# ＯＣＴＡ

ソフトマテリアルのための統合化シミュレータ

# ＵＤＦ文法
# リファレンスマニュアル

ＯＣＴＡユーザーズグループ

APR. 2 2005

## 執筆者

西本正博

## プログラム開発者

プログラム設計　芝野真次，西尾裕三
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，西康之

## 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。

# 目次

# 表目次

# 第１章　ＵＤＦ文法

## 1.1 はじめに

本書ではUDFデータ書式の基本文法および規約を説明します。UDFは科学技術計算で用いられるすべてのデータの記述に用いることができ、この書式で記述されたデータはOCTAシステムを用いてどのような構造のデータでも読み書きできます。

## 1.2 UDF の構造

UDFは、ヘッダー部、単位定義部、データ名定義部及びデータ実体部からなります。ヘッダー部はデータファイル自体の説明をするための情報を一定の書式で記述する部分です。単位定義部は実数データに付ける新たな単位をＳＩ単位系から定義する部分です。データ名定義部にはデータ実体部で使用するすべてのデータの名称と型宣言を記述します。データ実体部はデータ名定義部で宣言された変数名に数値や文字のデータを割り当てます。

## 1.3 ヘッダー部

ヘッダー部はデータファイルに関する情報を記録しておく部分です。データファイルに関する情報とは、ＵＤＦデータファイルが使用される解析エンジンモジュールの名称やバージョン番号、入出力タイプ、コメントなどです。
UDFデータファイルにはヘッダー部が無くてもファイルの読み込みや書き込みはできるようになっています。しかし、データファイルにどのようなデータがあるのかを記録しておくことは重要です。[1]

ヘッダー部の書式は以下のとおりです。\begin{header}～\end{header}の間にヘッダー情報を記述します。\begin{def}～\end{def}はヘッダー部のデータ定義です。\begin{data}～\end{data}にはデータ内容を記述します。

```
\begin{header}
\begin{def}
```

[1] GOURMET 及び UDFManager for Python では、ヘッダー部の情報を参照・編集する機能があります。

```
EngineType:string;
EngineVersion:string;
IOType:string;
ProjectName:string;
Comment:string;
Action:string;
\end{def}
\begin{data}
EngineType:"EngineTypeName"
EngineVersion:"EngineVersionName"
IOType:"IN or OUT"
ProjectName:"ProjectNameString"
Comment:"CommentString"
Action:"ActionFile1;ActionFile2;ActionFile3"
\end{data}
\end{header}
```

EngineTypeNameおよびEngineVersionNameは、解析エンジンの名称および開発バージョン番号です。

IOTypeには入力用あるいは出力用データかによって、INまたはOUTを記述します。

ProjectNameStringおよびCommentには、解析名称およびコメントを記述します。

Actionには、GOURMETを使用している時に実行するアクションファイルの名称を記述します。

ヘッダー部の定義部分（\begin{def}～\end{def}）とデータ部分（\begin{data}～\end{data}）は、分割して別のヘッダーブロック（\begin{header}～\end{header}）に記述することができます。例えば、定義部分のみを後述のインクルードファイル内に記述しておくことができます。

## 1.4 単位定義部

ＵＤＦでは実数データに単位を指定しておくことにより、エンジン間でデータのやり取りを行う際やGOURMETでデータの参照・編集を行う時に単位変換機能を利用できます。データ毎の単位指定は後述のデータ名定義部で行います。単位定義部では各データに指定する単位をSI基本単位から構成し、UDFデータに指定する新しい単位を定義しておきます。

新しい単位名に使用できる文字は大小アルファベット、_(下線)及び２文字目以降に数字(0-9)を使用することができます。アルファベットの大小の違いは異なる文字とみなされます。

データ名定義部で使用する単位が基本単位あるいは１０の何乗かを示すための接頭語をつけた基本単位からの乗除指数演算のみで表現できる時は、単位定義部で定義しておく必要はありません。

新たな単位を構成するための出発点となるSI基本単位は以下の通りです。

[kg] (kilogram)

[m] (meter)
[s] (second)
[A] (ampere)
[K] (kelvin)
[mol] (mole)
[cd] (candela)
[rad] (radian)
[sr] (steradian)

単位の接頭語を表1.1 に示します。

| 接頭語(読み) | 倍率 |
|---|---|
| Y (yotta) | $10^{24}$ |
| Z (zetta) | $10^{21}$ |
| E (exa) | $10^{18}$ |
| P (peta) | $10^{15}$ |
| T (tera) | $10^{12}$ |
| G (giga) | $10^{9}$ |
| M (mega) | $10^{6}$ |
| k (kilo) | $10^{3}$ |
| h (hecto) | $10^{2}$ |
| da (deka) | 10 |
| d (deci) | $10^{-1}$ |
| c (centi) | $10^{-2}$ |
| m (milli) | $10^{-3}$ |
| mc (micro) | $10^{-6}$ |
| n (nano) | $10^{-9}$ |
| p (pico) | $10^{-12}$ |
| f (femto) | $10^{-15}$ |
| a (atto) | $10^{-18}$ |
| z (zepto) | $10^{-21}$ |
| y (yocto) | $10^{-24}$ |

表1.1 単位の接頭語

単位定義部は、\begin{unit}～\end{unit}の間に次のように記述します。１つの単位定義は１行で記述します。

\begin{unit}
*New_Constant = CONSTANT*
*[New_Unit] = [Unit_Expression]*
*[New_Unit] = CONSTANT[Unit_Expression]*
\end{unit}

*New_Constant* は、単位定義部で使用する定数値です。

*CONSTANT* には、数値および数値型のUDFデータ名を使用することができます。UDFデータ名を使用するときには、次のようにUDFデータ名の頭に$を付けて{}で囲みます。

*[New_Unit] = {$UDF_Data}[Unit_Expression]*

*Unit_Expression* には、SI基本単位あるいはここで使用するより前の行で定義された単位の乗除指数演算式を記述します。演算子は、乗算、除算および指数演算に対してそれぞれ、* / ^ を用います。

単位定義部の記述例を示します。

```
\begin{unit}
PI=3.141592654
[N] = [m*kg/s^2]
[J] = [N*m]
[R] = 8.314472[J/(K*mol)]
[amu]=1.66054e-27[kg]
[degree]=PI/180[rad]
[cal] = 4.1855[J]
[epsilon] = 0.5 [kJ/mol]
[Temp] = [epsilon/R]
[Mass] = 23.3e-27[kg]
[Len] = 0.38[nm]
[Me] = {$Me_val}[g/mol]
[sigma]={$Reduced_Parameters.Length}[nm]
\end{unit}
```

## 1.5 データ名定義部

データ名定義部にはデータ実体部で使用するすべてのデータの名前および型を記述します。ま

たデータの説明文や単位を記述します。
まずデータ名定義部であることを宣言するために以下のキーワードを用います。

・\def
　１行でデータの定義が完了する場合のキーワード
・\begin{def}
　データ名定義を複数の行で行う場合の開始キーワード
・\end{def}
　データ名定義を複数の行で行う場合の終了キーワード
・\begin{global_def}
　どのレコードにいても共通のデータを参照・編集できるグローバルなデータ名定義を行う場合の開始キーワード。
　データ実体は、コモン部に記述する。レコード部にあると無視される。
・\end{global_def}
　グローバルなデータ名定義を行う場合の終了キーワード

### 1.5.1 基本データ型

UDFで使用する基本データ型は以下のとおりです。

・int
　データが整数であることを宣言する場合に用います。
・long
　データが長整数であることを宣言する場合に用います。
・short
　データが短整数であることを宣言する場合に用います。
・arraysize
　配列サイズを指定する整数型であることを示します。
・float
　データが短精度実数であることを宣言する場合に用います。(singleと同じ)
・double
　データが倍精度実数であることを宣言する場合に用います。
・single
　データが短精度実数であることを宣言する場合に用います。(float と同じ)
・string

