OCTA

Integrated simulation system for soft materials

UDF SYNTAX

REFERENCE

OCTA User's Group

DEC. 25 2002

Authors of the Manual

The Japan Research Institute, Limited

Program Developers

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

Acknowledgment

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

Contents

List of Tables

Chapter 1

UDF syntax

1.1 Introduction

This document specifies the basic syntax and the conventions of the UDF, which is a general-purpose technical calculation data format. The UDF can be used in order to describe all kinds of data used for scientific calculations.

1.2 Structure of UDF

The UDF consists of the header part, the unit definition part, the data name definition part and the data entity part. In the header part, the information for explaining the data file itself is described with loosely fixed format. In the unit definition part, new units for variables is defined based on the SI unit(the International System of Units). In the data name definition part, the names and type declarations of all variables which are used in the data entity part are described. The data entity part assigns numeric values or letters to the variables defined in the data name definition part.

1.3 Header part

The information about a data file is recorded in the header part. The information to be recorded is the name and the version number of analytic module with which the UDF data file is used, the I/O type of the data and the comments.

Even when there is no header part in a UDF data file, the reading and writing of the UDF file are completed. However, it is important to record what data is in the file. [1]

The format of the header part is as follows. The information on the UDF data is described into the character string shown in double quotes between \begin{data} and \end{data}.

```
\begin{header}
\begin{def}
    EngineType:string
    EngineVersion:string
    IOType:string
    ProjectName:string
    Comment:string
    Action:string
\end{def}
\begin{data}
    EngineType:"EngineTypeName"
```

[1] In GOURMET and UDF Manager for Python, the features to refer to and edit the information on header data is offered.

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

EngineTypeName and EngineVersionName are the name and the development version number of the analytic-engine module, respectively. "IN" or "OUT" is described to IOType. It is shown that "IN" is data for input and "OUT" is outputted data. An analysis name and a comment are described at ProjectNameString and CommentString. In Action, the names of action files required for the operations of GOURMET are described. Whitespace is allowed in the file names.

1.4 Unit definition part

In UDF, a real number variable can provide the unit of measurement. The unit conversion can be performed, when the data value is exchanged between numerical-analysis simulators, and when referring to or editing data values in GOURMET. Setting up a unit for the real number type data performs in the below-mentioned data name definition part. In the unit definition part, the units which should be assigned to each variable are defined from the primary units described below.

Letters which can be used for the unit definitions are capital or small letters of alphabet, and _ (underline). Moreover, numbers (0-9) can be used after the 2nd character. In the definitions of units, the difference between capital and small letter of alphabet is important.

When the unit can be expressed only by multiplication, division or power operation from the primary units or the primary units with prefix, there is no need of defining units in the unit definition part.

The primary units are given below.

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

Prefixes of units are given in Table 1.1.

The unit definition part is described as follows between $\backslash end\{unit\}$ from $\backslash begin\{unit\}$. You should describe one unit definition in one line.

$New\_Constant = CONSTANT$
$[New\_Unit] = [Unit\_Expression]$
$[New\_Unit] = CONSTANT[Unit\_Expression]$

New_Constant is a constant value used by unit definitional equations.

A unit is surrounded with square brackets in the UDF. In *CONSTANT*, a numeric value, a variable name or those computational expression is described. When using a variable name, $ must be attached to the head of the variable name, and surround by {}, as follows.

$[New\_Unit] = \{\$UDF\_Variable\}[Unit\_Expression]$

Symbol (name)	Factor
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}

Table 1.1: Prefixes of units

You can define new units by describing the computation expression of the base units or already defined derived units in *Unit_Expression*. You can use `* / ^` as the operators of multiplication, division, and power operation, respectively.

