィールドによる描画は行われません。また、このメソッドの呼び出し以降にメッシュフィールドオブジェクトに対するメソッドが呼ばれても有効ではありません。
このメソッドの引数にオプションとして下記の描画条件を与えることができます。

・フレームを線で描画する場合の、線の描画属性を与えます。
frame ： 線属性ID または線属性リスト［色（RGBT）］
・ベクトル値を矢印で表示する場合の矢印の描画属性を与えます。
arrow ： 矢印属性ID または矢印描画属性リスト［色（RGBT）、矢半径、矢高さ]
（例）mobj.draw(skip=1,frame=[0,0,1,0.5],arrow=[1,0,0,1,0.2,0.2])
・構造格子におけるコンター描画時のメッシュ細分化を制御します。（注[4]）
subdivision ： 0:細分化を行わない。N:最大N階層の細分化を行う。(N <= 2)
・構造格子からの等値面描画時に側面処理ありおよび側面処理なしを設定する。
iso_side_surface：0：側面処理なし、1：側面処理あり

（５－２）構造メッシュのボリュームレンダリング描画オブジェクト生成

meshfield.drawVolume(color = [r,g,b,a],intensity = 0.5,slope_factor = 2)

ボリュームレンダリング描画オブジェクト生成します。
ボリュームレンダリングは構造メッシュ格子上に分布しているスカラー値を､色の濃淡を利用して描画します。
color:色を RGBA（それぞれ 0～1.0 の範囲）で指定します

[4] 構造メッシュのコンター描画において、１区間の値の差が全体の差(最大値-最小値)の１０パーセント以上あるとき､区間を半分に細分化していく処理を最大２回繰り返すことができます。この細分化処理により､メッシュ分割数が少ない時でもコンターが滑らかに描画できます。また、色相範囲による色指定において一区間の値の変動が大きすぎて綺麗なグラデーションにならない場合に利用することができます。

intensity:表示の強さを指定します。（0〜1.0）大きいほど強くなります。
slope_factor:表示のボカシ具合を調整します。大きいほどシャープになります。

3.3.3.4 描画属性

描画関数に描画オブジェクトの色や大きさなどを指定するために、引数として描画属性IDまたは描画属性値のPythonリストを指定します。描画属性IDは描画属性値データを１つの整数値で表わしたものであり、描画属性ファイルによって描画属性IDと描画属性値データを関係付けます。描画属性ファイルの指定が無い場合に、描画属性IDが使われた時はデフォルト値が描画時に使用されます。描画属性ファイルの書式は描画対象の種類を示すキーワードに続いて属性ＩＤ及び属性値データを１行に記述します。属性ＩＤには１以上の整数値を使用します。

描画属性ファイルの記述例を以下に示します。

```
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
```

LineAttr及びPointAttrは、それぞれ線分と点の描画属性指定キーワードです。#で始まる行はコメント行です。
描画属性指定キーワードを下記（１）に、描画属性値を（２）に示します。

1）キーワード
LineAttr：　線分
PointAttr：　点
PolygonAttr：　三角形、四角形、・・多角形
PolylineAttr：　ポリライン
DiskAttr：　円盤
Ellipse1Attr：　楕円盤１（中心1 点の座標指定）

Ellipse2Attr： 楕円盤２（焦点２点の座標指定）
CylinderAttr： 円柱
SphereAttr： 球
Ellipsoid1Attr：楕円体１（中心1 点の座標指定）
Ellipsoid2Attr：楕円体２（焦点２点の座標指定、長軸2a）
TetraAttr： 四面体
CubeAttr: 立方体
ArrowAttr： 矢印
TextAttr： テキスト
CContourAttr： 連続コンター
DContourAttr： 不連続コンター
ClevelAttr： 等値面

2）描画属性
（色（RGB、透明度）は、0.0～1.0 で指定します。）
LineAttr： ID,色(RGB,透明度)
PointAttr： ID,色(RGB,透明度)
PolygonAttr： ID,色(RGB,透明度)
PolylineAttr： ID,色(RGB,透明度)
DiskAttr： ID,色(RGB,透明度),半径,法線ベクトル(XYZ)
Ellipse1Attr： ID,色(RGB,透明度),径長(a,b),径a方向ベクトル(XYZ),法線ベクトル(XYZ)
Ellipse2Attr： ID,色(RGB,透明度),径長(a),法線ベクトル(XYZ)
CylinderAttr： ID,色(RGB,透明度),半径
SphereAttr： ID,色(RGB,透明度),半径
Ellipsoid1Attr：ID,色(RGB,透明度),径長(a,b,c),径a方向ベクトル(XYZ),径c方向ベクトル(XYZ)
Ellipsoid2Attr：ID,色(RGB,透明度),径長(a)
TetraAttr： ID,色(RGB,透明度)
CubeAttr: ID,色(RGB,透明度)
ArrowAttr： ID,色(RGB,透明度),矢半径,矢高さ,(長さ倍率[5])
TextAttr： ID,色(RGB,透明度),サイズ
CContourAttr： ID,最小値色(RGB,透明度),最大値色(RGB, 透明度)
DContourAttr： ID,色(RGB,透明度)
ClevelAttr： ID,色(RGB,透明度)

描画属性ファイルの記述規約は以下のとおりです。
. データ値の間は空白で区切ります。
. １桁目が"#"の行はコメント行です。
. 同じＩＤ指定がある場合は、先に書かれた属性値が採用されます。
. 属性ファイルの指定が無い時は、デフォルト値を使用します。

描画属性ファイルに無い正の整数値を指定した場合は、描画属性ファイルの色指定が繰り返し使用

[5] 「長さ倍率」は描画属性を描画関数で直接指定する時のオプションです。従って「長さ倍率」は属性指定ファイルには記述しません。「長さ倍率」が描画関数に指定されていない時は１倍となります。

されます。属性ID値がゼロの場合は白、負値の場合は黒になります。
描画属性ファイルが無い場合、ID値に対応するデフォルト色は以下のようになります。

属性ID値	色
1	red
2	green
3	blue
4	yellow
5	magenta
6	cyan
7	orange
8	purple
9～	1～繰り返し
ゼロ	white
負値	black

α値は全て1となります。

描画属性ファイルの指定方法

描画属性ファイルを描画時に使用するためには、GOURMETの描画画面のメニューから描画属性ファイルを指定します。

描画関数での直接指定

描画属性を直接Python描画関数に渡る場合は、上記の2)描画属性からＩＤを除いた数値をPythonリストにまとめて描画関数の最後の引数に渡します。

```
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]) # 長さ倍率を用いた例
```

メッシュフィールドのdrawメソッドでメッシュ枠と矢印の描画属性を与えた例

```
meshf.draw(frame=[0,1,1,0.6],arrow=[1,0,0,1,0.1,0.1,1.5])
```

3.3.3.5 Cognac 描画クラス

Cognac描画クラスは、粗視化分子動力学シミュレータCOGNACのＵＤＦデータをGOURMETで高速に描画するためのPython拡張クラスです。この描画クラスを使用すれば、描画を高速に行えるだけでなく、Viewer画面でAtomあるいはBondをピッキングすることができます。

この描画クラスを使用するためには以下のようなデータ構造を持つ必要があります。

データ構造

Cognac描画クラスを利用するためには、分子データ、AtomデータおよびBondデータが以下のような関係になっている必要があります。

```
分子の構造体配列--+--(1) 分子の名前（文字列)
                  |--(2)Atomの構造体配列--+--(3)Atomのタイプ（文字列または整数)
                  |                       +--(4)Atomの座標構造体--+-- X（実数:float/double)
```

```
|                                             |-- Y (実数:float/double)
|                                             +-- Z (実数:float/double)
+--(5)Bondの構造体配列--+--(6)Bondのタイプ (文字列または整数)
                        |--(7)Bond始点のAtomの配列インデックス (整数)
                        +--(8)Bond終点のAtomの配列インデックス (整数)
```

これらのデータは１つの構造体にまとめられている必要はありません。Atomがどの分子に属しているかといった関係は配列インデックスで示されていれば十分です。例えば分子データの[0]番目には、配列[0,0]と[0,1]のAtom が属していて、Bond[0,0]の始点と終点に対応するAtomの配列インデックスは[0,0]と[0,1]などと関係付けられていれば良いのです。

上図の(1)～(8) のデータを持つＵＤＦデータ名をCognac描画クラスのコンストラクタに指定します。

Cognacでは下記のＵＤＦデータ名を指定しています。

(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'

例えば、２つのAtomからなる分子が１つ存在し、Bond両端のAtomのインデックスが0および1であるようなＵＤＦデータは以下のようになります。このUDFデータはCognacで使う描画データ部分のみを抜き出したものです。データ構造は"Set_of_Molecules"および"Structure"に分かれていますが配列インデックスによって分子・Atom・Bondの関係が付けられています。

```
¥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}
```

クラスとメソッドの解説
COGNAC描画クラスは、Atom描画クラス、Bond描画クラスおよびAtom座標を操作するクラスからなります。

Atom描画クラス

・オブジェクト作成
atom = createAtom(molecule_type_path,atom_type_path,atom_data_path,atom_position_path)

molecule_type_path：molecule種別名のUDFデータパスを指定します。描画時の色分けのために必要です。（省略可）
atom_type_path:atom種別名のUDFデータパスを指定します。描画時の色分けのために必要です。（省略可）
atom_data_path：atomデータのパス名を指定します。ピッキングのために必要です。
atom_position_path：atomの位置座標値を持つUDFデータ名を指定します。必須。

