OCTA

Integrated simulation system for soft materials

GOURMET Primer

TUTORIAL

Version 1.1

OCTA User ’s Group

DEC. 25 2002

Authors of the Manual

Part I Yuzo Nishio, The Japan Research Institute, Limited, Masao Doi, Nagoya University
Part II Masao Doi, Nagoya University

Program Developers

Yuzo Nishio The Japan Research Institute, Limited

Acknowledgment

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.

Copyright ©2000-2002 OCTA Licensing Committee All rights reserved.

Contents

I Scripting GOURMET **1**

1 Introduction: baseball project **3**

2 Getting started **7**
2.1 Starting GOURMET . . . 7
2.2 Using Editor . . . 7
2.3 Running python script . . . 11
2.4 Using Viewer . . . 11

3 Theoretical background **15**

4 Data structure **17**
4.1 Unit definition . . . 17
4.2 Class definition . . . 17
4.3 Data definition part of baseball simulator . . . 17
4.3.1 Definition for constant data . . . 17
4.3.2 Definition for calculated results . . . 19
4.4 Data part of baseball simulator . . . 19

5 Calculation **21**
5.1 Loading python script . . . 21
5.1.1 Python script for calculation . . . 21
5.2 Running the script for calculation . . . 24
5.3 Viewing results . . . 24
5.3.1 Python script for 3D object viewer . . . 25
5.4 Running the script for 3D object drawing . . . 26

6 Drawing graphs **27**
6.1 Using GraphSheet . . . 27
6.1.1 Using the plot tool . . . 27
6.2 Plotting graphs by python . . . 31

7 Action **33**
7.1 What is action . . . 33
7.2 Kicking action . . . 33
7.3 Defining action . . . 35
7.3.1 Example of defining actions . . . 35
7.4 Actions of baseball simulator . . . 36
7.4.1 Basic actions . . . 36
7.4.2 Input action-simple . . . 38
7.4.3 Input action-detailed . . . 39

8 More and more **41**
8.1 Golf simulator . . . 41
8.2 Other simulator to be explored . . . 43
8.2.1 Tabletennis simulator . . . 43
8.2.2 Tennis simulator . . . 43
8.3 What now . . . 43

II Programing UDF **47**

9 Introduction **49**

10 Basic operation **51**
10.1 Simple data . . . 51
10.1.1 Simple example of UDF file . . . 51
10.1.2 Reading a UDF file in C++ program . . . 52
10.1.3 Writing a UDF file in C++ program . . . 53
10.2 Structure type . . . 54
10.2.1 Definition of structure type . . . 54
10.2.2 Data part of structured data . . . 54
10.2.3 Reading structured data in C++ . . . 55
10.2.4 Writing structured data in C++ . . . 56
10.3 Unit . . . 56
10.4 Record . . . 57

11 Class **63**
11.1 Introduction . . . 63
11.2 UDF class . . . 63
11.2.1 Reading the structured data into C++ class object . . . 64

12 Pointer expression **67**
12.1 Introduction . . . 67
12.2 KEY . . . 68
12.3 ID . . . 68
12.4 Reading in C++ . . . 69
12.5 Other functions related to ID and KEY . . . 70

13 Handling UDF file by python **71**
13.1 The $ prefix . . . 71

14 Documents for GOURMET **73**

References **75**

15 Minimum reference of Python in GOURMET **77**
15.1 Variable . . . 77
15.2 Tuple, list, dictionary . . . 77
15.3 Control statements . . . 78
15.4 Function and class . . . 78
15.5 Module . . . 79
15.6 UDFManager in GOURMET . . . 79
15.6.1 Summary of UDFManager methods . . . 80
15.6.2 Summary of Drawing methods . . . 81
15.7 Bibliography . . . 81

List of Figures

1.1 Parabola drawing: First step of using GOURMET 4
1.2 Calculated results of Fast and Curve ball 5
1.3 2D plot 5
1.4 Golf Simulator 6
1.5 Tableteniss Simulator 6