データが文字型であることを宣言する場合に用います。

・select{"選択文字列１","選択文字列２"}

データがセレクト型であることを宣言する場合に用います。値は文字列です。[2]

・KEY

データがユーザー定義型の中に定義されていて、ユーザー定義型として定義された複合データ配列を指示するためのユニークな文字列型データであることを宣言する場合に用います。

・<ユーザー定義型,KEY>

データが複合型データ内のKEY型データを参照していることを宣言します。

・ID

データがユーザー定義型の中に定義されていて、ユーザー定義型として定義された複合型データを指示するためのユニークな整数型データであることを宣言する場合に用います。

・<ユーザー定義型,ID>

データが複合型データ内のID型データを参照していることを宣言します。

## 1.5.2 区切り記号

複合型データ・配列などの定義区切りを示すために以下の区切り記号を用います。

・{}

データが複合型データであることを宣言する場合に複合型データの定義範囲を明示する場合に用います。

・[]

データ名が配列である事を宣言する場合に用います。またデータ名の単位を指定する場合に用います。

・;

データ名定義の終了を明示する場合に用います。（省略可能）

・:

データ名とデータ型の間に区切り子として用います。

・"ダブルクォートで挟まれた文字列"

データ名の説明文を記述します。

[2] セレクト型の宣言をした変数は、GOURMET の入力支援サービスを受けることができる。GOURMET の UDF 編集画面でデータを入力する時、"選択文字列１","選択文字列２"等が選択リストに表示されるようになり、リストの中から選択するだけでデータ入力をすることができる。GOURMET に格納されるデータは文字列型と同じである。

### 1.5.3 データ名定義宣言

実際にデータを定義する方法について説明します。データを定義する方法としては、単純に一つのスカラーデータを宣言する、配列として宣言する、複合型データとして宣言する、ユーザー定義型から複合型データを定義するなどの方法があります。

**・スカラー及び配列データのデータ名定義**

スカラーデータ名の定義は以下の形式で行います。配列データ名の定義は、データ名称の最後に[] を１つ以上付けます。

*|begin{def}*
*Variable_Name1:Type1*
*Variable_Name2:Type2;*
*Array_Name1[]:Type1*
*Array_Name2[]:Type2;*
*Matrix_Name1[][] :Type1*
*Matrix_Name2[][ArraySize1][ArraySize2] :Type2;*
*|end{def}*

複数行のデータ名定義部は*|begin{def}*で始まり*|end{def}*で終了します。
*Variable_Name*および*Array_NameN* はデータ名であり、*TypeN* はデータの型です。最後の";"は省略可能です。
１次元配列データのサイズはデータ実体部に記述されるデータ数です。多次元配列（[] の数が２以上）の配列サイズは各次元でデータ実体部に記述されるデータ数の最大値となります。[3]
ArraySizeN は多次元配列の固定サイズを指定するための数値です。下記のように任意の位置に固定サイズを指定することができます。
（例）
marray1[10][10][]:int
marray2[][][3]:double

１行で定義の記述ができる場合は以下のように|def文を用いてデータ定義を行えます。

|def *Variable_Name*:*Type*;

[3] エンジンライブラリおよび UDFManager(C++/Python) のサイズ取得関数を使用した場合のサイズです。

（例）

```
\def temperature:double;
\begin{def}
Steps:int
Time:double
Nscf[]:int
\end{def}
```

### ・複合型データのデータ名定義

複合型データのデータ名定義は以下のように{}を用いてデータのグループ化を行います。

```
|begin{def}
Variable_Name1:{
  Child_Name1:Type1
  Child_Name2[]:Type2
  Child_Name3[][]:Type3
}
Variable_Name2[]:{
  Child_Name1:Type1
  Child_Name2[]:Type2
  Child_Name3[][]:Type3
}
Variable_Name3[][]:{
  Child_Name1:Type1
  Child_Name2[]:Type2
  Child_Name3[][]:Type3
}
|end{def}
```

同様な形式を用いることにより、{} を用いて任意の深さの複合データを作ることができます。

（例）

```
\begin{def}
  config:{
    time:double,
    atom[]:{
```

```
      pos[]:float,
      vel:{x:int,y:int,z:int},
      id:int
    }
    kinetic_energy:int
  }
};
\end{def}
```

### 1.5.4 サイズ指定をもつ配列定義

配列データ（１次元配列および多次元配列）を定義する時に、以下のように配列のサイズを指定することができる。*Array_SizeN* は多次元配列のサイズを指定するための数値または他のデータ名です。

```
Variable_Name1:{
  Child_Name1[ArraySize1]:Type1
  Child_Name2[ArraySize2][]:Type2
  Child_Name3[ArraySize3][ ArraySize4]:Type3
}
Variable_Name2[ArraySize1]:{
  Child_Name1[ArraySize2]:Type1
  Child_Name2[][ArraySize3]:Type2
  Child_Name3[ArraySize4][ArraySize5]:Type3
}
Variable_Name4[ArraySize1][ArraySize2]:{
  Child_Name1[ArraySize3]:Type1
  Child_Name2[][ArraySize4]:Type2
  Child_Name3[ArraySize5][ArraySize6]:Type3
}
```

配列サイズを他のデータ名により指定する場合に、サイズを指定するデータは arraysize 型で定義されていなければなりません。またサイズ指定には配列指示子である[]を含めたデータ名を使うことはできません。ただし、サイズ指定データとサイズ指定される配列が同じ構造体内にある場合は、下記例のようにサイズ指定データ名を相対的に指定することができます。

```
\begin{def}
m0:arraysize           // (0)
n0:arraysize           // (0)
Root:{
  child[]:{
    m0:arraysize
    n1:arraysize
    aa[][]:int         // (1)
    bb[10][20]:double  // (2)
    cc[][10]:double    // (3)
    dd[.m0][n1]:double          // (4)
    ee[..mn.m2][..mn.n2]:double // (5)
    ff[m0][n0]:double           // (6)
  }
  mn:{
    m2:arraysize               // (7)
    n2:arraysize
  }
}
\end{def}
```

(1)のように配列の中に何も書かずに配列を定義すると､配列サイズはデータ数によって決まります｡サイズ指定のない多次元配列の場合は､配列サイズは各次元でデータ実体部に記述されるデータ数の最大値となります。

(2)および(3)は配列サイズを固定サイズで定義しています。配列サイズが一部空([])の場合は(1)と同様にデータ数によってサイズが決まります。

(4)(5)(6)は配列サイズを他の変数により指定しています。

(4)および(5)は定義する配列を起点としてサイズ指定に用いる変数を相対的に指定しています。サイズを相対的に指定するためには、サイズ指定される配列の位置から見て、サイズ指定データが構造体内の同じ並びにある場合、点（ピリオド(.)）を１つ入れてサイズ指定データ名を記述します。

サイズ指定される配列の位置から見て、サイズ指定データが１つ上位の構造体にある場合、点を２つ(..)入れてサイズ指定データ名を記述します｡同様に相対位置を１つ上位に上げていく度に点(.)を１つづつ増やしてサイズ指定変数を記述します。

サイズ指定変数を絶対指定できる場合(後述の(6)参照)は、点(.)を記述しません。サイズ指定変数の絶対指定と点（ピリオド(.)）１つを含むデータ名指定が混同しない場合には、点（ピリオド(.)）１つを省略することができます。（上記例(4)）