・描画メソッド
atom.draw(index=[],attr=-1,type='line',moltypedic=0,atomtypedic=0,coord=0)

index：描画するatomの配列インデックスを指定します。[]の場合はatom全てを描画します。

[i]の場合は分子単位で描画します。[i,j]の場合はatom単位で描画します。
attr：色の指定です。描画属性ID、描画属性値またはキーワード"atom","molname","mol"のいずれかを与えます。
attrが「描画属性ID」「描画属性値」の場合、「moltypedic」「atomtypedic」等が指定されていても無視されます。
attrが"mol"の時、molecule名が同じであっても分子毎に色分けして描かれます。
attrが"molname"の場合、molecule名の出現順に描画属性IDを使用するか、moltypedicの属性を利用して描きます。
type:"point","line","ball","ball-stick","stick","rod","volume"のいずれかを指定します。
"line"は単独（1Atom分子）の時に点を描画します。"stick","rod","volume"は単独の時に球を描画します。
moltypedic：molecule種別名（molecule_type_pathデータ名で指定される）に対する描画属性をPython辞書で与えます。
atomtypedic：atom種別名（atom_type_pathデータ名で指定される）に対する描画属性をPython辞書で与えます。
（例）"atom1":1,"atom2":2
coord：CognacCoordクラスのオブジェクトを与えます。描画時にAtom座標を変更したい時に使用します。（後述の「座標マッピング」を参照してください。）

・座標取得メソッド
（１）個別座標取得（Atom座標をリスト形式で返します）
atom.getCoord(index)

（２）平均座標取得（Atom座標の全平均または分子平均をリスト形式で返します）
atom.averageCoord(index=())

全座標平均：xyz = atom.averageCoord()
分子座標平均：xyz = atom.averageCoord([3])

Bond描画クラス

・オブジェクト作成
```
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：Atom描画クラスと同じです。
bond_type_path：bond種別名のUDFデータ名を指定します。描画時の色分けのために必要です。

bond_atom1_id_path,bond_atom2_id_path：atomのインデックスデータを持つUDFデータパスを指定します。
bond_data_path：bondデータ名を指定します。

・描画メソッド
```
bond.draw(
        index=[],attr=-1,type='line',
        moltypedic=0,atomtypedic=0,bondtypedic=0,coord=0,maxlen=0.0)
```

index：描画するbondの配列インデックスを指定します。[]の場合はbond全てを描画します。[i]の場合は分子毎に描画します。[i,j]はbond毎に描画します。
attr:色属性を指定します。描画属性ID、描画属性または、キーワード"atom","bond","molname","mol"のいずれかを与えます。
attrが「描画属性ID」「描画属性」の場合、「moltypedic」「atomtypedic」等が指定されていても無視されます。
attrが"mol"の時、名前が同じであっても分子毎に色分けして描きます。
attrが"molname"の場合、名前の順あるいはmoltypedicの属性を利用して描きます。
type:"point","line","ball-stick","stick","rod","volume"(ellipsoid)のいずれかを指定します。
moltypedic：molecule種別名（molecule_type_pathデータ名で指定される）に対する描画属性をPython辞書で与えます。
atomtypedic：atom種別名（atom_type_pathデータ名で指定される）に対する描画属性をPython辞書で与えます。
bondtypedic：bond種別名（bond_type_pathデータ名で指定される）に対する描画属性をPython辞書で与えます。
coord：CognacCoordクラスのオブジェクトを与えます。描画時にAtom座標を変更したい時に使用します。
maxlen:0.0より大きい時、描画しようとするBond長がそれ以上の場合、描かないようにします。

座標マッピング

・オブジェクト作成
```
coordmap = CognacCoord()
```
インデックスと座標のマップオブジェクトを作成します。失敗した場合は、Noneを返します。

・インデックスと座標のセット
```
coordmap.set(index,coord)
```
index：マップのキーです。[i,j] または(i,j) 形式でインデックスを指定します。
coord：３次元座標値です。[x,y,z] または(x,y,z) 形式です。
成功した場合は、indexの次数を返します。失敗した場合は、Noneを返します。

・サイズ取得
```
nsize = coordmap.size()
nsize = len(coordmap)
```

・インデックスから座標を取得
coord = coordmap.get(index)

3.3.3.6 ３次元移動（平行移動・回転移動）アクション用メソッド

・移動後座標の取得
udf.GetMovedObject(drawTypeList)
drawTypeList:座標を取り出す描画タイプのリストです。
描画タイプ名は描画メソッド名と同じです。現状では下記の描画タイプを使用できます:
["sphere","point","line","cylinder"]
３次元移動（平行移動・回転移動）アクションの移動後の座標を取り出します。このメソッドはアクションスクリプトの中でのみ実行できます。
戻り値は次の形式のリストです：
[(アクションターゲットロケーション,移動後座標), ...]

ここでアクションターゲットロケーションとは３次元移動（平行移動・回転移動）アクションのターゲットデータのデータロケーション（データ名＋配列インデックス）です。

以下に平行移動アクションの例を示します。
アクションのターゲットは、"Structure.Position.mol[].atom[]"ですから、
udf.GetMovedObject(["sphere"])の戻り値は、
[('Structure.Position.mol[0].atom[6]',[0.3986, 3.3675, 7.3317]),...]
のリストになります。

```
action Structure.Position.mol[].atom[] : $Translate():¥begin

# pathxyzList : Positions after the translation.
# pathxyzList = [('Structure.Position.mol[0].atom[6]',[0.3986, 3.3675, 7.3317]),...]
pathxyzList = _udf_.GetMovedObject(["sphere"])
if type(pathxyzList) == type([]):
        for pathxyz in pathxyzList:
                print pathxyz[1],":",pathxyz[0]
                _udf_.put(pathxyz[1],pathxyz[0])

¥end
```

・３次元描画対象のセレクトおよびリセット

udf.selectDrawObject(locationsList, drawTypeList)
locationsList:UDFデータロケーション（データ名＋配列インデックス）のリストです。
drawTypeList:セレクトする描画タイプのリストです。
描画タイプ名は描画メソッド名と同じです。現状では下記の描画タイプを使用できます:
["sphere","point","line","cylinder"]

UDFデータと関連付けられた３次元描画対象をセレクト状態にします。セレクト状態にする描画対象のUDFデータロケーションのリストおよび描画タイプを引数で指定します。

udf.unselectDrawObject(drawTypeList=None)
drawTypeList:セレクトをリセットする描画タイプのリストです。
描画タイプ名は描画メソッド名と同じです。現状では下記の描画タイプを使用できます:
["sphere","point","line","cylinder"]
３次元描画対象のセレクト状態をリセットします。drawTypeListを指定しない場合は、全てのセレクト状態がリセットされます。

・スクリプトの履歴制御

udf.EraseFromHistory()
この命令を含むスクリプト文全体を実行履歴に残さないようにします。
描画アクションスクリプトに続けてデータの変更を含む３次元移動（平行移動・回転移動）などのアクションを実行してスクリプト実行履歴がスクリプト画面に残っていると、ユーザーが描画アクションと間違えてデータ変更のスクリプトを実行してしまう場合があります。間違いやすいスクリプトをあえて実行履歴に残さないようにするために使用します。

第４章　ユーティリティモジュール

ここではUDFManager以外のPythonスクリプトモジュールについて説明します。

4.1　UDF ファイル操作用ユーティリティ

UDFファイル操作用ユーティリティスクリプトには以下の機能があります：
・２つのUDFファイルのデータ定義を比較して相違点を出力する。
・１つのUDFファイルから他のUDFファイルにデータをコピーまたは変換する。
・２つのUDFファイルのデータを比較する。
・UDFファイルの１レコードのデータまたは全データを消去する。

　スクリプトファイル名：UDFOperations.py
　スクリプトファイル位置：GOURMET_200X/python/

以下にUDFファイル操作用ユーティリティの解説をします。

・UDFファイルのデータ定義比較
compareUDFDefinition(targetUdf1,targetUdf2,include_class_type = 0)
　　UDFファイルの定義を比較して相違点を出力します。
　　targetUdf1,targetUdf2: UDFファイル
　　include_class_type=1とすると"class"として定義された構造データを含めて比較します。
　　（注）定義順序が異なる場合は現バージョンでは検出できません。

以下に２つのUDFファイルの定義を比較するスクリプト使用例および結果を示します。

定義１：convert_test_def.udf

```
¥begin{def}
Root:{
 a[]:{
  a1:string
  a2:select{"A","B"}
  A:int
  B:long
 }
 b[]:{
  b1:int
  b2:long
  b3[]:int
 }
 c[]:{
  c1:float
  c2:double
  c3[]:float
 },
```

```
};
¥end{def}
```

定義２：convert_test_dif.udf

```
¥begin{def}
Root:{
 a[]:{
  a1:string
  a2:select{"A","B"}
  A:int
  B:long
 }
 b[]:{
  b1:int
  b2:float
  b3[]:int
 }
 c[]:{
  c1:float
  x2:double
  c3[]:float
 },
};
¥end{def}
```

使用例：

```
from UDFManager import *
import UDFOperations
targetUdf1 = UDFManager(r'convert_test_def.udf')
targetUdf2 = UDFManager(r'convert_test_dif.udf')
UDFOperations.compareUDFDefinition(targetUdf1,targetUdf2)
```

結果：

```
different type: Root.b[].b2 type: long <-> float
different: Root.c[].c2 in convert_test_def.udf
different: Root.c[].x2 in convert_test_dif.udf
```

・データ変換またはコピー

convertUDFCurrentRecord(sourceUdf, destinationUdf, dataName)

UDFファイルから他のUDFファイルへカレントレコードのデータを変換します。

データ変換方法は同種のデータタイプかつ同じデータ名のデータをコピーします。同種のデータタイプとは下記の各データタイプを同一タイプとすることです：

整数タイプグループ："int","short","long","ID","<*,ID>"

実数タイプグループ："float","single","double"

文字列タイプグループ："string","select","KEY","<*,KEY>"

sourceUdf: 変換元UDFファイル

destinationUdf: 変換先UDFファイル
dataName: 変換またはコピーするデータ名。dataNameが構造データの場合は構造データ内の全てのデータが変換されます。dataName=""とすると全てのデータ名を対象にします。
全く同じデータ定義の場合、データがそのままコピーされます。

使用例：