2.1 Parabola UDF opened in GOURMET Editor 7
2.2 Parabola UDF opened all structures 9
2.3 Parabola UDF editing by Table view 10
2.4 Unit dialog for MaxT 10
2.5 Opening 3D Viewer in GOURMET 12
2.6 Simple drawing on 3D Viewer 13
2.7 Parabola drawing 14

3.1 Forces committed on a ball 15

5.1 View of calculated results 24
5.2 View of 3D Object drawing 26

6.1 Adding time series data to GraphSheet 28
6.2 Making plot command from GraphSheet 29
6.3 gnuplot window executed by plot tool simply 30
6.4 gnuplot window executed by plot tool 30
6.5 gnuplot window executed by python script using plot library 31

7.1 View of simulated Curve ball by Action 34
7.2 UDF header dialog 35
7.3 setDetailCondition Argument Dialog 39

8.1 setSimpleCondition Argument Dialog of GolfInput 41
8.2 setDetailCondition Argument Dialog of GolfInput 41
8.3 Which club to use? 43
8.4 3D result for Tabletennis servicing 44
8.5 3D result for Tennis servicing 45

10.1 UDF data-hierarchy in the structured type variable 56

11.1 The data structure for the file ”file3.udf” 64

List of Tables

2.1 Parabola.udf 8
2.2 Parabola.py 14

4.1 Unit definition 17
4.2 Class definition 18
4.3 Definition of constant data 18
4.4 Calculated_results object definition 19
4.5 Baseball data 20

5.1 python script file for calculation 21
5.2 python script file for 3D object Viewer 25

6.1 python script file for adding time series data to GraphSheet 27
6.2 python script file for plotting Forces vs. time 32

7.1 Basic actions 36
7.2 setSimpleCondition action 38
7.3 setDetailCondition action 40

8.1 Actions of GolfInput 42

10.1 Baseball simulator definition and data 59

Part I

Scripting GOURMET

Chapter 1

Introduction: baseball project

The purpose of this booklet is to give you an introduction of what you can do with GOURMET. GOURMET is a simulation platform developed in the OCTA project. If you want to use the simulation engines developed in the OCTA project, you need to learn how to use GOURMET. With GOURMET, you can create input files for the engines, operate simulation engines, browse output files, do data analysis, plot data, and make 3D animations etc.

GOURMET is designed to be flexible and customizable. GOURMET offers you various ways of modifying the system. You can create your own interface to the engines, add your own functions for data analysis and data viewing.

These service of GOURMET is not restricted to the engines developed in the project. GOURMET can be used as an interface for any engines. The only requirement for the engines to have the GOURMET interface is to write the input and output files in a certain text file format, called UDF (user definable format). GOURMET will be useful for you to develop new engines. In fact, using GOURMET, you can easily create simple programs which has functions of data editing and browsing, data plotting and 3D graphics.

In the first part, we would like to demonstrate this to you by conducting a small project; the project of making a baseball simulator. We will make a simulator which calculates the trajectory of a ball thrown in the air, and show various functions implemented in GOURMET.

Fig. 1.1 shows the output of our simple simulator which you can make after you read next 7 pages. Here what is drawn is the trajectory of a ball when the aerodynamic effect is completely ignored. In reality, the whole trick of baseball pitching utilizes the aerodynamic force. In the following chapters, we will make a simulator which calculate the trajectory of a thrown ball taking into account of the spinning of the ball, and the bouncing of the ball on the ground. The output is shown in Fig. 1.2, and Fig. 1.3). You can modify this simulator to make a golf-ball simulator (see Fig.1.4) or a tabletennis simulator (see Fig.1.5).

We hope you will enjoy this project.