(4)は dd[][]と同じ並びに定義されている arraysize 型変数をサイズ指定に使用しています。

(5)は ee[][]の１つ上位の構造体を起点にしてサイズ指定変数名を記述しています。

(6)はサイズ指定に用いる変数を絶対位置で指定しています。((0)の変数をサイズ指定に使っています。)

(7)サイズを指定するための変数はサイズ指定として使用される前に定義しておく必要はありません。全てのデータ名定義終了までに定義されていれば良いのです。

## 1.5.5 ユーザー定義型

複合型データはいくつかのデータをひとまとめにして扱うための仕組みであり、ＵＤＦでは一度に多階層の複合型データを定義することができます。

ユーザー定義型は複合型データ型の定義を新たなデータ型として扱ったものです。

ユーザー定義型として宣言された変数は、ユーザー定義型が持つ構成要素（変数）をメンバーに持つ複合型データ変数となります。なお、配列型の複合型データをユーザー定義型として扱うことはできません。

複合型データの宣言をユーザー定義型だけにしか使わないことを明示するために下記の予約語を用います。

・class

純粋なユーザー定義型の宣言であることを示します。

（GOURMETの編集画面で定義情報の表示を制御するために用いられます。UDF定義にclass接頭語を付けると、そのデータ定義はデータの編集対象とならないため、GOURMETのUDF編集画面に表示されません。）

下記の例ではVector3d及びAtomをユーザー定義型として使用しています。

```
\begin{def}
class Vector3d:{x:float, y:float, z:float}; // 座標値のための複合データ型
Atom:{
  molID:int,
  typeName:string,
  coord:Vector3d, // 複合データ型をユーザー定義型として使用
  vel:Vector3d,   // 複合データ型をユーザー定義型として使用
  force:Vector3d, // 複合データ型をユーザー定義型として使用
  totalID:int
};
\end{def}
\begin{def}
Molecule:{
  name:string,
  atom[]:Atom // 複合データ型をユーザー定義型として使用
};
```

```
\end{def}
```

上記のユーザー定義型を使って記述したデータ名定義を、複合型定義のみを使って書き下した場合、以下のようになります。

```
\begin{def}
Molecule:{
  name:string,
  atom[]:{
    molID:int,
    typeName:string,
    coord:{x:float, y:float, z:float}
    vel:{x:float, y:float, z:float}
    force:{x:float, y:float, z:float}
    totalID:int
  }
};
\end{def}
```

## 1.5.6 KEY 及び ID 型のデータ名定義

KEY及びID型のデータ名定義は以下のようにします。

```
class Class_name1:{
  Id_Variable:ID
  Other_Variable ...
};
class Class_name2:{
  Key_Variable:KEY
  Other_Variable ...
};
```

KEY及びID型データを参照する変数は、それぞれ*<複合型名,KEY>*及び*<複合型名,ID>*を用いて定義します。

```
class Other_Class:{
  Reference_id[]:<Class_name1,ID>
  Reference_key[]:<Class_name2,KEY>
```

```
  Other Variable ...
};
```

一例を示すと以下のようになります。

```
\begin{def}
class Vector3D:{ // 3D ベクトル型
  x:float // X 座標
  y:float // Y 座標
  z:float // Z 座標
};
class Vertex: { // 点のクラス
  id:ID // Vertex クラスデータを指すID
  position:Vector3D // 座標
};
class Edge: { // 辺のクラス
  id:ID // Edge クラスデータを指すID
  vertex[]:<Vertex,ID> // 点のID のリスト
};
class Face: { // 面のクラス
  id:ID // Face クラスデータを指すID
  vertex[]:<Vertex,ID> // 点のID のリスト
};
class Cell: { // 体積要素のクラス
  id:ID // Cell クラスデータを指すID
  vertex[]:<Vertex,ID> // 点のID のリスト
};
class Neighbor:{ // 近傍の要素配列型
  vertex[]:<Vertex,ID> // 点のID のリスト
  edge[]:<Edge,ID> // 辺のID のリスト
  face[]:<Face,ID> // 面のID のリスト
  cell[]:<Cell,ID> // 体積要素のID のリスト
};
class Mesh:{ // メッシュ型
  name:KEY // メッシュ名をKEY とする
  type:string // タイプ
  axes[]:MeshAxis // メッシュ軸の配列
  periodic[]:int // メッシュ軸の周期境界条件の配列
};
class MeshData:{
  name:<Mesh,KEY>
  vertex[]:Vertex // メッシュ点の位置ベクトルの配列
  edge[]:Edge // 辺要素のデータ配列
  face[]:Face // 面要素のデータ配列
  cell[]:Cell // 体積要素のデータ配列
  neighbor_of_a_vertex[]:Neighbor //点要素の近傍の要素配列
```

```
  neighbor_of_a_edge[]:Neighbor //辺要素の近傍の要素配列
  neighbor_of_a_face[]:Neighbor //面要素の近傍の要素配列
  neighbor_of_a_cell[]:Neighbor //体積要素の近傍の要素配列
};
\end{def}
```

### 1.5.7 単位の指定

float型およびdouble型のデータには単位を指定することができます。スカラーデータおよび配列データへの単位の指定は、以下のように定義文に続けて単位を[]で括って行記述します。

*|begin{def}*
*Variable_Name:Type**[unitname]***
*Array_Name[]:Type**[unitname]***
*|end{def}*

ユーザー定義型を使用している場合の単位指定は、ユーザー定義型に仮単位を指定しておき、使用する単位を後で指定することができます。単位を後で指定する時、実際には単位を指定する必要がない時、空の[]を指定します。

単位指定の例を以下に示します。

```
\begin{def}
class Vector3d:{x:float[unit], y:float[unit], z:float[unit]}[unit]
class Cell:{a:float[unit1], b:float[unit1], c:float[unit1],
            alpha:float[unit2], beta:float[unit2], gamma:float[unit2]}[unit1][unit2]
class Output:{
    Steps:int
    Num_of_Grid:Vector3d []
    Radius:float [sigma]
    grid:Vector3d [unit]
    cell[]:Cell [sigma][degree]
}[unit]
OutData:Output[Len]
\end{def}
```

### 1.5.8 ヘルプ

全ての単純データ、配列データ、ユーザー定義型および構造体型に説明文を付けておくことができます。

説明文をデータ定義に指定する形式は、定義式の最後に説明文をダブルクオート（""）で囲んで指定します。ダブルクオートで囲んだ説明文は、任意の回数続けて記述することができます。
説明文中の***改行***、***ダブルクオート***、***|マーク***は、それぞれ*|n*、*|"*、|| を用います。

```
class Angle:{
  theta0:float[degree] "ExternalAngle-\"theta0\""
  K:float
  Min_Position:Vector3d[Len]
  "ExternalAngle-in_Position with unit.\n"
  "ExternalAngle-in_Position help-line-2."
  Max_Position:Vector3d[Len]
  Acceleration[]:Vector3d[m/s^2]
} "Angle class"
angle:Angle "ExternalAngle"
```

## 1.6 データ実体部

データ実体部では、データ名定義部で宣言された変数に実際にデータを割り当てます。データ実体部には数値データまたは文字列データを空白, 改行, ;, {}, [] などの区切り記号で区切ります。文字列データは引用符( "または' ）で括ります。
データ実体部にはコモンデータ部及びデータレコード部の種別があります。コモンデータは解析ステップが異なっても変化しない変数（すなわちコンスタント）です。データ名定義部とコモンデータ部は、順序が混在していてもかまいませんが, データ名定義がデータ実体に先行して記述されている必要はあります。
データレコード部は、解析のタイムステップ毎に変化するデータを記述するために用い、*|begin{record}* および *|end{record}* で囲むことによって明示します。
データ実体部であることを宣言するために以下の予約語を用いる。