The examples of the definitions of new units are shown below:

```
\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 Data name definition part

In the data name definition part, all the variable name and the user defined type used in the data entity part are declared. A variable declaration consists of the name and the data type at least. You can also append the unit and the help to the variable declaration.

In the data name definition part, the following reserved words are used.

- **\def**
  When a variable definition is completed with just one line, you describe the declaration of variable after this keyword.

- **\begin{def}**
  When there are data name definitions with multiple lines, this keyword begines the data definition declarations.

- **\end{def}**
  When there are data name definitions with multiple lines, you should declare the end of the range of definition declarations with this keyword.

- **\begin{global_def}**
  This keyword shows that the definitions of global variable are started. Global variables can be accessed and have the same value in any data record.

  The data entity of global variable is described in the common data part.

- **\end{global_def}**
  This keyword shows that the declarations of global variable are ended.

1.5.1 Primitive data type

The primitive data types used in the UDF are as follows.

- **int**

  This shows that the variable is a integer.

- **long**

  This shows that the variable is a long integer.

- **short**

  This shows that the variable is a short integer.

- **float**

  This shows that the variable is a floating number.

- **double**

  This shows that the variable is a long floating number.

- **single**

  This shows that the variable is a floating type.

- **string**

  This shows that the variable is a character string.

- **select{"string_candidate1","string_candidate2",...}**

  This declares that the variable is a string type which has the selected string candidates. [2]

- **KEY**

  This is used for the string type member of a compound data type. The unique string type identifier for the array of the compound data type is stored.

- ***< User defined type*,KEY >**

  This declares the reference to KEY type data.

[2] When inputting data in the data edit display of GOURMET, a data input can be made only by selecting from the character string candidates.

- **ID**

  This is used for the integer type member of a compound data type. The unique integer type identifier for the entity array of the compound data type is stored.

- ***<User defined type*,ID>**

  This declares the reference to ID type data.

1.5.2 Delimiters

The following delimiters are used in the data name definition part.

- **{}**

  The scope (the range of a definition) of a compound data is shown by this pair.

- **[]**

  This shows the variable is an array.

- **;**

  This shows the termination of a data definition. This is omissible.

- **:**

  A variable name finishes and the data type or the compound data declaration follows this letter.

- **"string enclosed in double quotes", 'string enclosed in single quotes'**

  The character strings are enclosed in single quotes or double quotes.

1.5.3 Data name definition

How to actually define variables is explained. There are several kinds of methods of defining variables, for example, definitions of a scalar variable, an array variable, a compound data variable and a compound data variable with a user defined type.

a scalar or array variable

The definition of scalar variable is the following form. The definition of array variable attaches one [] to the end of a variable name.

\begin{def}
Variable_Name1*:*Type1
***Variable_Name2*:*Type2*;**
Array_Name1[]*:*Type1
***Array_Name2[]*:*Type2*;**
\end{def}

Here, *Variable_Name*,*Array_Name* are the arbitrary variable names which start in the letter from a to Z, and *Type* is data type. It begins by \begin{def} and ends by \end{def}.

When describing a definition at one line, the variable can be defined using a \def sentence as follows.

\def *Variable_Name*: *Type*;

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

Definitions of compound data

The definition of compound data forms the group of data variables using {} as follows.

\begin{def}
***Variable_Name1*: {**

Child_Name1*: *Type1
Child_Name2[]*: *Type2

}
\end{def}

It is possible to nest compound data (compound data containing other compound data), for example:

```
(Example)
\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 User-defined type

Compound type variable is the structure for treating some variables in one bundle, in UDF, the compound data of many hierarchies can be declared at once.

A user defined type treats a compound data declaration as new data type.

However, the compound data of an array type cannot be used as a user defined type.

In order to specify using the declaration of compound data only for a user defined type, the following reserved word is used.

- **class**

  This is a keyword which shows that it is the pure user defined type declaration. [3]

In the following example, Vector3d and Atom are used as a user defined type.

```
\begin{def}
class Vector3d:{x:float, y:float, z:float}; // Compound data for a coordinate value
Atom:{
    molID:int,
    typeName:string,
    coord:Vector3d,  // Compound data is used as a user defined type.
    vel:Vector3d,    // Compound data is used as a user defined type.
    force:Vector3d,  // Compound data is used as a user defined type.
    totalID:int
};
\end{def}
```

[3] It uses in order to control presenting of definition information by the GOURMET edit display. If a class prefix is attached to a data definition, the data variable will no longer be displayed on the GOURMET UDF edit display.

```
\begin{def}
Molecule:{
    name:string,
    atom[]:Atom     // Compound data is used as a user defined type.
};
\end{def}
```

The above-mentioned description can be rewritten only using a compound data definition as follows:

```
\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}
```