```
from UDFManager import *
import UDFOperations
sourceUdf = UDFManager(r'convert_test_data.udf')
sourceUdf.jump(0)
destinationUdf = UDFManager(r'convert_test_template.udf')
destinationRecordNumber = destinationUdf.totalRecord()
destinationUdf.newRecord( sourceUdf.getRecLabel() )
destinationUdf.jump( destinationRecordNumber + 1 )
UDFOperations.convertUDFCurrentRecord( sourceUdf, destinationUdf, dataName = "" )
```

convertUDFFile(sourceUdf, destinationUdf, topName = "", startRecord = -1, endRecord = -2)

UDFファイルから他のUDFファイルへ指定したレコード範囲のデータを変換します。
destinationUdfにレコードが無い場合は作成します。
変換の方法は同種のデータタイプかつ同じデータ名のデータをコピーします。
sourceUdf: 変換元UDFファイル
destinationUdf: 変換先UDFファイル
topName: 変換またはコピーするトップデータ名。topNameが構造データの場合は構造データ内の全てのデータが変換されます。topName=""とすると全てのデータ名を対象にします。
全く同じデータ定義の場合、データがそのままコピーされます。
startRecord, endRecord: 変換レコード範囲を指定します。endRecord=-2とすると最終レコードを指定することになります。
（注）定義順序が異なっても同名同種タイプであればデータをコピーします。

使用例：

```
from UDFManager import *
import UDFOperations
sourceUdf = UDFManager(r'convert_test_data.udf')
destinationUdf = UDFManager(r'convert_test_template.udf')
UDFOperations.convertUDFFile(sourceUdf, destinationUdf)
```

・データ値の比較

compareUDF(targetUdf1,targetUdf2, startRecord = -1, endRecord = -2, rigid_or_homogeneous_type = 1, verbose = 0)

UDFファイルの値を比較して相違点を出力します。
targetUdf1,targetUdf2: 比較するUDFファイル
startRecord,endRecord: レコード範囲を指定します。endRecord=-2とすると最終レコードを指定することになります。
rigid_or_homogeneous_type = 1 とすると全てのデータを比較します。
rigid_or_homogeneous_type = 0 とすると同種タイプのデータのみを比較します。
verbose = 0 の場合、見つかった全相違点数のみを出力します。

verbose = 1 の場合、見つかった相違点を全て出力します。

使用例：

```
from UDFManager import *
import UDFOperations
targetUdf1 = UDFManager(r'convert_test_data.udf')
targetUdf2 = UDFManager(r'destination2.udf')
UDFOperations.compareUDF(targetUdf1,targetUdf2,verbose = 1)
```

結果例：