・\data
　１行でデータの割り当てが完了する場合に用います。
・\begin{data}
　データの割り当てを複数の行で行う場合の開始キーワードです。
・\end{data}
　データの割り当てを複数の行で行う場合の終了キーワードです。
・\begin{record}{"Record Label"}
　データレコードの開始を宣言します。
・\end{record}

データレコードの終了を宣言します。

### 1.6.1 スカラー及び配列データのデータ実体部

スカラーデータにデータを割り当てる時は、変数名、コロン、値の順に並べます。配列データの場合は、コロンの後に値の列を[]で囲んで記述します。値の区切りは空白またはコンマを使います。値の列[]の中は空でもよく、大きさゼロの配列データとして扱われます。

```
|begin{data}
VariableName1:Value1
VariableName2:Value2 ;
ArrayName1[]:[v1,v2,v3,v4,...]
ArrayName2[]:[w1 w2 w3 w4];
ArrayName3[]:[] ;
|end{data}
```

１行で記述ができる場合は以下のように|data文を用いて、データの定義を行うことができます。

```
|data Variable Name: Value;
```

（例）

```
\begin{def}
intArrayValue[]:int;
intValue:int;
stringArrayValue[]:string;
stringArrayValue:string;
\end{def}
\begin{data}
intArrayValue[]:[0,1,2,3,4,5,6,7,8,9]
intValue:3616
stringArrayValue[]:[ ' text1 ' , ' text2 ' ]
stringArrayValue:"textstring"
\end{data}
\def temperature: double;
\data temperature: 1.0;
```

### 1.6.2 複合型データのデータ実体部

複合型データにデータを割り当てる時、複合型データの最も上位にあるデータ名の次にコロン

を置き、その右に{}及び[]を区切りにしてデータを並べます。

配列でない複合型データの場合は、

***複合型データ名:{ データ並び }***

複合型データが配列の場合は、

***複合型データ名[]:[{ データ並び},{ データ並び},{ データ並び}]***

等とデータが並びます。

複合型データがネストしている場合でも、"下位のデータ名："は記述せずにデータの並べ方のみ同様な方法で続けていきます。

下記に例を示します。

```
\begin{def}
config:{
  time:double,
  atom[]:{
    pos[]:int,
    vel:{x:int,y:int,z:int},
    id:int
  }
  kinetic_energy:int,
  pressure:double
};
\end{def}
\begin{data}
config:{
  10.0
  [
    {[10,11,12],{-10,-11,-12},15},
    {[20,21,22],{-20,-21,-22},25},
    {[30,31,32],{-30,-31,-32},35}
  ]
  40
  5.56
};
\end{data}
```

## 1.6.3 データレコード部

データレコード部は下記の書式です。

\begin{record}{"*Record_Label*"}

\begin{data}

DATA ...

\end{data}

\end{record}

*Record_Label* はデータレコードを検索するためのキーとなる文字列です。[4]
エンジンインターフェイスライブラリやUDFManagerからデータレコード部を操作する時にレコード番号を数値で指定することができます。このとき最初のレコードの番号はゼロから始まります。

## 1.7 コメント部及びコメント書式

ヘッダー部、データ定義部、コモンデータ部及びデータレコード部は、例えば\begin～\end というようなキーワードで示されます。ＵＤＦデータファイルにおける上記データ部以外の部分は、ＵＤＦの有効なデータとみなされないコメント部となります。
また、それぞれのデータ部内においては下記の書式により、コメントを記述することができます。
１行のみ有効なコメントとして、// 以降の行端までをコメントとみなします。
複数行有効なコメントとして、/* から*/ までをコメントとみなします。
さらに、コモンデータ部及びデータレコード部においては、ダブルクォートで挟まれない区切り記号を含まない文字列をコメントとみなします。
（例）

```
\begin{def}
config:{
  time:double, // TIME
  atom[]:{
    pos[]:int, // x:pos[0],y:pos[1],z:pos[2]
    vel:{x:int,y:int,z:int},
    id:int
  }
  kinetic_energy:int,
  pressure:double
};
\end{def}
COMMENT AREA...
\begin{data}
config:{
  time 10.0
  atom [
    {[10,11,12],{-10,-11,-12},15},
    // {[20,21,22],{-20,-21,-22},25},
    {[30,31,32],{-30,-31,-32},35}
```

[4] Record_Label を使って、GOURMET の UDF 表示画面や Python でレコード（計算ステップ）を検索できる。

```
  ]
  kinetic_energy 40
  pressure 5.56
};
\end{data}
```

## 1.8 インクルード機能

データ名定義部およびコモンデータ部には､他のUDFファイルのデータ名定義部およびコモンデータ部をインクルード機能を用いて埋め込むことができます。ただし埋め込まれるUDFファイルにはデータレコード部を含めることはできません。

インクルードの書式は、以下ようになります。

*|include{"included_file_name"}*

*included_file_name*はインクルードされるファイル名であり、インクルードするUDFファイルからの相対パスあるいは絶対パスで指定します。

さらにインクルードされるファイルがあるディレクトリを環境変数UDF_DEF_PATHに設定しておけばそのディレクトリからインクルードされるファイルが検索されます。環境変数UDF_DEF_PATHに複数のディレクトリを設定する場合は､Windowsではセミコロン";"で区切り、その他のＯＳではコロン":"で区切ります。

## 1.9 UDF データ型の数値精度・範囲

UDFデータ型が扱える数値精度・範囲は以下の表のとおりです。

| UDF データ型 | | 精度（注） | 範囲 |
|---|---|---|---|
| short | 最小値 | | -32768 |
| | 最大値 | | 32767 |
| int | 最小値 | | -2147483647 |
| | 最大値 | | 2147483647 |
| long | 最小値 | | -2147483647 |
| | 最大値 | | 2147483647 |
| float | 最小値 | 小数点以下６桁 | 1.175494351e-38 |
| | 最大値 | 小数点以下６桁 | 3.402823466e+38 |

| | | | |
|---|---|---|---|
| single | 最小値 | 小数点以下６桁 | 1.175494351e-38 |
| | 最大値 | 小数点以下６桁 | 3.402823266e+38 |
| double | 最小値 | 小数点以下１４桁 | 2.2250738585072014e-308 |
| | 最大値 | 小数点以下１４桁 | 1.7976931348623157e+308 |
| string | 最小値 | | 長さゼロ |
| | 最大値 | | 長さ制限なし |

（注）GOURMETのEditor画面から右欄範囲の桁数を入力できますが、ファイルへの書き出しおよびエンジンインタフェースライブラリでは精度に示した有効桁数となります。

## 1.10 ＵＤＦパーサーエラーメッセージ

ＵＤＦ書式で記述されたファイルをGOURMETに取り込むとき、ＵＤＦパーサーが構文解析を行い、正しいＵＤＦ書式ファイルかどうかが判定されます。ＵＤＦ書式ファイルに不整合がある場合、下記に示すエラーメッセージをダイアログなどに出力してＵＤＦ書式ファイルのエラー箇所を表示します。

ファイル読み込み時にエラーが起こった場合、定義部分及びコモンデータ部分でエラーが起こるとデータは全く読み込まれません。レコード部でエラーが起こった時は、エラー発生直前のデータまでは読み込まれます。

下記例のエラーメッセージのfilename にはＵＤＦ書式ファイル名、line 99の99の部分には不整合のある付近の行番号が示されます。

(1) ヘッダー部に関するエラー

■ヘッダーにstring型以外のデータ型を指定している。

[error 1]filename:line 99 near "int":no supported type in header.

(2) 定義部に関するエラー

■定義記述法が間違っている。

[error 1]filename:line 99 near ";":parse error.