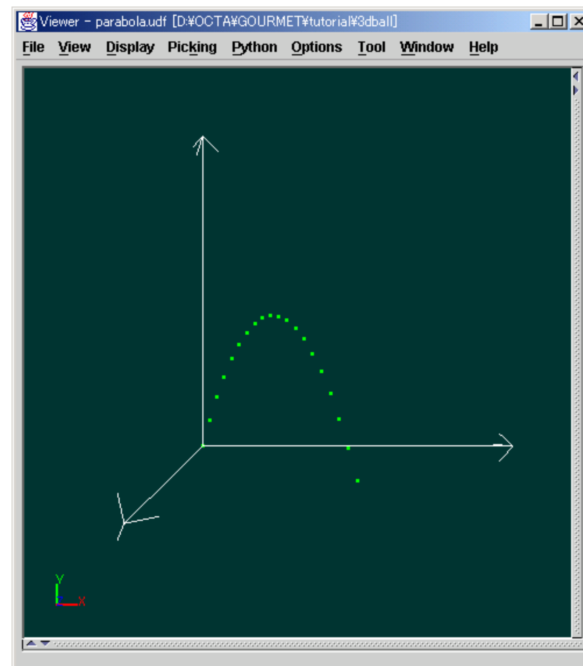


Figure 1.1: Parabola drawing: First step of using GOURMET Viewer

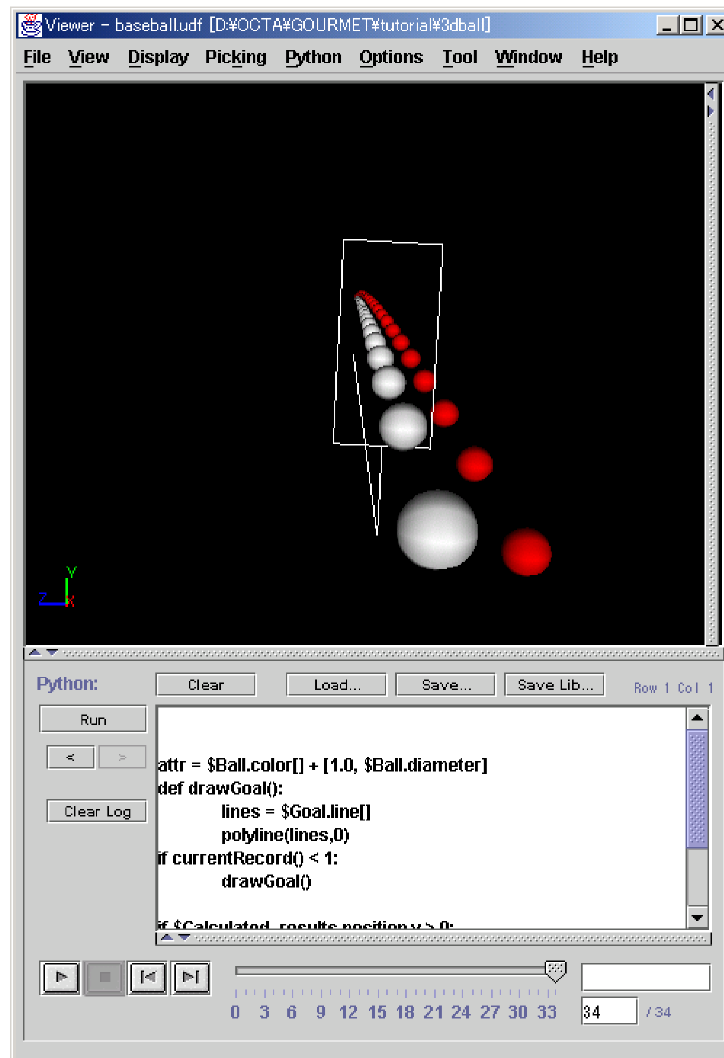


Figure 1.2: Calculated results of Fast and Curve ball

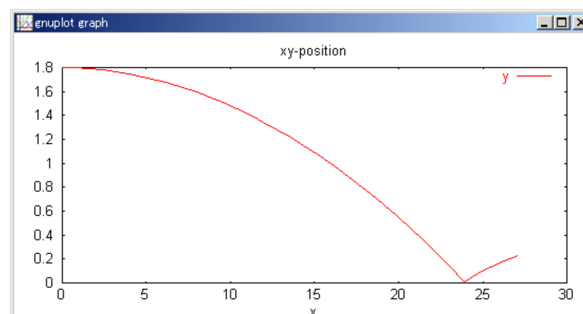


Figure 1.3: 2D x-y position of Fast ball

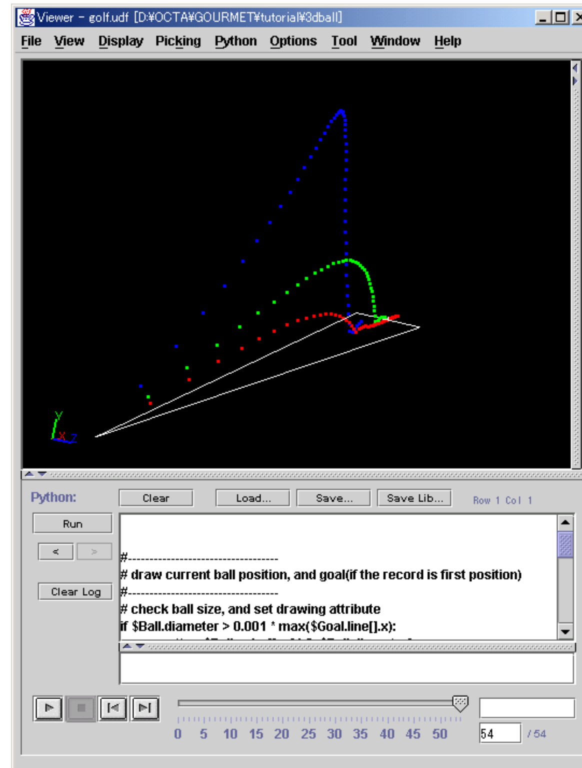


Figure 1.4: 3D results calculated by Golf Simulator

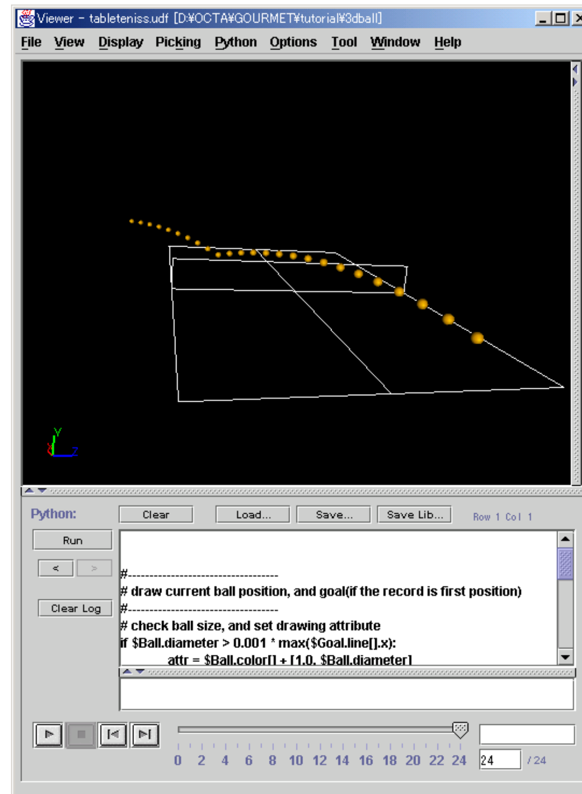


Figure 1.5: 3D results calculated by Tableteniss Simulator

Chapter 2

Getting started

2.1 Starting GOURMET

Let us start our baseball project by first getting GOURMET running. If you are using linux, execute the following command.

```
% /home/OCTA/GOURMET/bin/gourmet.sh
```

If you are using windows, execute the following command.

```
C:\> C:\OCTA\GOURMET\bin\gourmet.bat
```

In both cases, make sure to use the path to the directory where your OCTA system is installed.