```
DIFFERENT(VALUE): Root.b[].b2
DIFFERENT(DEF1): Root.c[].c2
DIFFERENT(DEF2): Root.c[].x2
DIFFERENT DATA NUMBER: 3 RECORD AT: -1
DIFFERENT(VALUE): Root.b[].b2
DIFFERENT(DEF1): Root.c[].c2
DIFFERENT(DEF2): Root.c[].x2
DIFFERENT DATA NUMBER: 3 RECORD AT: 0
TOTAL DIFFERENT DATA NUMBER: 6
```

・データ削除

eraseUDFRecordData(targetUdf, dataname = '')

カレントレコードの指定したデータ名のデータを削除します。
dataname=""とすると全てのデータ名を対象にします。

eraseAllUDFData(targetUdf)

UDFファイルの全データを削除します。

4.2 構造メッシュおよび非構造メッシュ操作用ユーティリティ

メッシュデータを描画および表面抽出処理するためのユーティリティモジュールについて説明します。

メッシュ操作用ユーティリティは、NASTRANデータコンバーターツールやAVSデータコンバーターツールで利用されます。一部のユーティリティクラスはメッシュフィールド描画関数から利用しやすいインターフェースを提供しています。

メッシュ操作用ユーティリティはPythonスクリプトから利用できるC++プログラムで作成されています。PythonスクリプトファイルおよびC++プログラムファイルは以下のとおりです：

スクリプトファイル名：Geometry3dElementUtilities.py
Cモジュール名：Geometry3dElementObjectPython（shaker.dll(so)と一体）
Cソース名：Geometry3dElementObjectPython.cpp（GOURMET_200X/src/pymgr）
スクリプトファイル位置：GOURMET_200X/python/

4.2.1 ユーティリティモジュール

以下にPythonスクリプトから利用できる関数およびクラスの説明をします。

関数 GetIndexFromSequence(meshsize, sequence,
index_priority = -1, space_dimension = -1)

１次元のシーケンス番号（0,1,2,3,4,5,6,...）を N 次元メッシュ点の配列インデックス[i,j,k]（[0,0,0],[1,0,0],[2,0,0],...,[0,1,0],...など）に変換します。
変換時に、メッシュサイズ（各座標軸方向の区間数）および[i,j,k]配列の優先順位が必要です。
meshsize：メッシュサイズ（各座標軸方向の区間数）をリストで与えます。
sequence：ゼロから始まるシーケンス番号。
index_priority：[i,j,k]配列の優先順位
index_priority >= 0 の時、末尾優先。
すなわち、[0,0,0],[0,0,1],[0,0,2],...,[0,1,0],[0,1,1]...
index_priority < 0 の時、先頭優先
すなわち、[0,0,0],[1,0,0],[2,0,0],...,[0,1,0],[1,1,0]...
space_dimension：空間次元数。-1 の時はメッシュサイズリストの長さを用いる。

関数 GetSequenceFromIndex(meshsize, ijk_list,
index_priority = -1, space_dimension = -1)

メッシュ点配列インデックスをシーケンス番号に変換します。
上記 GetIndexFromSequence(...)の逆関数です。
meshsize：メッシュサイズ（各座標軸方向の区間数）をリストで与える。
ijk_list：メッシュ点配列インデックスのリスト[i,j,k]。
index_priority：[i,j,k]配列の優先順位
index_priority >= 0 の時[0,0,0],[0,0,1],[0,0,2],...,[0,1,0],[0,1,1]...
index_priority < 0 の時[0,0,0],[1,0,0],[2,0,0],...,[0,1,0],[1,1,0]...
space_dimension：空間次元数。-1 の時はメッシュサイズリストの長さを用いる。

クラス UTILITY_INTEGER_SET

整数値を登録し、後である整数値が登録済みかどうか調べることができます。Python のマップより大量のデータを高速に処理するためのクラスです。

コンストラクタ：UTILITY_INTEGER_SET()

メソッド：

clear()

全ての登録内容を消去する。

add(int_list)

整数値のリストを引数にして整数値を登録する。

size()

登録した個数を返す。

exist(int_value)

整数値が登録済みか調べる。

クラス UTILITY_ID_INTEGER_MAP

整数 ID に対する整数値を登録するためのクラスです。

コンストラクタ：UTILITY_ID_INTEGER_MAP()

メソッド：

clear(self)

全ての登録内容を消去する。

add(id,val)

整数 ID に対する整数値を登録する。

size()

登録した個数を返す。

get(id)

整数 ID に対する整数値を返す。登録されていない場合は None を返す。

クラス UTILITY_ID_FLOATS_MAP

整数 ID に対する複数の実数値を登録するためのクラスです。

コンストラクタ：UTILITY_ID_FLOATS_MAP

メソッド：

clear()

全ての登録内容を消去する。

add(id,val)

整数 ID に対する複数の実数値を登録する。val は数値のリストかタプル。
整数値でも実数値として登録される。

size()

登録した個数を返す

get(id)

整数 ID に対する実数値を返す。登録されていない場合は None を返す。

クラス UTILITY_COLOR_LIST_CREATOR

値に対する描画色（RGBA）を生成するためのクラスです。メッシュフィールドのコンター描画で利用するコンター色生成処理をクラスとして利用できるようにしたものです。

コンストラクタ：UTILITY_COLOR_LIST_CREATOR()

メソッド：

setMinMax(minvalue,maxvalue)
　最大値および最小値を登録する
getMinMax()
　登録済の最大値および最小値を返す
setColor(rgba_list)
　単色（RGBA）を登録する。
setMinMaxColor(minrgba_maxrgba_list)
　最小値色および最大値色を登録する。
setHSV(rgba_angle_list)
　HSV カラーを登録する。
addDiscreteColor(uppervalue,rgba_list)
　離散色を登録する
getColor(value)
　値に対する色（RGBA）を返す
getColorMinMax(value,minvalue,maxvalue)
　最大値および最小値を与えて値に対する色（RGBA）を返す

クラス　UnstructuredMeshBoundaryCreator
　３次元有限要素の表面要素を抽出します。
コンストラクタ：UnstructuredMeshBoundaryCreator()
メソッド：
setCoordinate1(cid, rid, ctype, anodeid, bnodeid, cnodeid)
　３つの節点 ID によって座標系を登録する。
　cid：座標系 ID
　rid：節点が属する座標系 ID
　ctype：座標系タイプ（0:CARTESIAN, 1:CYLINDRICAL, 2:SPHERICAL）
　　座標系タイプは NASTRAN の座標系指定方法に対応しており、それぞれ
　　CORD1R、CORD1C および CORD1S に対応しています。
　anodeid, bnodeid, cnodeid：節点 ID
setCoordinate2(cid, rid, ctype, axyz, bxyz, cxyz):
　３つの座標値によって座標系を登録する。
　cid：座標系 ID
　rid：座標値が属する座標系 ID
　ctype：座標系タイプ（0:CARTESIAN, 1:CYLINDRICAL, 2:SPHERICAL）
　　座標系タイプは NASTRAN の座標系指定方法に対応しており、それぞれ
　　CORD2R、CORD2C および CORD2S に対応しています。
　axyz, bxyz, cxyz：座標値のリストまたはタプル

sizeCoordinate():
　登録済の座標系数を返す。
addNode(node_id,cord_id, x,y,z):
　節点を登録する。
　node_id：節点 ID
　cord_id：節点が属する座標系 ID
　x,y,z：座標値
getNode(sequence, with_cid = 1)

配列インデックスから節点データを返す。[node_id,cord_id, [x,y,z]]形式
with_cid：0の場合、cord_idを除いた[node_id, [x,y,z]]形式を返す。

getNodeList(output_mask)

節点データリストを返す。[[id,cid,[x,y,z]],...]形式等
output_mask：出力形式を与える
output_mask = 1 : id, 2 : cid, 4 : [x,y,z].を返す。
例えば、output_mask = 1|4 = 5の場合、[[id,[x,y,z]],...]を返す。

getNodeByID(node_id, with_cid = 1)

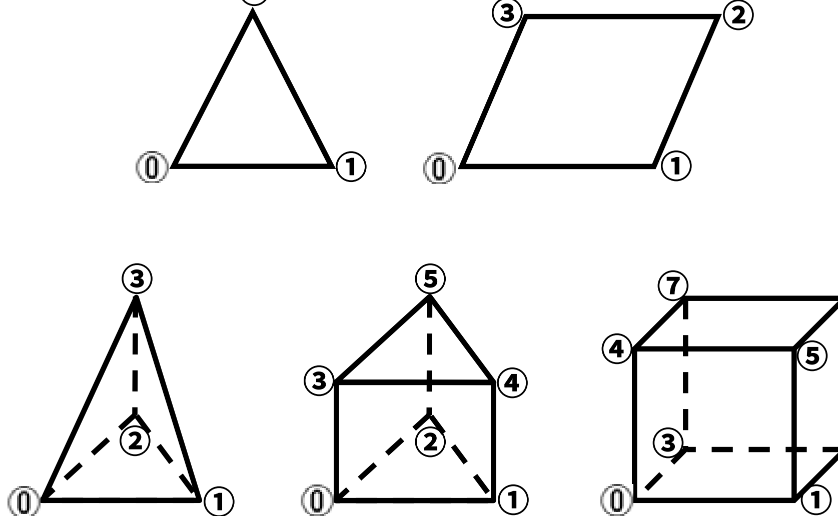
節点IDから節点データを返す。[node_id,cord_id, [x,y,z]]形式
with_cid：0の場合、cord_idを除いた[node_id, [x,y,z]]形式を返す。

sizeNode()

節点数を返す。

addElement(elem_type, elem_id, prop_id, node_id_list):

要素を登録する。
elem_type：要素タイプ名（"point"/ "line"/ "tri"/ "quad"/ "pyramid"/ "tetra"/ "penta"/ "hexa"）または数値タイプ番号
（POINT:1,LINE:2,TRIANGLE:3,QUADRANGLE:4,PYRAMID:5,TETRAHEDRON:6,PENTAHEDRON:7,HEXAHEDRON:8)
elem_id：要素ID
prop_id：プロパティID
node_id_list：節点IDのリスト
要素の節点指定順序はNASTRANに準拠しており、下図のようになります：



getElement(sequence)

要素の配列インデックスから要素データを返す。要素タイプはタイプ番号で返す。
[elem_type_number, id, pid, node_id_list]形式
sequence：要素の配列インデックス

getElementByID(elem_id):
　　要素 ID から要素データを返す。
　　[elem_type_number, id, pid, node_id_list]形式
　　elem_id：要素 ID
sizeElement()
　　要素数を返す。
queryElementDimension(dimension_list = (2,3))
　　次元別要素が存在するかどうか調べる。
　　dimension_list：0:点要素、1:線要素,2:面要素,3:ソリッド要素のタプル
　　dimension_list = (2,3)の場合は面要素またはソリッド要素の存在
searchPartialRegion(deg_angle = 180.0, classification = 0, max_color = 0)
　　部分領域への分割処理を行う。詳細は次節で説明します。
　　deg_angle：分割処理に使用する法線ベクトルの角度（°）
　　classification：分割処理方法
　　 (0/1)法線ベクトルの角度, (2)要素のプロパティ, (3)角度およびプロパティ
　　max_color：最大使用色（ゼロの場合は必要な色数分使用）
sizeFace()
　　部分領域分割後の境界面または境界辺の数を返す。
getFace(index)
　　部分領域分割後の境界面または境界辺のデータを返す。
　　index：境界面または境界辺の配列インデックス
　　[FACE-ID, [NODE-ID,...], [[ELEMENT-ID, SIDE-ID],...]]形式
getFaceList(output_mask, nnode_list = 0)
　　部分領域分割後の境界面または境界辺のリストを返す。
　　output_mask：出力形式を制御するビットマスク
　　　　output_mask=0 の場合は全データを返す。
　　　　[[FACE-ID,NODE-ID-LIST,ELEMENT-ID-SIDE-ID-LIST],...]形式
　　　　output_mask に次のビットマスクを指定するとそのデータを返す。
　　　　1 : FACE-ID, 2 : NODE-ID-LIST, 4 : ELEMENT-ID-SIDE-ID-LIST
　　　　例えば、output_mask = 2 の時、 [[NODE-ID,...],...]のみを返す。
　　nnode_list：出力する境界面または境界辺の節点数のタプルまたはリスト
　　　　ゼロを指定すると全ての節点数の境界面または境界辺を返す。
　　　　例えば、nnode_list = (3,4)の時、'triangle'および'quadangle' を返す。
sizePartialRegion()
　　部分領域分割後の部分領域数を返す。
getPartialRegionColor(region_index)
　　部分領域分割後の部分領域数を色分けする最小色数のインデックスを返す。
　　region_index：部分領域インデックス
sizePartialRegionFace(region_index):
　　部分領域分割後の部分領域に属する境界面または境界辺の個数を返す。
　　region_index：部分領域のインデックス
　　# Face = [FACE-ID, NODE-ID-LIST, [[ELEMENT-ID, SIDE-ID],...]]
getPartialRegionFace(region_index, face_index):
　　部分領域分割後の部分領域に属する境界面または境界辺データを返します。
　　[FACE-ID, NODE-ID-LIST, [[ELEMENT-ID, SIDE-ID],...]]形式
　　SIDE-ID は要素の辺または側面を指定するためのインデックスで、要素形状に応じて要素点が

下記のようになっている部分要素です。
例えば、四面体要素ではSIDE-ID=0 は、要素内の要素点順序{1,2,3}よりなる三角形です。

・三角形要素　[SIDE-ID]={要素点順序番号}
辺：[0]={1,2}, [1]={2,0}, [2]={1,2}

・四角形要素
辺：[0]={0,1}, [1]={1,2}, [2]={2,3}, [3]={3,0}

・四面体要素
面：[0]={1,2,3}, [1]={0,3,2}, [2]={0,1,3}, [3]={0,2,1}

・五面体要素
面：[0]={0,2,1}, [1]={3,4,5}, [2]={0,1,4,3}, [3]={1,2,5,4}, [4]={0,3,5,2}

・六面体要素
面：[0]={ 0,4,7,3 }, [1]={ 1,2,6,5 }, [2]={ 0,1,5,4 },
[3]={ 2,3,7,6 }, [4]={ 0,3,2,1 }, [5]={ 4,5,6,7 }

searchBoundaryFace()
３次元要素の表面抽出処理のみを行います。
抽出処理後の面データはgetFaceList(...)などで取り出します。

4.2.2　部分領域抽出処理

部分領域抽出処理は以下の手順によって行っています。

（１）　表面要素の抽出

３次元要素(CTETRA：四面体、CPENTA：五面体、CHEXA：六面体）については、まず外殻表面要素の抽出処理を行います。
処理の手順は、全ての要素を巡回し、各面について他の要素の面となっているかを調べ、他の要素の面となっていない面を外殻表面として登録します。

２次元要素(CTRIA3：三角形, CQUAD4：四角形)についても同様に、全ての要素を巡回し各辺について他の要素の辺となっているかを調ます。２次元要素の処理は以下の「部分境界の作成」処理と同時に行っています。

（２）　部分境界の作成

３次元要素(CTETRA：四面体、CPENTA：五面体、CHEXA：六面体）については、外殻表面要素の法線ベクトルが、指定された角度よりも大きいときに境界辺を登録します。また、３次元要素のプロパティ ID が違うときに境界辺を作るように指定することができます。

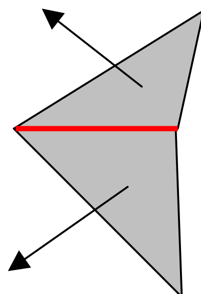


図 4. 1　法線ベクトルによる部分領域の抽出条件

２次元要素(CTRIA3：三角形, CQUAD4：四角形)については、要素自体の法線ベクトルが、指定された角度よりも大きいときに境界辺を登録します。また、２次元要素のプロパティ ID が違うときに境界辺を作るように指定することができます。

登録された境界辺について、境界辺が閉ループを作るように端点の除去処理を行います。例えば、次々に端点を除去して下図の赤で示した部分を除去します。

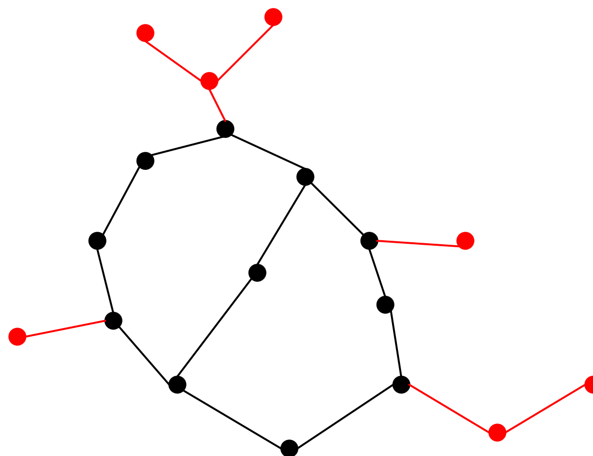


図 4.2　閉領域を作るための端点除去処理

（３）　最小色による塗り分け

３次元要素(CTETRA：四面体、CPENTA：五面体、CHEXA：六面体）の外殻表面要素についての部分領域分割処理では、各部分領域について隣接部分領域の色 ID を調べて、隣接部分領域とは違う最も小さな色 ID を順に割り当てて、最も少ない色で塗り分けることができます。

4.2.3　NASTRANデータコンバータースクリプト

4.2.3.1　NASTRANデータ読み込み

ファイル名：FEM_UDF_NASTRAN_BULK_READ.py
位置：GOURMET_200X/tool/NASTRAN
関数名：ReadNastranBulk(udfmgr, filename)
　　NASTRAN バルクファイルをFEM-UDFに読み込む。
　　udfmgr：UDFManager オブジェクト
　　filename：バルクファイルのファイルパス

使用方法：