(例)

aaa;int

■ 2 次元以上の配列を定義しようとした。

[error 1]filename:line 99 near ":":array is limitted in dimension 1.

■データ名にデータ型名を使用している。

[error 1]filename:line 99 near "データ名":Illegal data/type name.

■データ定義の配列インデックス部に文字列がある。
[error 1]filename:line 99 near "abc":parse error.
（例）
xx[abc]:int
■クラスの中にID 型が２回以上定義されている。
[error 1]filename:line 99 near "abc":ID is defined twice in a class.
■クラスの中にKEY 型が２回以上定義されている。
[error 1]filename:line 99 near "abc":KEY is defined twice in a class.
■ KEY/ID 参照型の<..,KEY>,<..,ID>の中に、"KEY"または"ID"以外の文字が記述されている。
[error 1]filename:line 99 near "abc":KEY/ID error.
■データ型の間違い。
[error 1]filename:line 99 near "intt":intt.
■括弧の間違い。
[error 1]filename:line 99 near "}":parse error.
■同じ名称のデータ定義を重複して行おうとした。
[error 1]filename:line 99 near "}":definition duplication error.
（注）複合型データの定義など宣言が複数行にわたる場合は、重複しているデータの定義が完了した時点の行番号が出る。
（例）
[error 1]filename:line 6 near "}":definition duplication error.
1: \begin{def}
2: aaa:int
3: aaa:{
4:  b1:int
5:  b2[]:int
6: }
7: \end{def}

(3) データ部に関するエラー
■データ型が間違っている（int であるのに文字列が書かれている等）。
filename:line 99:Type mismatch,definition[定義名]data[書かれているデータ].
■括弧の間違い（と[] の間違い）。
[error 1]filename:line 99 near "}":bracket error.
■データに過不足がある。
[error 1]filename:line 99 near "データ値":.
（例）

```
[error 1]filename:line 11 near "88":.
1: \begin{def}
2: bbbb:{
3: c1:int
4: c2[]:int
5: }
6: \end{def}
7: \begin{data}
8: bbbb:{
9: 9999 // bbbb.c1
10: [ 1 ,2, 3] // bbbb.c2[]
11: 88 // Error data.
12: }
13: \end{data}
```

■データ定義がされていない。

```
[error 1]filename:line 99 near ":":No data definition[エラーデータ名].
```

## 1.11 ＵＤＦパーサー警告メッセージ

ＵＤＦ書式で記述されたファイルが読み込まれるとき、基本的なデータの読み込みには問題が無くても、単位変換機能などの付加的機能が制限されるような誤りがＵＤＦファイルに記述されている場合があります。そのようなときには、その機能が作用しない、あるいはデータが無視されるなどの警告を出して処理を続けます。

■ UDF ファイルに書き込み権が無い。

```
Readonly file. Please save an another file.
```

■”global def”定義のデータ実体が、レコードに記述されているので無視するメッセージ。

```
Data are disregarded[DATANAME],since the data defined in the global-def is in a record.
```

■単位変換で使用するＵＤＦデータ（{$udf_name}により指定される）が無い。

```
No udf variable required for unit conversions:[UNITNAME].
```

■基本単位(kg,m,s,A,...) がユーザー単位によって上書きされた。

```
The base unit[UNITNAME] is overwritten by the user unit.
```

■単位定義の意味間違い。

```
The unit expression is wrong.
```

■データ定義の単位指定の意味間違い。

Cannot recognize as a unit:[UNITNAME].

■単位定義に問題がある場合の一般メッセージ。

Unit conversion service cannot be opened.

■単位系定義ファイルの単位指定(semantics) の誤り。

Error: near [UNITNAME].

■単位[UNITNAME] はユーザー単位として定義されているので、単位系の単位として使用されない。

Since the following unit has been used:[UNITNAME],it does not include in the unit system.

■コモンおよびカレントレコードにデータが無い場合、そのデータの定義の修正が定義編集画面で可能となる。

ユーザーがその定義情報を修正して定義編集画面を終える時、他のレコードにデータがあった場合、そのデータが破棄された旨のメッセージ。

The following entity data was lost,[DATANAME].

# 第 2 章サンプル

UDF データ書式のサンプルを示します。

## 2.1 基本サンプル

基本的な型と配列変数、単純な複合型データのある例を示します。

```
\begin{header}
\begin{def}
EngineType:string;
EngineVersion:string;
IOType:string;
ProjectName:string;
Comment:string;
\end{def}
\begin{data}
EngineType:"WGO"
EngineVersion:"VO"
IOType:"IN"
ProjectName:"UDFBasicSample"
Comment:""
\end{data}
\end{header}
\begin{def}
intArrayValue[]:int;
intValue:int;
shortValue:short;
longValue:long;
floatValue:float;
singleValue:single;
doubleValue:double;
stringValue:string;
shortArrayValue[]:short;
```

```
longArrayValue[]:long;
floatArrayValue[]:float;
singleArrayValue[]:single;
doubleArrayValue[]:double;
stringArrayValue[]:string;
componentType:{
  intType:int,
  shortType:short,
  longType:long,
  floatType:float,
  singleType:single,
  doubleType:double,
  stringType:string
};
componentArrayType:{
  intArrayType[]:int,
  shortArrayType[]:short,
  longArrayType[]:long,
  floatArrayType[]:float,
  singleArrayType[]:single,
  doubleArrayType[]:double,
  stringArrayType[]:string,
};
\end{def}
\begin{def}
baseComponent:{
  minValueSet:componentType,
  maxValueSet:componentType,
};
baseArrayComponent:{
  componentSet[]:componentArrayType
};
\end{def}
\begin{data}
intArrayValue[]:[0 1 2 3 4 5 6 7 8 9 ]
intValue:1
```

```
shortValue:1
longValue:2
floatValue:3.0
singleValue:4.0
doubleValue:5.0
stringValue:"test string"
shortArrayValue[]:[0 1 2 3 4 5 6 7 8 9 ]
longArrayValue[]:[0 1 2 3 4 5 6 7 8 9 ]
floatArrayValue[]:[0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 ]
singleArrayValue[]:[0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 ]
doubleArrayValue[]:[0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 ]
stringArrayValue[]:["string0" "string1" "string2" "string3" "string4" ]
\end{data}
\begin{record}{"Step 0"}
\begin{data}
baseComponent:{
{
  -2147483647, // int min
  -32768, // short min
  -2147483647, // long min
  1.175494351e-38, // float min
  1.175494351e-38, // single min
  2.2250738585072014e-308, // double min
  ""
},
{
  2147483647, // int max
  32767, // short max
  2147483647, // long max
  3.402823266e+38, // float max
  3.402823266e+38, // single max
  1.7976931348623157e+308, // double max
  "abcdefghijklmnopqrstuvwxyz"
}
}
baseArrayComponent:{
```

```
  [{
    [0],
    [10,11],
    [30,31,32,33],
    [40,41,42,43,44],
    [50,51,52,53,54,55],
    [60,61,62,63,64,65,66],
    ["0","1","2","3","4","5","6"]
  } {
    [0,1,2,3,4,5,6,7],
    [10,11,12,13,14,15,16],
    [30,31,32,33,34],
    [40,41,42,43],
    [50,51,52],
    [60,61],
    ["0"]
  }]
}
\end{data}
\end{record}
```