2.2 Using Editor

When you start GOURMET, you will see the Editor window of GOURMET. GOURMET requires you to input some data file. So select the **File/Open...** menu at the top, and open the UDF file named "GOURMET/tutorial/3dball/parabola.udf". (Note that all files in this chapter are in the directory "GOURMET/tutorial/3dball".) You will see the window shown in Fig. 2.1.

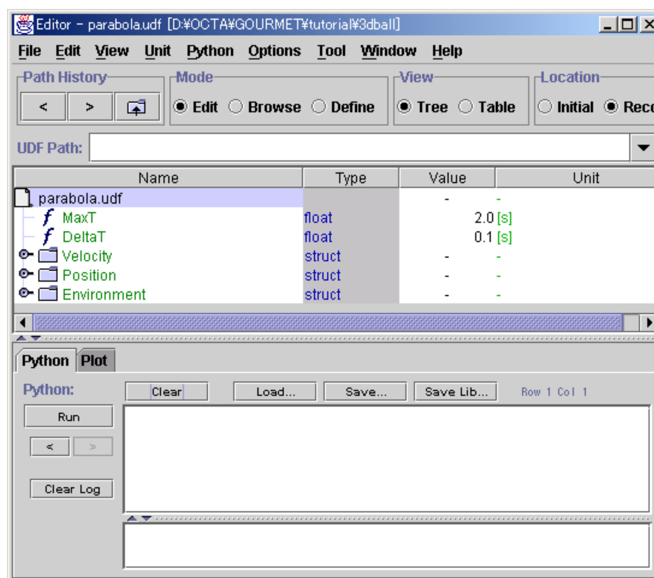


Figure 2.1: Parabola UDF opened in GOURMET Editor

The file you just opened "parabola.udf" is a text file. So let us open this file by a text editor. You will see the text shown in Table 2.1. This is the UDF(User Definable Format) file.

Table 2.1: Parabola.udf

```
// definition part of parabola data
\begin{def}
MaxT:float [s] "max of calculation time"
DeltaT:float [s] "delta of calculation time"
Velocity: {
     x:float [m/s] "initial x-velocity of the ball"
     y:float [m/s] "initial y-velocity of the ball"
     z:float [m/s] "initial z-velocity of the ball"
} "initial velocity of the ball"
Position: {
     x:float [m] "initial x-coodinate of the ball"
     y:float [m] "initial y-coodinate of the ball"
     z:float [m] "initial z-coodinate of the ball"
} "initial position of the ball"
Environment:{
  gravity:float [m/s^2] "gravity of environment"
} "environment parameter"
Result[]:{
    time:float [s]
    x:float [m]
    y:float [m]
    z:float [m]
   }
\end{def}

// data part for parabola
\begin{data}
MaxT:2.0000000
DeltaT:0.1000000
Velocity:{5.0000000,18.000000,3.0000000}
Position:{0.0,0.0,0.0}
Environment:{9.8000002}
\end{data}
```

The UDF file has a part of data definition and a part of data itself. The definition part is written between `\begin{def}` and `\end{def}`, and defines the name and units of the data. The data part is written between `\begin{data}` and `\end{data}`, and gives the actual data.

For example the definition

```
MaxT:float [s] "max of calculation time"
```

means that MaxT is a data of float type, having the unit of [s] (second). These information is shown explicitly in the data window of GOURMET. The last part of the data definition, `"max of calculation time"` is a help message. Place your curor on top of the data name MaxT. You will see the message "max of calculation time" popping up.

The data part gives the actual data. The definition part `Matx:float [s]` and the data part `MaxT:2.0000` together indicates that the `MaxT` is 2.0 [s]. This is seen explicitly in the window of GOURMET.