```
from FEM_UDF_NASTRAN_BULK_READ import *
ReadNastranBulk( _udf_, nastran_bulk_filepath )
```

解説：
NASTRANバルクデータの以下に示す主要なデータをコンバートして、FEM-UDFデータ構造ファイルに格納します。

節点：GRID , GRID* (倍精度)
要素：２次元要素(CTRIA3：三角形, CQUAD4：四角形)
　３次元要素(CTETRA：四面体、CPENTA：五面体、CHEXA：六面体)
　但し、高次要素は対象としない。倍精度には対応。
座標系：直交直角座標系、円筒座標系および球面座標系に対応
　CORD1C,CORD1Rなどのカードがある場合、座標データを読み込んでFEM-UDFに格納し、描画時などに変換して使用します。
プロパティデータ/マテリアルデータ：２次元要素では厚みなどの３次元形状を作成するためのデータがあり、これはNASTRANではPropertyとして、定義されています。これについてはPID(Property ID)を、要素のグループ識別用にのみ用い、厚みなどのプロパティデータは無視します。また、このプロパティデータから、材料の物性を規定したマテリアルデータを参照していますが、これも要素のグループ識別用にのみ用い、マテリアルデータは無視しています。

4.2.3.2　NASTRANデータ書き込み

ファイル名：FEM_UDF_NASTRAN_BULK_WRITE.py
位置：GOURMET_200X/tool/NASTRAN
関数名：WriteNastranBulk(udfmgr, filename)
　　FEM-UDFの内容をNASTRAN バルクファイル形式で書き出す。
　　udfmgr：UDFManager オブジェクト
　　filename：バルクファイルのファイルパス

使用方法：

```
from FEM_UDF_NASTRAN_BULK_WRITE import *
WriteNastranBulk ( _udf_, nastran_bulk_filepath )
```

4.2.3.3　部分領域構成

ファイル名：FEM_UDF_REGION.py
位置：GOURMET_200X/tool/NASTRAN
クラス名：FEM_UDF_PARTIAL_REGION
メソッド：searchPartialRegion(
　　　　angle = 180.0, classification = 0, max_color = 0, group_index = -1)

FEM-UDF の内容を取り込んで、面の法線ベクトル角度あるいは要素のプロパティにより部分領域を構成する。処理結果を格納する UDF データについては、
「３．１　有限要素解析データ」を参照。

引数：

deg_angle：分割処理に使用する法線ベクトルの角度（°）
classification：分割処理方法
　(0/1)法線ベクトルの角度で分割
　(2)要素のプロパティで分割
　(3)角度およびプロパティで分割
max_color：最大使用色（ゼロの場合は必要な色数分使用）
group_index：処理結果を格納する"group[]"データの配列インデックス
-1 の場合は末尾に追加する。

使用方法：

```
from FEM_UDF_REGION import *
region = FEM_UDF_PARTIAL_REGION(_udf_)
region.searchPartialRegion(  angle_degree  )
```

4.2.3.4　描画

ファイル名：FEM_UDF_DRAW.py
位置：GOURMET_200X/tool/NASTRAN
関数名：DrawFemElement(udfmgr)
　要素を描画する。
　udfmgr：UDFManager オブジェクト

関数名：DrawPartialRegion(udfmgr, arg_group_index = -1, arg_region_index = -1)
　部分領域を描画する。

引数：

udfmgr：UDFManager オブジェクト
arg_group_index：描画対象データ"group[]"の配列インデックス
arg_region_index：描画対象部分領域データ"group[].region[]"の配列インデックス

使用方法：

```
# 要素描画
from FEM_UDF_DRAW import *
DrawFemElement( _udf_ )

# 領域描画
```

```
from FEM_UDF_DRAW import *
DrawPartialRegion( _udf_, arg_group_index = 0, arg_region_index = 0 )
```

第５章　サンプル

簡単なUDFデータファイルを用いたUDFManagerの使用例を示します。

5.1 UDFManager for Python のサンプル

下記のUDFデータに対して、UDFManagerクラスを使ってデータを参照・編集する例です。

```
sample.udf
¥begin{def}
class Point:{
  x:float
  y:float
  z:float
}
class Vertex:{
  id:ID
  position:Point
}
class Face: { // 辺のクラス
  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}
```

Pythonを起動して拡張ライブラリUDFManager.pyをインポートした後、上記のＵＤＦファイルを読み込み、データの参照、編集およびファイルへの書き出しを行います。

```
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
```

5.2 Python in GOURMET のサンプル

上記のUDFManager for Pythonのサンプルと全く同じ内容をGOURMET上で実行する場合のスクリプト例を示します。ここではスクリプトを実行する前に、GOURMETでUDFファイルを読み込んでおきます。

```
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")
```

PythonスクリプトをGOURMETで実行した結果は、以下のようになります。GOURMETではレコードを持つUDFファイルを読み込んだ場合、最初のレコードがカレントレコードになります。

```
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]]]
```

5.3 Python 描画関数のサンプル

描画関数に座標及び描画属性を引数として与えます。

```
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])
```

（メッシュフィールドの使用例１）：regularメッシュ

```
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) # ここでcontour を生成する
```

（メッシュフィールドの使用例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()
```

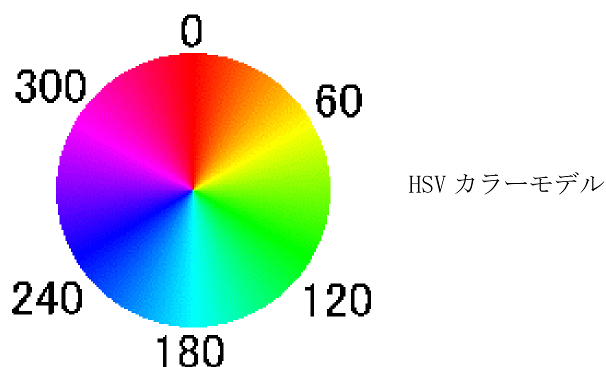
（メッシュフィールドの使用例３）：(5.1)サンプルＵＤＦを用いた非構造格子描画例

```
mlist=[]
mid=Location('parameter.vertex[0].id')
mpos=Location('parameter.vertex[0].position')
msiz=size('parameter.vertex[]')
for i in range(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()
```

5.4　色相範囲による描画のサンプル

メッシュフィールドクラスを利用したコンターおよび等値面の描画では、色相範囲指定による多色表示をすることができます。色相範囲とは、下図の HSV カラーモデルにおいて色相（HUE）角度に対応する値を指定することに対応します。

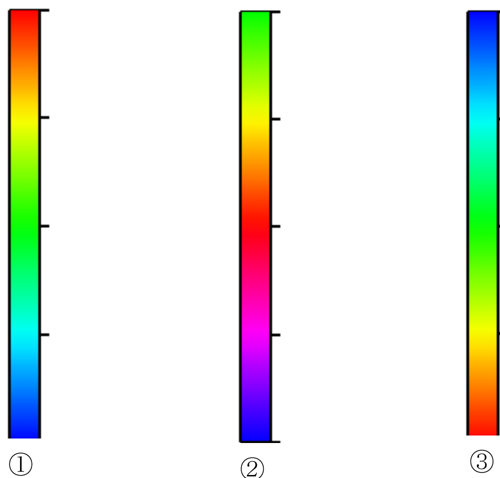


HSV カラーモデル

色相範囲指定による色指定は、以下の例のようになります。

①（最小値色 RGBA および方向付き色相範囲）=[0,0,1,1, -240]（青から負の向きに 240°）
②（最小値色 RGBA および方向付き色相範囲）=[0,0,1,1, 240]（青から正の向きに 240°）
③（最小値色 RGBA および方向付き色相範囲）=[1,0,0,1, 240]（赤から正の向きに 240°）

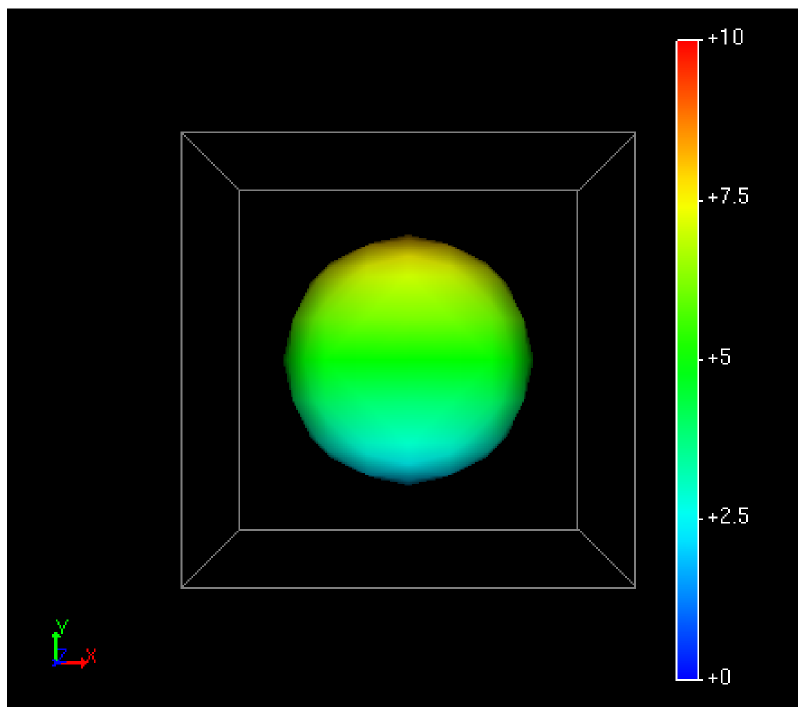
上記例を描画したときの凡例はそれぞれ次の図のようになります。



等値面描画の Python スクリプト例と描画の様子を以下に示します。

スクリプト例では、１０区分の３次元メッシュに、(5.0,5.0,5.0)を中心にして球状にスカラー値を分布させています。また、y 方向のインデックス値を色分けのためのスカラー値として設定しています。値 0.1 で等値面を描き、色を([3]最小値色に対応する RGBA 値および最大値までの色相範囲を与える方法で)[0,0,1,1,-240]と指定しています。

```
meshf = meshfield("regular",[[0,10],[0,10],[0,10]],[10,10,10])
for i in range(0,11):
        for j in range(0,11):
                for k in range(0,11):
                        rr = (i-5.0)*(i-5.0) + (j-5.0)*(j-5.0) + (k-5.0)*(k-5.0)
                        if rr < 0.5: rr = 1.0
                        else:        rr = 1.0/rr
                        meshf.set([i,j,k],rr) # 面生成用
                        meshf.setSubValue([i,j,k],j) # 色分け用
meshf.isovalue(0.1,[0,0,1,1,-240]) # meshf.clevel(0.1,0.1,[0,0,1,1,-240])
meshf.draw(frame=[1,1,1,0.5])
```