## 2.2 KEY 型/ID 型サンプル

KEY 型及びID 型を用いたデータ例を示します。

Vertexデータに点の座標データがあり、Edge（辺）、Face（面）及びCell（体積要素）はVertexデータのIDのみを参照しています。

```
\begin{header}
\begin{def}
EngineType : string;
EngineVersion : string;
IOType : string;
ProjectName : string;
Comment : string;
\end{def}
\end{header}
```

```
\begin{def}
class Vector3D:{ // 3D ベクトル型
  x:float // X 座標
  y:float // Y 座標
  z:float // Z 座標
};
class Vertex:{ // 点のクラス
  id:ID
  position:Vector3D // 座標
};
class Edge:{ // 辺のクラス
  id:ID
  vertex[]:<Vertex,ID> // 点のID のリスト
};
class Face:{ // 面のクラス
  id:ID
  vertex[]:<Vertex,ID> // 点のID のリスト
};
class Cell:{ // 体積要素のクラス
  id:ID
  vertex[]:<Vertex,ID> // 点のID のリスト
};
class Neighbor:{ // 近傍の要素配列型
  vertex[]:<Vertex,ID> // 点のID のリスト
  edge[]:<Edge,ID> // 辺のID のリスト
  face[]:<Face,ID> // 面のID のリスト
  cell[]:<Cell,ID> // 体積要素のID のリスト
};
class MeshAxis:{
  values[]:double
};
class Mesh:{ // メッシュ型
  name:KEY // 名前
  type:string // タイプ
  axes[]:MeshAxis // メッシュ軸の配列
  periodic[]:int // メッシュ軸の周期境界条件の配列
```

```
};
class MeshData:{
  name:<Mesh,KEY>
  vertex[]:Vertex // メッシュ点の位置ベクトルの配列
  edge[]:Edge // 辺要素のデータ配列
  face[]:Face // 面要素のデータ配列
  cell[]:Cell // 体積要素のデータ配列
  neighbor_of_a_vertex[]:Neighbor //点要素の近傍の要素配列
  neighbor_of_a_edge[]:Neighbor //辺要素の近傍の要素配列
  neighbor_of_a_face[]:Neighbor //面要素の近傍の要素配列
  neighbor_of_a_cell[]:Neighbor //体積要素の近傍の要素配列
};
mesh_parameter:Mesh;
mesh_data:MeshData;
\end{def}
// COMMON
\begin{data}
mesh_parameter:{
   "TESTMESH"
  "UNSTRUCTURED_RECT"
  [
    {
      [1.0,2.0,15.0]
    } {
      [1.0,2.0,15.0]
    } {
      [1.0,2.0,15.0]
    }
  ]
  [0 0 0]
}
mesh_data:{
  "TESTMESH"
  []
  []
  [
```

```
    {
      1,
      [1,2,3]
    } {
      2,
      [1,3,4]
    } {
      3,
      [1,4,2]
    } {
      4,
      [2,4,3]
    }
  ]
  []
  []
  []
  []
  []
}
\end{data}
\begin{record}{"#0"}
\begin{data}
mesh_data:{
  "TESTMESH",
  [
    {1,{0,0,0}},
    {2,{1,0,0}},
    {3,{0,1,0}}
    {4,{0,0,1.2}}
  ]
  []
  []
  []
  []
  []
```

```
  []
  []
}
\end{data}
\end{record}
```

## 2.3 入力データサンプル

入力データに解析条件が多数ある場合の例を示します。select型を用いるとGOURMETの編集画面からの入力が簡単に行えるようになります。

```
\begin{header}
\begin{def}
EngineType:string;
EngineVersion:string;
IOType:string;
ProjectName:string;
Comment:string;
\end{def}
\end{header}
\begin{def}
Simulation_Conditions:{
  Solver:{
    Solver_Type:select{"Dynamics","Minimize"}, // Dymanics/Minimize の選択
    Dynamics:{
      Dynamics_Algorithm:select{ // Dymanics の種類
        "NVT_Nose_Hoover",
        "NVT_Berendsen",
        "NVT_Kremer_Grest",
        "NPH_Andersen",
        "NPH_Parrinello_Rahman",
        "NPH_Brown_Clarke",
        "NPT_Andersen_Nose_Hoover",
        "NPT_Andersen_Kremer_Grest",
        "NPT_Parrinello_Rahman_Nose_Hoover",
```

```
  "NPT_Parrinello_Rahman_Kremer_Grest",
  "NPT_Berendsen",
  "NPT_Brown_Clarke",
  "SLLOD_PT_Const",
  "OTHER"
},
NVT_Nose_Hoover:{ // それぞれの選択時に必要な入力項目
  Q:float
}
NVT_Berendsen:{ // それぞれの選択時に必要な入力項目
  tau_T:float
}
NVT_Kremer_Grest:{ // それぞれの選択時に必要な入力項目
  Friction:float
}
NPH_Andersen:{ // それぞれの選択時に必要な入力項目
  Cell_Mass:float
}
NPH_Parrinello_Rahman:{ // それぞれの選択時に必要な入力項目
  Cell_Mass:float, // Cell Mass
  Fix_Angle:short, // Cell Angle を固定するかどうかのフラグ
  Fix_Cell_Length:string // 任意のCell length(x or y or z) を
  // 固定する。x,xy,yz 等string で入力。
  // Fix_Angle=true の時のみ有効
}
NPH_Brown_Clarke:{
  tau_P:float, // Coupling constant of pressure bath
  Fix_Angle:short, // Cell Angle を固定するかどうかのフラグ
  Fix_Cell_Length:string // 任意のCell length(x or y or z) を
  // 固定する。x,xy,yz 等string で入力。
  // Fix_Angle=true の時のみ有効
}
NPT_Andersen_Nose_Hoover:{
  Cell_Mass:float, // Cell Mass
  Q:float // Coupling constant
}
```

```
  NPT_Andersen_Kremer_Grest:{
    Cell_Mass:float, // Cell Mass
    Friction:float // Friction constant
  }
  NPT_Parrinello_Rahman_Nose_Hoover:{
    Cell_Mass:float // Cell Mass
    Fix_Angle:short // Cell Angle を固定するかどうかのフラグ
    Fix_Cell_Length:string // 任意のCell length(x or y or z) を
    // 固定する
    Q:float // Coupling constant
  }
  NPT_Parrinello_Rahman_Kremer_Grest:{
    Cell_Mass:float // Cell Mass
    Fix_Angle:short // Cell Angle を固定するかどうかのフラグ
    Fix_Cell_Length:string // 任意のCell length(x or y or z) を
    // 固定する
    Friction:float // Friction constant
  }
  NPT_Berendsen:{
    tau_P:float // Coupling constant of pressure bath
    tau_T:float // Coupling constant of heat bath
  }
  NPT_Brown_Clarke:{
    tau_P:float // Coupling constant of pressure bath
    Fix_Angle:short // Cell Angle を固定するかどうかのフラグ
    Fix_Cell_Length:string // 任意のCell length(x or y or z) を
    // 固定する
    tau_T:float // Coupling constant of heat bath
  }
  SLLOD_PT_Const:{
    tau_P:float // Coupling constant of pressure bath
    Fix_Cell_Length:string // 任意のCell length(x or y or z) を
    // 固定する
  }
}
Minimize:{
```

```
    Minimize_Algorithm:select{ // Minimize アルゴリズムの選択。
      "SD", // steepest descent、
      "CG", // conjugate gradient、
      "Cascade", // SD とCG の組み合わせ
      "MSI"
    }
    SD:{
      Max_Iteration:int, // 最大iteration
      Converge_Force:float // 収束判定の際のmaximum force
      Output_Interval_Steps:int, // データ出力の間隔。
    }
    CG:{
      Max_Iteration:int, // 最大iteration
      Converge_Force:float // 収束判定の際のmaximum force
      Output_Interval_Steps:int, // データ出力の間隔。
    }
    Cascade:{
      SD_Max_Iteration:int, // SD の最大iteration
      SD_Converge_Force:float // SD 収束判定の際のmaximum force
    }
    MSI:{
      Constraint:short // Bond constraint を加えるかどうか
      Min_Coord:float
      Max_Coord:float
      MF_Parameter:float // 制約式のスラック変数の重み
      MO_Parameter:float // 目的関数の重み
    }
  }
}
Boundary_Conditions:{
  a_axis:string
  b_axis:string
  c_axis:string
  Periodic_Bond:short
}
Output_Flags:{
```

```
    Statistics:{
      Energy:short, // energy を出力するかどうか
      Temperature:short, // temperature を出力するかどうか
      Pressure:short, // pressure を出力するかどうか
      Stress:short, // stress を出力するかどうか
      Volume:short, // volume を出力するかどうか
      Density:short, // density を出力するかどうか
      Cell:short, // cell を出力するかどうか
      Wall_Pressure:short // wall pressure を出力するかどうか
      Energy_Flow:short // 温度スケールにより出入りしたエネルギー
    }
    Structure:{
      Position:short // 座標を出力するかどうか
      Velocity:short // 速度を出力するかどうか
      Force:short // 力を出力するかどうか
    }
  }
  Fix_Node[]:{
    Mol_Index:int // 座標を固定するAtom の配列インデックスの配列
    Atom_Index:int
  }
};
\end{def}
// COMMON
\begin{data}
Simulation_Conditions:{
  {"Dynamics",
    {"NVT_Kremer_Grest",
      {20.0}
      {100.0}
      {0.50}
      {20.0}
      {20.0,0,""}
      {100.0,1,""}
      {20.0,20.0}
      {20.0,0.50}
```

```
        {20.0,0,"",20.0}
        {20.0,1,"",0.50}
        {100.0,100.0}
        {100.0,0,"",100.0}
        {100.0,"x"}
      }
      {"MSI",
        {1000,0.10,100},
        {1000,0.10,100},
        {1000,1000.0}
        {1,-100.0,100.0,5.0e-002,0.950}
      }
    }
    {"PERIODIC","PERIODIC","REFLECTIVE2",0}
    {{1,1,1,1,0,0,0,0,0}{1,0,0}}
    []
}
\end{data}
```