1.5.5 Definitions of KEY or ID type

The data name definitions of KEY type or ID type variables are as follows:

class *Class_name1*: {
Id_Variable :ID
Other_Variable ...
};

class *Class_name2*: {
Key_Variable :KEY
Other_Variable ...
};

The variable which refers to KEY and ID variable is defined using <ClassName,KEY >and <ClassName,ID >, respectively.

class Other_Class: {
Reference_id[] : <*Class_name1*,ID >
Reference_key[] : <*Class_name2*,KEY >
Other_Variable ...
};

The example actually used is as follows.

```
\begin{def}
class Vector3D: { // 3D vector type
    x:float // X coordinate
    y:float // Y coordinate
    z:float // Z coordinate
};
class Vertex: {
```

```
    id :ID      // ID indicating Vertex data
    position:Vector3D // coordinates
};
class Edge: {
    id :ID      // ID indicating Edge data
    vertex[] :<Vertex,ID> // The array of vertexes ID
};
class Face: {
    id :ID      // ID indicating Face data
    vertex[] :<Vertex,ID> // The array of vertexes ID
};
class Cell: {
    id :ID      // ID indicating Cell data
    vertex[] :<Vertex,ID> // The array of vertexes ID
};
class Neighbor: {
    vertex[] :<Vertex,ID> // The array of vertexes ID
    edge[] :<Edge,ID> // The array of edges ID
    face[] :<Face,ID> // The array of faces ID
    cell[] :<Cell,ID> // The array of cells ID
};
class Mesh:{
    name:KEY // A mesh name is set to KEY.
    type:string
    axes[]:MeshAxis // The array of the mesh axes
    periodic[]:int // Periodic-boundary-condition
};
class MeshData : {
    name : <Mesh,KEY>
    vertex[] : Vertex // The array of the position vectors of mesh points
    edge[] : Edge // array of edge elements
    face[] : Face // array of face elements
    cell[] : Cell // array of cell elements
    neighbor_of_a_vertex[] : Neighbor
    neighbor_of_a_edge[] : Neighbor
    neighbor_of_a_face[] : Neighbor
    neighbor_of_a_cell[] : Neighbor
};
\end{def}
```

1.5.6 Specification of units

A unit can be given to float or double type data. Specification of the unit to scalar data or array data is performed by surrounding a unit by square brackets [] after definition statement as follows.

\begin{def}
Variable_Name : ***Type***[*unit_name*]
Array_Name[] : ***Type***[*unit_name*]
\end{def}

Units may be given to the each variables in a user defined type from the first.

However, a user defined type may be treated as a general-purpose data type. In that case, at first the temporary unit is specified to a user defined type, and concrete data variables are known later, a true unit is specified to the user defined type.

When there is no need of specifying units to the user defined type variable with temporary units, empty [] is specified.

In the following examples, unit, unit1 and unit2 are temporary units.

```
\begin{def}
Me_val:double
Reduced_Parameters:{
  Mass:double
  Length:double
}
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}
```

1.5.7 Help feature

At the time of the declaration of a data definition, the explanatory note about the datum can be described. The help string is located at the last of the declaration of definition.

How to write the help about a data variable is very simple. The data-type of a definition declaration is followed by the help string, which is enclosed in double quotes. The help string enclosed with double quotes can be described continuously the any numbers of time.

In a help string, a newline, a double quote and a \ mark use \n, \" and a \\ sign, respectively.

```
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 Data entity part

The data entity part is the place which actually assigns values to the variable declared in the data name definition part.

The data entity part has two classification, those are the common data part and the data record part.

The common data is a variable not changing, even if an analysis step changes during analysis calculation.

The data name definition parts and the common data parts may be intermingled in a UDF file, but a data variable needs to be described before the data entity. If there is the data record parts, which followes the data name definition parts and the common data parts.

The data record part is the part prepared in order to describe the data which changes for every time step of analysis. A data record part is surrounded by \begin{record} and \end{record}.

In a data record part, no data name definition can be described.

In order to declare a data entity part, the following reserved word is used.

- **\data**

  This keyword is used when assignment of a value to a variable is completed in one line.

- **\begin{data}**

  This is an opening keyword used when assigning values in multiple lines.

- **\end{data}**

  This is a closure keyword used when assigning values in multiple lines.

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

  Opening of a data record part is declared.

- **\end{record}**

  Closure of a data record part is declared.

1.6.1 The data entity part of a scalar and array data

When assigning a value to a scalar variable, a variable name, a colon and a value are described in this order. In the case of an array variable, the sequence of a value is enclosed in square brackets([]) after the colon. Whitespace or a comma is used for the delimiter of the sequence of value. If the inside of square brackets is empty, the size of the array is zero.

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

When the description of data values is completed in one line, data values can be assigned using a \data sentence as follows.

\data *Variable Name*: *Value*;