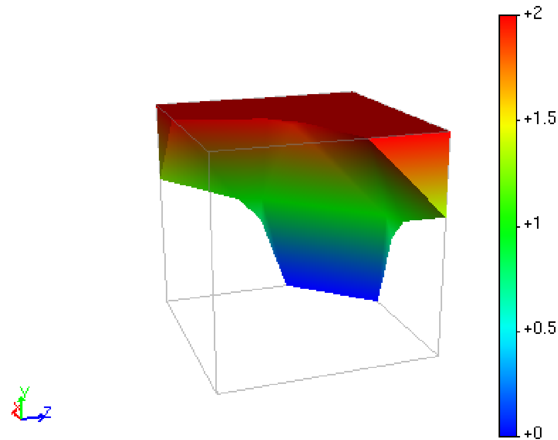
フタがある場合の等値面の色分け描画例を以下に２例示します。

（１）等値面を色分けするための補助値（SubValue）を設定して側面を色分けすることができます。

```
import math
meshf=_udf_.meshfield("regular",[[0,2],[0,2],[0,2]],[2,2,2])
for i in range(0,3):
        for j in range(0,3):
                for k in range(0,3):
```

```
                        rr = math.sqrt((i)*(i) + (j)*(j) + (k)*(k))
                        meshf.set([i,j,k],rr)
                        meshf.setSubValue([i,j,k],j)
meshf.isovalue(2.4,[0,0,1,1,-240])
meshf.draw(frame=[0.5,0.5,0.5,0.5],iso_side_surface=1)
```

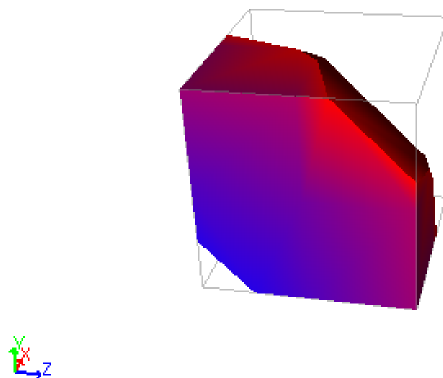
下図が描画例です。



（２）２枚の異なる色の等値面間を補間色で描画することができます。

```
import math
meshf=_udf_.meshfield("regular",[[0,2],[0,2],[0,2]],[2,2,2])
for i in range(0,3):
        for j in range(0,3):
                for k in range(0,3):
                        rr = math.sqrt((i)*(i) + (j)*(j) + (k)*(k))
                        meshf.set([i,j,k],rr)
                        #meshf.setSubValue([i,j,k],j)
meshf.isovalue(2.4,[1,0,0,1])
meshf.isovalue(0.5,[0,0,1,1])
meshf.draw(frame=[0.5,0.5,0.5,0.5],iso_side_surface=1)
```

下図が描画例です。



5.5　チューブ・リボンの描画のサンプル

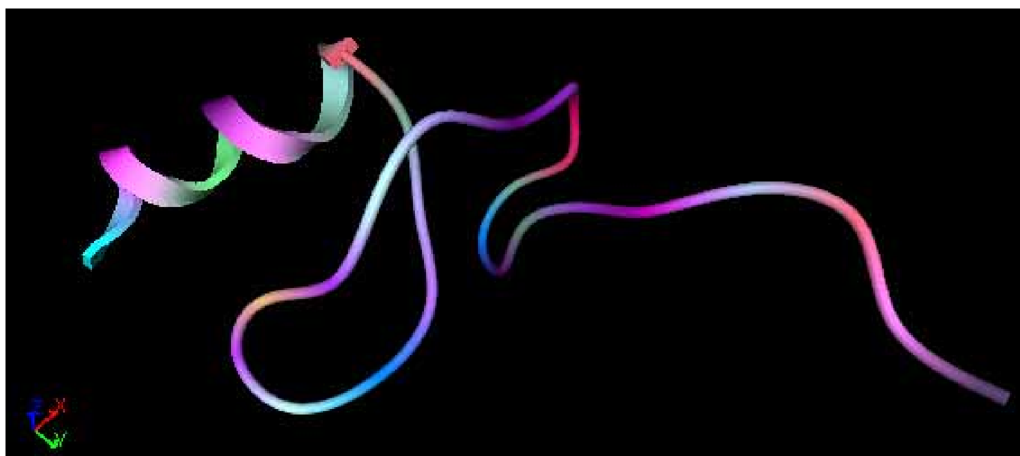
チューブおよびリボン描画機能を用いてタンパク質の一部を描画するサンプルを以下に示します。

```
# VECTOR PRODUCT
def vproduct(v1,v2):
        return
((v1[1]*v2[2]-v1[2]*v2[1]),(v1[2]*v2[0]-v1[0]*v2[2]),(v1[0]*v2[1]-v1[1]*v2[0]))

# RES COLOR
resname_color_map = {
        'ARG': (1.0,0.0,1.0,1.0),
        'LYS': (1.0,0.5,1.0,1.0),
        'ASP': (1.0,0.0,0.5,1.0),
        'ASN': (1.0,0.2,0.5,1.0),
        'GLU': (1.0,0.4,0.5,1.0),
        'GLN': (1.0,0.6,0.5,1.0),
        'HIS': (1.0,0.8,0.5,1.0),
        'PRO': (0.7,0.0,1.0,1.0),
        'TYR': (0.7,0.2,1.0,1.0),
        'TRP': (0.7,0.4,1.0,1.0),
        'SER': (0.7,0.6,1.0,1.0),
        'THR': (0.7,0.8,1.0,1.0),
        'GLY': (0.7,1.0,1.0,1.0),
        'ALA': (0.4,0.0,0.5,1.0),
        'MET': (0.4,0.3,0.5,1.0),
        'CYS': (0.4,0.6,0.5,1.0),
        'PHE': (0.4,0.9,0.5,1.0),
        'LEU': (0.0,0.0,1.0,1.0),
        'VAL': (0.0,0.5,1.0,1.0),
        'ILE': (0.0,1.0,1.0,1.0)
}
def rescolor(resname):
        return resname_color_map.get( resname, (1.0,1.0,1.0,1.0) )

# RES NAME
resn = [
 'MET','LYS','GLU','THR','ARG','TYR','CYS','ALA','VAL','CYS','ASN','ASP','TYR',
 'ALA','SER','GLY','TYR','HIS','TYR','GLY','VAL','TRP','SER','CYS','GLU','GLY',
 'CYS','LYS','ALA','PHE','PHE','LYS','ARG','SER','ILE']

# NITROGEN
nxyz = [
(50.465,24.781,79.46),(48.853,25.314,81.803),(50.068,23.740,84.563),
(49.281,22.007,86.487),(46.408,20.315,87.927),(43.711,21.102,90.213),
(41.636,20.324,93.000),(38.425,21.146,92.864),(36.971,21.463,95.158),
(38.462,21.244,97.573),(40.865,22.107,96.997),(43.026,20.444,96.907),
(45.515,18.476,95.117),(44.143,17.591,91.995),(42.724,14.579,90.828),
```

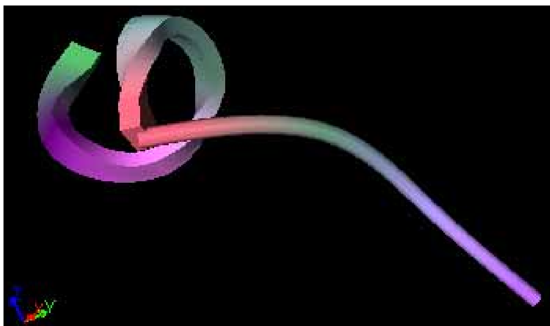
```
(41.373,13.380,88.812),(38.557,13.796,86.780),(35.813,14.054,88.313),
(32.501,13.464,88.981),(31.889,15.537,87.576),(33.683,17.724,88.014),
(36.987,18.895,88.297),(38.563,16.774,90.563),(39.812,15.644,93.892),
(40.842,12.426,95.123),(39.774,12.181,97.736),(37.556,14.179,97.847),
(35.908,13.021,95.922),(35.153,10.409,96.852),(34.020,11.149,99.269),
(31.716,12.632,98.173),(30.572,10.602,96.718),(29.778,8.693,98.869),
(28.005,10.386,100.095),(25.977,10.652,98.31) ]

# CARBON(CA)
caxyz = [
(50.332,26.116,80.055),(48.432,24.872,83.117),(50.966,22.643,84.886),
(48.572,21.082,87.348),(44.996,20.418,88.265),(43.354,21.165,91.617),
(40.334,19.913,93.447),(37.587,22.289,92.942),(36.372,21.538,96.486),
(39.330,21.496,98.68),(42.132,22.606,96.568),(44.192,19.607,96.943),
(45.768,18.002,93.738),(43.020,16.866,91.472),(43.215,13.289,90.344),
(40.695,13.084,87.598),(37.455,14.684,86.721),(34.794,13.232,88.914),
(31.232,14.051,89.264),(31.793,16.730,86.762),(34.578,18.800,88.314),
(38.212,18.275,88.62),(38.524,16.529,91.972),(40.817,14.800,94.516),
(40.172,11.171,95.502),(38.966,12.468,98.885),(36.212,14.711,97.572),
(35.135,12.100,95.101),(34.634,9.368,97.774),(33.069,11.87,100.054),
(30.477,12.843,97.492),(29.975,9.265,96.485),(29.035,8.235,100.024),
(26.908,11.206,100.46),(24.801,10.457,97.481) ]

# OXYGEN
oxyz = [
(50.739,26.439,82.436),(49.474,22.807,82.592),(50.480,20.344,85.438),
(46.697,22.492,87.419),(45.331,19.624,90.530),(41.096,21.162,90.971),
(39.847,22.059,94.229),(36.612,23.632,94.627),(37.128,22.882,98.195),
(41.523,22.149,99.148),(44.449,22.060,96.664),(43.534,19.416,94.630),
(44.022,16.440,93.829),(44.758,15.461,90.637),(43.109,12.176,88.240),
(39.588,15.038,88.258),(35.956,12.906,86.386),(33.717,15.126,89.746),
(30.156,16.166,88.906),(32.480,18.993,86.595),(35.952,17.014,88.876),
(37.765,18.830,90.916),(40.350,14.962,91.786),(38.874,13.421,94.776),
(38.159,10.923,96.845),(36.563,12.461,98.810),(34.074,13.808,96.915),
(33.157,10.846,95.836),(32.480,9.559,98.906),(30.649,11.636,99.806),
(28.628,11.441,97.412),(27.939,8.505,97.571),(26.810,8.698,100.762),
(24.613,11.125,100.023),(23.100,8.787,97.158) ]

# HELIX(start)
helix = 24

# TUBE
vertexlist=[]
nlen = len(caxyz)
for i in xrange(0,helix+2):
        vertexlist.append((caxyz[i],rescolor(resn[i])))
tube( vertexlist, radius=0.2, smooth="curve", hidden=[0,1])

# RIBON
```