# 第 3 章単位についての補足

単位の次元および単位系について説明します。ユーザーは９個 SI 基本単位から新たな単位を作ることができます。新たな単位は、基本単位の積の何乗かという次元情報を持っています。データの単位変換は、同じ単位の次元を持つ単位の間で実行することができます。

## 3.1 単位の次元

UDFでは下記の９種のSI基本単位から単位を組み立てます。

*[kg](kilogram)*
*[m](meter)*
*[s](second)*
*[A](ampere)*
*[K](kelvin)*
*[mol](mole)*
*[cd](candela)*
*[rad](radian)*
*[sr](steradian)*

組み立てられた単位は、９次元の単位次元を持ちます。例えば以下のように定義された単位は、表3.1に示す単位次元を持ちます。

```
\begin{unit}
PI=3.141592654
[degree]=PI/180[rad]
[N]=[kg*m/s^2] // force
[Pa]=[N/m^2] // pressure
[J]=[N*m] // energy
[W]=[J/s] // power
[C]=[A*s] // electric charge
[V]=[W/A] // voltage
[F]=[C/V] // electrostatic capacity
[ohm]=[V/A] // electric resistance
```

```
[S]=[A/V] // conductance
[Wb]=[V*s] // magnetic flux
[T]=[Wb/m^2] // magnetic flux density
[H]=[Wb/A] // inductance
[m^2] // area
[m^3] // volume
[kg/m^3] // mass density
[m/s] // velocity
[m/s^2] // acceleration
[rad/s] // angular velocity
[rad/s^2] // angular acceleration
\end{unit}
```

| 単位名 | M(注1) | L | T | E | TT | A | LI | PA | SA |
|---|---|---|---|---|---|---|---|---|---|
| PI(定数) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| [degree] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| [N] | 1 | 1 | -2 | 0 | 0 | 0 | 0 | 0 | 0 |
| [Pa] | 1 | -1 | -2 | 0 | 0 | 0 | 0 | 0 | 0 |
| [J] | 1 | 2 | -2 | 0 | 0 | 0 | 0 | 0 | 0 |
| [W] | 1 | 2 | -3 | 0 | 0 | 0 | 0 | 0 | 0 |
| [C] | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| [V] | 1 | 2 | -3 | -1 | 0 | 0 | 0 | 0 | 0 |
| [F] | -1 | -2 | 4 | 2 | 0 | 0 | 0 | 0 | 0 |
| [ohm] | 1 | 2 | -3 | -2 | 0 | 0 | 0 | 0 | 0 |
| [S] | -1 | -2 | 3 | 2 | 0 | 0 | 0 | 0 | 0 |
| [Wb] | 1 | 2 | -2 | -1 | 0 | 0 | 0 | 0 | 0 |
| [T] | 1 | 0 | -2 | -1 | 0 | 0 | 0 | 0 | 0 |
| [H] | 1 | 2 | -2 | 2 | 0 | 0 | 0 | 0 | 0 |
| [m^2] | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| [m^3] | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| [kg/m^3] | 1 | -3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| [m/s] | 0 | 1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 |
| [m/s^2] | 0 | 1 | -2 | 0 | 0 | 0 | 0 | 0 | 0 |
| [rad/s] | 0 | 0 | -1 | 0 | 0 | 0 | 0 | 1 | 0 |
| [rad/s^2] | 0 | 0 | -2 | 0 | 0 | 0 | 0 | 1 | 0 |

表3.1: 単位および単位の次元数

（注１）M:mass [kg](kilogram), L:length [m](meter), T:time [s](second), E:electric current [A](ampere), TT:thermodynamic temperature [K](kelvin), A:amount of substance [mol](mole), LI:luminous intensity [cd](candela), PA:plane angle [rad](radian), SA:solid angle [sr](steradian)

## 3.2 単位系指定ファイルの書式

ここで使用する単位系とはデータ値を表示するために基準とする単位の集まりです。通常、単位が指定されたＵＤＦがGOURMETに読み込まれた場合、ＵＤＦに定義された単位でデータ値が表示されます。

単位系を指定することにより、単位系に記述された単位と同じ単位次元をもつデータは、一斉に単位系に記述された単位での値に変換されます。

データ値を表示するための基準単位を単位系指定ファイルに記述しておくことにより、単位の一括変換が可能になります。

単位系指定ファイルの書式は以下のとおりです。１つの単位の定義は、改行をしないで１行に記述します。

*|begin{unit_system}{"unit system name"}*
*New_Constant = CONSTANT*
*[New_Unit1] = [Unit_Expression1]*
*[New_Unit2] = CONSTANT[Unit_Expression2]*
*[Unit_Expression3]*
*|end{unit_system}*

ＳＩ単位系の単位系指定の例を以下に示します。

（ＳＩ単位系はGOURMETに組み込み済みです）

```
\begin{unit_system}{"SI"}
[kg]
[m]
[s]
[A]
[K]
[mol]
[cd]
[rad]
```

```
[sr]
[Hz]=[1/s] // frequency
[N]=[kg*m/s^2] // force
[Pa]=[N/m^2] // pressure
[J]=[N*m] // energy
[W]=[J/s] // power
[C]=[A*s] // electric charge
[V]=[W/A] // voltage
[F]=[C/V] // electrostatic capacity
[ohm]=[V/A] // electric resistance
[S]=[A/V] // conductance
[Wb]=[V*s] // magnetic flux
[T]=[Wb/m^2] // magnetic flux density
[H]=[Wb/A] // inductance
[m^2] // area
[m^3] // volume
[kg/m^3] // mass density
[m/s] // velocity
[m/s^2] // acceleration
[rad/s] // angular velocity
[rad/s^2] // angular acceleration
[N*m] // moment of force
[N/m] // surface tension
[Pa*s] // viscosity
[m^2/s] // kinematic viscosity
[W/m^2] // heat flux density, irradiance
[J/K] // heat capacity, entropy
[J/kg/K] // specific heat capacity
[W/m/K] // thermal conductivity
[V/m] // electric field strength
[C/m^2] // electric flux density
[F/m] // dielectric constant
[A/m^2] // current density
[A/m] // magnetic field strength
[H/m] // permeability
[mol/m^3] // molarity
```

```
[cd/m^2] // luminance
[1/m] // wave number
\end{unit_system}
```