```
(Examples)
\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 The data entity part of compound data

When assigning values to a compound data variable, the root data name of a compound data variable, a colon, and values surrounded by {} or [] are described in this order.

In the case of the compound data variable which is not an array:

data_name:{ values_of_components }

When a compound data variable is an array, as follows:

data_name[]:[{ values_of_components },{ values_of_components },...]

Even if a compound data is nested, no data names of components (data members) need to be described.

(Examples)

```
\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 Data record part

The format of the data record part is as follows.

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

DATA ...

\end{data}
\end{record}

Record_Label is the unique string of a record. It is a key for searching a record (calculation step) by using ***Record_Label***. You can also look for a record by the integer value corresponding to the sequence of the record written to the UDF file. In such case, the first record number is zero.

1.7 How to write comments

The header part, the unit definition part, the data declaration part, the common data part and the data record part are specified by the \begin{} and \end{} keyword. Portions other than the above-mentioned data division in a UDF data file are the so-called comment parts, which is not regarded as effective data.

The following format can describe comments in each effective data division.

The description from // until the end of the physical line is a comment. The multiple line comment is surrounded by /* and */. In the common data part and the data record part, the text which is not surrounded in double qoutes is a comment.

```
(Examples)

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

1.8 Include feature

An include feature allows a UDF files to refer to external UDF files. Although headers, unit definitions, data name definitions and common data part may be written in the external UDF file referred to, there must not be no record part.

The format of "include" is as follows.
\include{"***included_file_name***"}

- the line begins with "\".
- "include" string is lower case characters only.
- Whitespace may enter between "\", "include", "{" and "included_file_name","}".
- The file name included is specified with the relative path or absolute path from the UDF file specified first. (Notes [4])

[4] When using OCTA, the search path of include files can be set a environment variable named UDF_DEF_PATH. When setting two or more paths as environment variable UDF_DEF_PATH, on Windows, they are separated by semicolon";", and separated by colon":" on other OS.

1.9 Precision and range of numeric data

The precision and the range of numeric value in the UDF are as in the following table.

UDF data type		Precision(Notes [5])	Range
short	Minimum		-32768
	Maximum		32767
int	Minimum		-2147483647
	Maximum		2147483647
long	Minimum		-2147483647
	Maximum		2147483647
float	Minimum	6 digits after the decimal point	1.175494351e-38
	Maximum	6 digits after the decimal point	3.402823466e+38
single	Minimum	6 digits after the decimal point	1.175494351e-38
	Maximum	6 digits after the decimal point	3.402823266e+38
double	Minimum	14 digits after the decimal point	2.2250738585072014e-308
	Maximum	14 digits after the decimal point	1.7976931348623157e+308
string	Minimum		Length is zero
	Maximum		Length is unlimited

1.10 Error messages

When the mismatching of syntax is found in a UDF formatting file, in order to show the positions of errors in the file, the following error messages are displayed on a standard output or a dialog.

A UDF file name is displayed on "filename" in the error messages. The line number of the neighborhood of a mismatching is displayed on "line 99" in the error messages.

(1)Errors about the header part

•Data type other than string is specified.
[error 1]filename:line 99 near "int":no supported type in header.

(2)Errors about the data name definition part

•The description of a definition is wrong.
[error 1]filename:line 99 near ";":parse error.
(Example)
aaa;int

•More than 2-dimensional array was defined.
[error 1]filename:line 99 near ":":array is limitted in dimension 1.

•A data name is the same as a data type, when using a user defined type.
[error 1]filename:line 99 near "name":Illegal data/type name.