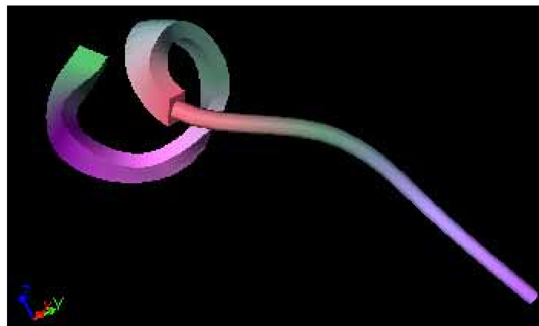
```
vertexlist=[]
for i in xrange(helix-1,nlen):
        vnca = (caxyz[i][0]-nxyz[i][0],caxyz[i][1]-nxyz[i][1],caxyz[i][2]-nxyz[i][2])
        vno = (oxyz[i][0]-nxyz[i][0],oxyz[i][1]-nxyz[i][1],oxyz[i][2]-nxyz[i][2])
        vnormal = vproduct(vnca,vno)
        vertexlist.append((caxyz[i],vnormal,rescolor(resn[i])))
ribbon(vertexlist, section=("rectangle",0.4,1.2), smooth="curve", hidden=[1,0])
vertexlist=0
```

上記スクリプトを実行した結果は下図のようになります。



リボンとチューブを滑らかにつながないと下図の左図のようになります。リボンおよびチューブを滑らかな曲線にするためにスプライン補間を使っていますので、スプライン補間のために端点を１頂点多く設定することにより右図のように滑らかにつながるようになります。追加した端点はスプライン補間計算のためにのみ使い、描画では隠します。





上図のスクリプトは、それぞれ以下のとおりです：

左図描画スクリプト

```
tube_vertexlist=[
 ((38.212, 18.275, 88.620), (0.7, 0.4, 1.0, 1.0)),
```

```
 ((38.524, 16.529, 91.972), (0.7, 0.6, 1.0, 1.0)),
 ((40.817, 14.800, 94.516), (0.4, 0.6, 0.5, 1.0)),
 ((40.172, 11.171, 95.502), (1.0, 0.4, 0.5, 1.0))
]
tube( tube_vertexlist, radius=0.2, smooth="curve", hidden=[0,0] )

ribon_vertexlist=[
 ((40.172, 11.171, 95.502),(-1.591, 0.137, -2.360), (1.0, 0.4, 0.5, 1.0)),
 ((38.966, 12.468, 98.885), (-0.013, -2.822, 0.695), (0.7, 1.0, 1.0, 1.0)),
 ((36.212, 14.711, 97.572), (-0.598, -0.295, 2.351), (0.4, 0.6, 0.5, 1.0)),
 ((35.135, 12.1, 95.101), (-1.706, 2.192, -0.852), (1.0, 0.5, 1.0, 1.0)),
 ((34.634, 9.368, 97.774), (-1.355, -1.398, -2.341), (0.4, 0.0, 0.5, 1.0)),
 ((33.069, 11.870, 100.054), (0.005, -2.136, 1.967), (0.4, 0.9, 0.5, 1.0))
]
ribbon(ribon_vertexlist, section=("rectangle",0.4,0.8), smooth="curve", hidden=[0,0])
```

右図描画スクリプト

```
tube_vertexlist=[ #(position),(rgba)
 ((38.212, 18.275, 88.620), (0.7, 0.4, 1.0, 1.0)),
 ((38.524, 16.529, 91.972), (0.7, 0.6, 1.0, 1.0)),
 ((40.817, 14.800, 94.516), (0.4, 0.6, 0.5, 1.0)),
 ((40.172, 11.171, 95.502), (1.0, 0.4, 0.5, 1.0)),
 ((38.966, 12.468, 98.885), (1.0, 1.0, 1.0, 1.0))
]
tube( tube_vertexlist, radius=0.2, smooth="curve", hidden=[0,1] )

ribon_vertexlist=[ #(position),(vector),(rgba)
 ((40.817, 14.800, 94.516), (-1.591, 0.137, -2.360), (1.0, 0.4, 0.5, 1.0)),
 ((40.172, 11.171, 95.502),(-1.591, 0.137, -2.360), (1.0, 0.4, 0.5, 1.0)),
 ((38.966, 12.468, 98.885), (-0.013, -2.822, 0.695), (0.7, 1.0, 1.0, 1.0)),
 ((36.212, 14.711, 97.572), (-0.598, -0.295, 2.351), (0.4, 0.6, 0.5, 1.0)),
 ((35.135, 12.1, 95.101), (-1.706, 2.192, -0.852), (1.0, 0.5, 1.0, 1.0)),
 ((34.634, 9.368, 97.774), (-1.355, -1.398, -2.341), (0.4, 0.0, 0.5, 1.0)),
 ((33.069, 11.870, 100.054), (0.005, -2.136, 1.967), (0.4, 0.9, 0.5, 1.0))
]
ribbon(ribon_vertexlist, section=("rectangle",0.4,0.8), smooth="curve", hidden=[1,0])
```

5.6　UDF データと描画オブジェクトの関連付けサンプル

UDF データと描画オブジェクトを関連付けることにより、描画オブジェクトをピッキングする（コントロール＋マウス左ボタン）ことにより、UDF データに対して設定されているアクションを実行できるようになります。

以下に UDF データと描画オブジェクトを関連付けるスクリプトを示します。

（現状では、メッシュフィールド描画オブジェクトとＵＤＦデータを関連付けることはできません）

setDrawRelation(udfpath_with_index) の引数 udfpath_with_index には UDF ロケーション（UDF データ名＋配列インデックス）を与える必要があります。

```
setDrawRelation('Set_of_Molecules.molecule[].atom[]',[0,0])
line([0,0,0],[0,1,0],1)
setDrawRelation('Set_of_Molecules.molecule[].atom[]',[0,1])
line([0,0,1],[0,1,1],2)
setDrawRelation('Set_of_Molecules.molecule[].atom[]',[0,2])
line([0,0,2],[0,1,2],3)
resetDrawRelation()

setDrawRelation('Set_of_Molecules.molecule[0]')
point([1,0,0],2)
polygon([[2,0,0],[3,0,0],[3,1,0]],3)
polyline([[4,0,0],[5,0,0],[5,1,0]],4)
disk([6,0,0],5)
ellipse1([7,0,0],6)
resetDrawRelation()

setDrawRelation('Set_of_Molecules')
cylinder([9,0,0],[9,1,0],8)
sphere([10,0,0],9)
ellipsoid1([11,0,0],10)
ellipsoid1([11,0,0],[0.0,0.0,1.0,1.0,0.1,0.05,0.05,1,0,0,0,1,0])
ellipsoid2([12,0,0],[12,1,0],11)
tetra([0,0,0],[-1,0,0],[0,-1,0],[0,0,-1],12)
arrow([13,0,0],[13,1,0],13)
cube([14,0,0],1,14)
text([15,0,0],"ABC",15)
resetDrawRelation()
```

付録 A　OCTA2007 リリースにおける新機能・変更点の一覧

OCTA2005 リリース以降に追加された機能および変更点を以下にまとめています。詳細は本編の説明を参照してください。

A.1 GOURMET 2007 の新機能および変更点

A.1.1 GOURMET2007 の新機能

・UDF ファイルの任意の位置にレコードを挿入する機能
newRecord(reclabel="",record_pos=-999,record_num=1)
指定レコード record_pos の前に record_num 個のレコードを追加します。

・UDF ファイルの任意レコード範囲を削除する機能
eraseRecord(record_pos=-999,record_num=-999)
現在レコードまたは指定レコード record_pos 以降を record_num 個削除します。

・スクリプトの履歴制御
EraseFromHistory()
この命令を含むスクリプト文全体を実行履歴に残さないようにします。

・チューブ・リボンの描画
異なるチューブ・リボンオブジェクトを滑らかにつなぐために、隠し頂点を両端に設定することができます。

・非構造格子の meshfield において、要素タイプの混在を可能にしました。
element_type：要素タイプの指定において下記の"2d"および"3d"を追加しました。
三角形："triangle"
四角形："quad"
四面体："tetra"
六面体："hexa"
２次元三角形要素・四角形要素混在："2d"
３次元四面体要素・六面体要素混在："3d"
要素形状が混在している時は、要素の節点数から要素形状を決定します。