もし単位系定義を新たに指定して、その単位系定義に含まれていない単位の組み合わせがUDFで使用されている場合、新たに指定した単位系定義に含まれている基本単位（SI基本単位と同じ単一の次元を持つ単位）を組み合わせて対応する単位名を自動生成します。基本単位に相当する単位が単位系定義に記述されていなければ、その基本単位についてはSI単位を用います。

ただし、次の単位次元を持つ単位については、単位系に定義されていれば、その単位名に置き換えられます。

```
[W] (power:次元[1, 2,-3, 0, 0, 0, 0, 0, 0])
[J] (energy:次元[1, 2,-2, 0, 0, 0, 0, 0, 0])
[Pa] (pressure:次元[1,-1,-2, 0, 0, 0, 0, 0, 0])
[N] (force:次元[1, 1,-2, 0, 0, 0, 0, 0, 0])
```

すなわち、[kg*m^2/s^2]は基本単位のみで表示されていますが、自動的に[J]に置き換える操作が実行されます。

[Pa]と同じ次元を持つ単位については、単位の次元がlength=-1,mass=1,time = -2±1であれば、"単位系定義内の[Pa]と同じ次元を持つ単位×残りの単位" に置き換えられます。

[N]についても同様に、[W]あるいは[J]に相当する単位に置き換えられなかった時、length=1±1,time=-2±1の単位次元を持つ単位が[N]と同じ次元を持つ単位×残りの単位にまとめられます。例えば、[kg*m/s^3]は[N/s]に置き換えられます。

# 第４章　バイナリＵＤＦ形式

## 4.1 バイナリＵＤＦとは

バイナリ UDF とはテキストデータファイルの入出力を高速化するために UDF のデータ部をバイナリ化したものです。データ定義部は定義部自体のデータサイズが比較的大きくないこと、定義部の可読性の良さなどからバイナリ化されていません。

バイナリ UDF 形式は、ヘッダー部および定義部は従来のテキスト UDF であり、コモン部およびレコード部はテキスト形式またはバイナリ形式で扱うことができます。レコード部は任意のレコード以降をバイナリ形式とすることができます。コモン部あるいはレコード部で一旦バイナリ形式にすると以降のレコードはバイナリ形式で扱われます。バイナリ形式になった以降のレコードをテキスト UDF に戻すことはできません。

## 4.2 プラットフォーム依存性

バイナリデータによるファイル入出力には一般にコンピュータプラットフォームの違いによりエンディアンが異なる問題が存在します。バイナリ化 UDF の入出力では、エンディアンの相違による影響を取り除く処理を行っているのでコンピュータプラットフォーム依存性はありません。

バイナリ UDF 形式は最初にバイナリデータが書き込まれたコンピュータプラットフォームのエンディアンを記録しておくので、バイナリ UDF ファイルを異なるエンディアンのコンピュータプラットフォームで処理した時にはバイト入れ替え処理が発生します。

## 4.3 バイナリフォーマット

UDFファイル中でテキスト部分からバイナリ部分に替わる位置に、\begin{binary} が書き込まれます。バイナリデータは以下のフォーマットに従います。

**バイナリデータフォーマット**

| **データ** | 形式・型 | 備考 |
|---|---|---|
| **バイナリ開始キー** | \begin{binary} | |
| **データ部開始キー** | "DATA" | |
| **バイナリフォーマットバージョン** | "01.00.00" | |
| **バイナリチェックビット** | 0,0D,0A,0,FF,FF,FF,FF | 8バイト（データチェック） |
| **[バイナリ情報部]** | 複合データ | 下記[バイナリ情報部]参照 |
| **[バイナリデータ名宣言部]** | 複合データ | 下記**[バイナリデータ名宣言部]参照** |
| **[ステップデータ部]** | 複合データ | ステップ数分繰り返し |
| … | | |

[バイナリ情報部]

| **エンディアン** | 1バイト ASCII | B: Big , L : Little |
|---|---|---|
| **データタイプ数** | 8ビット符号無整数 | |
| **タイプ名長** | 8ビット符号無整数 | 以下[**データタイプ**数]分繰り返し |
| **タイプ名** | CHAR | |
| **タイプID** | 8ビット符号無整数 | [バイナリ化データ]部で使用 |
| **データ長** | 8ビット符号無整数 | バイナリデータバイト数 |
| **拡張フラグ** | 0,0,0,0 | 4バイト |

[バイナリデータ名宣言部]

| **キーワード** | "NPIF" | 4バイト |
|---|---|---|
| **宣言数** | 32ビット符号無整数 | |
| **データ名** | 無制限長文字列形式 | (len/255 + len + 1)バイト |
| **データ ID** | 32ビット整数 | |
| **データ親 ID** | 32ビット整数 | |
| **タイプ ID** | 8ビット符号無整数 | |
| **配列次元** | 8ビット符号無整数 | |

[ステップデータ部]内容

| | | |
|---|---|---|
| **開始キー** | 4 バイト文字列 | コモン："COMN" , ステップ："STEP" |
| **ステップラベル** | 無制限長文字列形式 | |
| **[時間発展データ位置情報]** | 複合 | |
| **ステップデータサイズ** | 64 ビット符号無整数 | 次（繰り返し数）からのサイズ |
| **バイナリ化データ繰り返し数** | 32 ビット符号無整数 | |
| **[バイナリ化データ]** | 複合 | データ繰り返し数分繰り返し |

[時間発展データ位置情報]内容

| | | |
|---|---|---|
| **時間発展データサイズ** | 32 ビット符号無整数 | |
| **時間発展データ数** | 32 ビット符号無整数 | |
| **データ ID** | 32 ビット整数 | |
| **データ開始キーからの位置** | 64 ビット符号無整数 | |

[バイナリ化データ]　配列データ（構造体および値）

| | | |
|---|---|---|
| **データ ID** | 32 ビット整数 | |
| **[配列サイズ情報]** | 複合 | 配列次元数分繰り返し |
| **データ数** | 32 ビット符号無整数 | ゼロの場合、"データ列"無し |
| **データ列** | データ型依存 | "データ数"分繰り返し |

[配列サイズ情報]内容　（配列次元数>0 の場合のみ存在）

| | | |
|---|---|---|
| **配列サイズ** | 32 ビット符号無整数 | |
| **配列データ** | 32 ビット符号無整数 | |

[バイナリ化データ]　スカラーデータ（親構造体が配列である場合がある）

| | | |
|---|---|---|
| **データ ID** | 32 ビット整数 | |
| **データ数** | 32 ビット符号無整数 | ゼロの場合、"データ列"無し |
| **データ列** | データ型依存 | "データ数"分繰り返し |

（フルスカラー構造体（全ての親構造体がスカラー）はバイナリデータに現れません。）

# 補遺　既知の問題点

ここでは新規開発中の機能で、一部完全に動作しない機能についてどのような場合に問題があるのかを説明します。

# 付録 A. OCTA2005 リリースにおける新機能、変更点の一覧

OCTA2003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。

・UDFファイルの仕様については、変更点はありません。