•A string is in an array index area.
[error 1]filename:line 99 near "abc":parse error.
(Example)
xx[abc]:int

[5]Although the value of the effective number of digits in a right-hand side column can be registered from the Editor screen of GOURMET, with the interface library, a data value serves as this effective number of digits.

•Data type is wrong.
[error 1]filename:line 99 near "intt":intt.

•There is an unmatched parentheses.
[error 1]filename:line 99 near "}":parse error.

•The data definition duplicates.
[error 1]filename:line 99 near "}":Duplication of definition.
(Note) When there is a declaration in multiple lines like the definition of compound data, the error line number is the line number which the declaration including the duplicate definition completed.
(Example)
[error 1]filename:line 6 near "}":Duplication of definition.

```
1:    \begin{def}
2:    aaa:int
3:    aaa:{
4:        b1:int
5:        b2[]:int
6:    }
7:    \end{def}
```

(3)Errors about the data entity part

•Data type is wrong. (The string is written although it is an int type, etc.)
filename:line 99:Type mismatch,definition[defined_name]data[written_data].

•The parentheses is wrong.
[error 1]filename:line 99 near "}":bracket error.

•There is too many or few data sequence.
[error 1]filename:line 99 near "value":.
(Example)

```
[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}
```

•There is no data definition.
[error 1]filename:line 99 near ":":No data definition[data_name].

Chapter 2

Samples

The samples using the UDF data format are shown below.

2.1 Basic sample

The example which used basic types, array variables and simple compound data is shown.

```
\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}
```

The example of data using KEY type and ID type is shown.

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

\begin{def}
class Vector3D: { // 3D vector type
    x:float // X coordinate
    y:float // Y coordinate
    z:float // Z coordinate
};
class Vertex: { // a point type
    id :ID
    position:Vector3D // coordinate
};
class Edge: { // an edge type
    id :ID
    vertex[] :<Vertex,ID> // list of ID in the point type
```

```
};
class Face: { // a face type
    id :ID
    vertex[] :<Vertex,ID> // list of ID in the point type
};
class Cell: { // a cell type
    id :ID
    vertex[] :<Vertex,ID> // list of ID in the point type
};
class Neighbor: { // a neighboring element array type
    vertex[] :<Vertex,ID> // list of ID in the point type
    edge[] :<Edge,ID> // list of ID in the edge type
    face[] :<Face,ID> // list of ID in the face type
    cell[] :<Cell,ID> // list of ID in the cell type
};
class MeshAxis:{
    values[] : double
};
class Mesh:{ // Mesh type
    name:KEY // name of the mesh
    type:string // type of the mesh
    axes[]:MeshAxis // array of the mesh axes
    periodic[]:int // array of the periodic boundary condition
};
class MeshData : {
    name : <Mesh,KEY>
    vertex[] : Vertex
    edge[] : Edge
    face[] : Face
    cell[] : Cell
    neighbor_of_a_vertex[] : Neighbor
    neighbor_of_a_edge[] : Neighbor
    neighbor_of_a_face[] : Neighbor
    neighbor_of_a_cell[] : Neighbor
};
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}
```

2.2 Sample using the select type

If a selected type is used, the data input operations will be simplified in the edit display of GOURMET.

```
\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"}, // Selection from Dymanics/Minimize
    Dynamics: {
      Dynamics_Algorithm:select{ // variety of Dymanics
        "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:{ // Input items required after selection
        Q:float
      }
      NVT_Berendsen:{ // Input items required after selection
        tau_T:float
      }
      NVT_Kremer_Grest:{ // Input items required after selection
        Friction:float
      }
      NPH_Andersen:{ // Input items required after selection
        Cell_Mass:float
      }
      NPH_Parrinello_Rahman:{ // Input items required after selection
        Cell_Mass:float,
        Fix_Angle:short,
        Fix_Cell_Length:string
      }
      NPH_Brown_Clarke:{ // Input items required after selection
        tau_P:float,
        Fix_Angle:short,
        Fix_Cell_Length:string
      }
      NPT_Andersen_Nose_Hoover:{ // Input items required after selection
        Cell_Mass:float,
        Q:float
      }
      NPT_Andersen_Kremer_Grest:{ // Input items required after selection
```