A.1.2 その他の変更点

・UDFManager 以外の Python スクリプトモジュールの解説を本ドキュメントに
「第４章　ユーティリティモジュール」として追記しました。

・チューブ・リボンオブジェクトを滑らかにつなぐための描画サンプルを追記しました。

A.2 GOURMET 2006 の新機能および変更点

A.2.1 GOURMET2006 の新機能

（１）等値面の描画に関する機能追加
・構造メッシュからの等値面にフタをする機能

```
meshfield.draw( frame = -9999, arrow = -9999, subdivision = 2, iso_side_surface = 1)
```

iso_side_surface：等値面描画時に側面処理ありおよび側面処理なしを設定する。
０：側面処理なし、１：側面処理あり

・非構造メッシュ（四面体および六面体メッシュ）から等値面を描画する機能
六面体からの等値面生成例

```
# UNSTRUCTURED HEXA.
mlist=[[1,[0,0,0]],[2,[1,0,0]],[3,[1,1,0]],[4,[0,1,0]],[5,[0,0,1]],[6,[1,0,1]],[7,[1,1,1]],[8,[0,1,1]]]
elelist=[[1,2,3,4,5,6,7,8]]
mf=meshfield("hexa",mlist,elelist)
mf.field([1,2,3,4,5,6,7,8],[1,2,3,4,5,6,7,8])
mf.subfield([1,2,3,4,5,6,7,8],[0,0,0,0,1,1,1,1])
mf.isovalue(6.5,[1,0,0,1,0,1,0,1])
mf.draw(frame=[0.5,0.5,0.5,0.5])
```

四面体からの等値面生成例

```
# UNSTRUCTURED HEXA.
mlist=[[1,[0,0,0]],[2,[1,0,0]],[3,[1,1,0]],[4,[0,1,0]],[5,[0,0,1]],[6,[1,0,1]],[7,[1,1,1]],[8,[0,1,1]]]
elelist=[[1,3,4,8],[1,3,8,7],[1,5,7,8],[1,7,2,3],[1,2,7,5],[6,7,2,5]]
mf=meshfield("tetra",mlist,elelist)
mf.field([1,2,3,4,5,6,7,8],[1,2,3,4,5,6,7,8])
mf.subfield([1,2,3,4,5,6,7,8],[0,0,0,0,1,1,1,1])
mf.isovalue(6.5,[1,0,0,1,0,1,0,1])
mf.draw(frame=[0.5,0.5,0.5,0.5])
```

（２）Python スクリプト実行における機能追加
・複数の異なる UDF ファイルを$$で指定する（$$拡張）機能
・指定した UDF レコードのデータを定義部付で書き出す機能（write メソッド）
・描画対象の３次元移動（平行移動・回転移動）機能（アクション）

A.3 GOURMET2005 の新機能および変更点（履歴）

A.3.1 GOURMET2005 の新機能

UDFManager クラスに以下のメソッドを追加しました。詳細は本編の説明を参照してください。

・ＵＤＦ定義の付加的情報を返します。
　class 付定義かどうか、ヘルプ説明文および select 型の選択項目リストを返します。
　queryDefine(udfpath,key)

・グローバル定義かどうかを返します。
　isGlobalDef(udfname)

・データ取得対象レコードの切り替えおよび取得を行います。
　getReferRecord()
　setReferRecord(rec_ref)

・単位関連の追加機能：ＵＤＦデータに指定された単位と交換可能な登録済単位のリストを返します。
　unitList(udf_path)

Python 描画関数に以下の機能を追加しました。

・チューブ・リボンの描画
　tube(vertexlist,radius=1.0,smooth="fit",arrow=None)
　チューブ（円断面）を描画します。

　ribbon(vertexlist,section=("rectangle",width,height),smooth="fit",arrow=None)
　リボン（楕円または矩形断面）を描画します。

・イメージファイル(jpeg)の表示
```
imageRectangle(imageFile,ratio=1.0,origin=[0,0,0],
               widthVector=[1,0,0],heightVector=[0,1,0])
```
　イメージファイルを元の縦横サイズで矩形に表示します。

　imageBackground(imageFile,spread=1,ratio=1.0)
　ビューのバックグラウンドにイメージを表示します。

・任意の描画対象と UDF データを関連付けるます。（ピッキング対象の拡大）
　setDrawRelation(udfpath_with_index)
　resetDrawRelation()

メッシュフィールドクラスに以下の機能を追加しました。

・等値面をコンター表示するためのスカラー値を設定します。
　mesh_obj_in_python.setSubValue(index_list,value)

setSubValue(...)で設定した値を取り出します。
　mesh_obj_in_python.getSubValue(index_list)

・ボリュームレンダリングを実行します。
　drawVolume(colors = (1.0,1.0,0.0,1.0),intensity = 0.5,slope_factor = 2)

・等值面の描くためのしきい値を指定します。
```
isovalue( value, draw_attr )
```

A.3.2 GOURMET2005 の変更点

メッシュフィールドクラスの下記機能を変更しました。

・コンターおよび等値面を描画するときに、色相範囲で描画色を指定する機能を追加しました。

・コンター描画の細分化レベルを指定する引数を draw()メソッドに追加しました。
```
mesh_obj_in_python.draw( frame = -9999,arrow = -9999,subdivision = 2 )
```

A.4 GOURMET2003 の新機能および変更点（履歴）

A.4.1 GOURMET2003 の新機能

UDFManager クラスに以下のメソッドを追加しました。詳細は本編の説明を参照してください。

・UDFファイルのパスおよびディレクトリパスを返す
```
udfFile()
udfDirectory()
```

GOURMETからのみ使用できるPython関数として以下が追加されています。

・新たなUDFファイルを読み込む
```
openUDF(udf_file_path)
```

・視線方向を変える
```
viewDirection(eyes,head)
viewDefaultDirection(eyes,head)
```

・点・線のポイント数を指定して描画する
下記の描画関数の属性pointにポイント数（ドット数）を指定します。ドット数を省略するとデフォルト値が使用されます。（デフォルト値は点が３、線が１）
```
point(xyz,[r,g,b,a,point])
line(xyz1,xyz2,[r,g,b,a,point])
polyline(xyzlist,[r,g,b,a,point])
```

・アニメーション（連続描画）のコマ送り調整
```
drawInterval(millisecond)
```
[Start]ボタンでPythonを連続実行する時のレコード移動待ち時間をミリ秒で与えます。
```
drawSkip(skip)
```
[Start]ボタンでPythonを連続実行する時のレコードスキップ数を与えます。

・レコード削除
```
eraseRecord()
```

現在位置のレコード以降が削除されます。削除後の現在レコードは１つ前のレコードになります。

eraseRecord(record_no)

指定レコード以降が削除されます。現在レコードが削除対象でないときは、現在レコード位置は変化しません。現在レコードが削除される時は、削除後の現在レコードは最後レコードになります。

A.4.2 GOURMET2003の変更点

・UDFManagerのput()およびget()のインデックス指定方法
UDFManagerのput()およびget()でインデックス指定が多すぎる時､エラーとなっていましたが有効なインデックス数のみを利用して残りは無視するようにしました。
（例） udf.get('abc[].x[]',[1,2,3,4,5])
としても有効なインデックス[1,2]のみを利用して[3,4,5]は無視します。

・GOURMETにおけるPython実行に関して、以下の変更があります。
(1) UDFファイルを開かずにPythonを実行できます。

(2) UDFファイルと同じディレクトリにあるPythonスクリプトファイルをインポートして実行できます。
すなわちUDFファイルがあるディレクトリがPYTHONPATHに追加されます。
UDFファイルを開かずにPythonを実行する場合は起動ディレクトリがテンポラリディレクトリとなります。
同名のPythonスクリプトがある場合の優先順位は以下のようになります。
$PF_FILES/python
$PF_FILES/lib/$PF_ARCH
$PF_ENGINE/python
$PF_ENGINE/lib/$PF_ARCH
$PYTHONHOME/Lib 以下
起動ディレクトリ

(3) Python実行時のテンポラリファイル作成ディレクトリを起動ディレクトリにしました。
今までは、関数tmpnamを使っていた為、UNIX/LINUXでは/var/tmp、Windowsではドライブの先頭となっていました。
関数tmpnamを使うと、LINUXではセキュリティに関する警告が表示され、Windowsではドライブの先頭が書き込み不可の場合があるなど不具合が発生する危険性がありました。

(4) Pyhton実行キャンセル機能
Python実行中に処理をキャンセルできます。ただし、Python実行を途中でキャンセルした場合、Pythonによるデータ変更処理が途中で終わっている為に、データに不整合が生じている危険性があります。Pyhton実行をキャンセルした場合は、対象のデータを破棄するようにしてください。
またPython実行がキャンセルされた時は、処理メッセージの表示は最後の４Kバイトのみが出力されるようになっています。

付録 B. 解決されたバグ一覧

B.1 OCTA2006 で解決されたバグ

・構造メッシュ用meshfieldで、Rectangular メッシュが描画できない不具合を修正しました。

B.2 OCTA2006 で解決されたバグ

・GOURMET_2005での描画実装部分の改造以降、非構造メッシュからのコンターが描画できなかった不具合を修正しました。
・Python実行時にオブジェクト参照が残っている個所があったため、メモリーが消費される不具合を修正しました。
・common_start() および common_end() が効かなかった不具合を修正しました。
・Meshfieldで構造メッシュのcrange()は効くが、非構造メッシュのcrange()が効かない不具合を修正しました。

B.3 OCTA2005 で解決されたバグ

・line($position[0],$position[1],1) など"$変数"を使用して描画することはできていましたが、間違った使用方法を実行すると異常終了する場合がある不具合を修正しました。
例えば、
line([$position[0],$position[1]],1)
なととすると、異常終了していました。

B.4 OCTA2003 で解決されたバグ

・描画関数の引数に配列ではない$変数を使用すると引数が渡っていなかったバグを修正しました。
（例） point($pos,2)
・GOURMETでUDFManagerのget,putを用いて単位を持つ数値データを取得する時、単位が変更されているとその単位に変換された値が返されていました。get,putメソッド自体に単位を指定しない限り単位変換は行われないようにしました。
・ファイルパスに日本語が混じっている時、udfFile()/udfDirectory()の戻り値を GOURMET でプリントアウトすると文字化けする不具合を修正しました。
・GraphSheet で列が無いのに行を挿入しようとすると異常終了する不具合を修正しました。
・polyline 描画関数で 32 個以上の頂点を与えると異常終了する不具合を修正しました。
・Location クラスで４次元以上の配列を扱うと異常終了する不具合を修正しました。