```
    Cell_Mass:float,
    Friction:float
  }
  NPT_Parrinello_Rahman_Nose_Hoover:{ // Input items required after selection
    Cell_Mass:float
    Fix_Angle:short
    Fix_Cell_Length:string
    Q:float
  }
  NPT_Parrinello_Rahman_Kremer_Grest:{ // Input items required after selection
    Cell_Mass:float
    Fix_Angle:short
    Fix_Cell_Length:string
    Friction:float
  }
  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
    Fix_Cell_Length:string
    tau_T:float             // Coupling constant of heat bath
  }
  SLLOD_PT_Const:{
    tau_P:float            // Coupling constant of pressure bath
    Fix_Cell_Length:string
  }
}
Minimize:{
  Minimize_Algorithm:select{  // The variety of Minimize algorithm
    "SD",                     // steepest descent
    "CG",                     // conjugate gradient
    "Cascade",                // combination of SD and CG
    "MSI"
  }
  SD:{
    Max_Iteration:int,
    Converge_Force:float
    Output_Interval_Steps:int,
  }
  CG:{
    Max_Iteration:int,
    Converge_Force:float
    Output_Interval_Steps:int,
  }
  Cascade:{
    SD_Max_Iteration:int,
    SD_Converge_Force:float
  }
  MSI:{
    Constraint:short
    Min_Coord:float
    Max_Coord:float
    MF_Parameter:float
    MO_Parameter:float
```

```
      }
    }
  }

  Boundary_Conditions:{
    a_axis:string
    b_axis:string
    c_axis:string
    Periodic_Bond:short
  }
  Output_Flags:{
    Statistics:{
      Energy:short,
      Temperature:short,
      Pressure:short,
      Stress:short,
      Volume:short,
      Density:short,
      Cell:short,
      Wall_Pressure:short
      Energy_Flow:short
    }
    Structure:{
      Position:short
      Velocity:short
      Force:short
    }
  }
  Fix_Node[]:{
    Mol_Index:int
    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}
```

Chapter 3

Supplement about the unit of UDF

The dimension of a unit and the system of units are explained here. You can generate a new unit from the nine primary unit, which are shown below. The new unit has the numbers of powers of each primary unit, that is the dimension of the unit.

3.1 Dimension of the unit

In UDF, all units consist of nine kinds of following SI primary units.

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

The generated unit has nine numbers of dimensions. The units defined as follows have the numbers of dimensions of units shown in the table 3.1.

```
\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}
```

Unit name	M(Note [1])	L	T	E	TT	A	LI	PA	SA
PI(Constant)	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

Table 3.1: The dimensional numbers of units

3.2 The formatting of a unit system specification file

The unit system used here is a set of the units used in order to display data values. When a UDF file including unit definitions is read into GOURMET, the data values are displayed based on the defined units.

If a unit system is specified, the dimension of the unit will be compared and the values will be changed into the units described in the unit system all at once.

The formatting of a unit system specification file is as follows. The definition of one unit is described in one line.

\begin{unit_system}{"unit_system_name"}
New_Constant = CONSTANT
[New_Unit] = [Unit_Expression]
[New_Unit] = CONSTANT[Unit_Expression]
[Unit_Expression]
\end{unit_system}

The example of specification of SI unit is shown below.

[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)

```
\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 conctant
[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}
```

If the combination of the units which is not included in the newly specified unit system definitions is used in a UDF data file, a corresponding unit name is generated automatically by combining the primary unit included in the new unit system definitions. If the unit equivalent to the primary unit is not described in the unit system definitions, the corresponding SI primary unit is used for the automatic generation of the unit name.

The units with the following unit dimensions are arranged into the unit names with a near dimensional

number as follows.

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

That is, `[kg*m^2/s]` shown with the primary units transposes to $[J * s]$.

About the unit with the same dimension as $[Pa]$, even if the dimensions of units is length = -1, mass = 1 and time = -2±1, it transposes to a unit with the same dimension as $[Pa]$ in the unit system definitions.

Also about $[N]$, it is similarly transposed to the unit with length=1±1,time=-2±1 has the same dimension as $[N]$.

For example, `[kg*m/s^3]` is transposed to `[N/s]`.