The definition part of "parabola.udf" tells that the next data, `Velocity` consists of three components, x,y,z each having unit [m/s]. If your window is like that shown in Fig.2.1, click the item `Velocity` on the left.

Then the item opens, and you can see the data of `Velocity`. You will see that the value of the component "x" of the "Velocity" is 5.0[m/s], and that this is indeed consistent with the data shown in Table 2.1. Fig. 2.2 shows the result when you opened all structures.

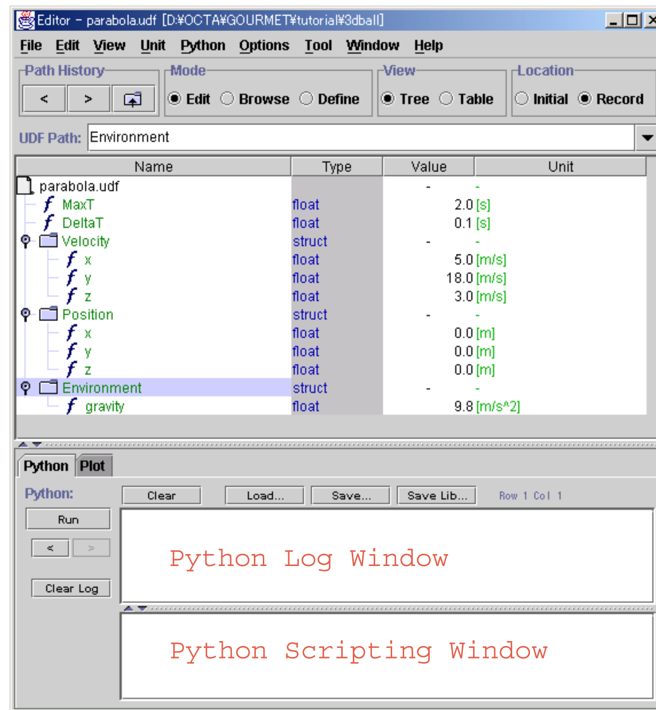


Figure 2.2: Parabola UDF opened all structures

In the Edit mode, you can edit the data content. You can use the standard operation of copy and paste by the key of [Ctrl]+C and [Cntrl]+V. Select any number or string in the window and press [Ctrl]+C. The data is then copied to the clip-board. You can paste the data in any application by [Cntrl]+V. Conversely the text data in the clipboard can be pasted into the GURMET window. Notice that you can select a multiple data in a square region, and can paste it into the other application.

Next check the button named **Table** below the menu bar. You can see the data in table form (see Fig. 2.3) Try the operation of copy and paste there.

To see the result of the editing, save the file in a different name (say "parabola_1.udf"), using **Save As** menu in **File**. Then open parabola_1.udf by some text editor, and check that your change is actually reflected.

In the Edit mode, you cannot change the data definition. To change the data definition, you have to edit the UDF file directly by a text editor, or use the Define mode. However, such operation should be done when you know more about UDF and GOURMET.

You can see the data in a different unit. Move your cursor to the field of unit in the row of MaxT in the GOURMET window, and press the right button. You will see the box shown in Fig. 2.4. Press the button at the right end of the popup window, and choose the unit, say [ms]. You will see that MaxT is now displayed as 2000.0 [ms].

Changing the unit is effective only in the display. Internally, the data is stored in the unit defined in the definition part. To see this, change MaxT from 2000.0 [ms] to 4000.0 [ms], and save the file. Check that the data part is written as `MaxT:4.0`.

GOURMET knows simple unit transformation (like the transformation from [J] (Joule) to [erg]). You can define new unit such as [cal], [atomic_mass], [eV] etc. See **GOURMET Operations Manual** for details.

Note: If you want to return initial state, re-open "parabola.udf" using **File/Open** menu.

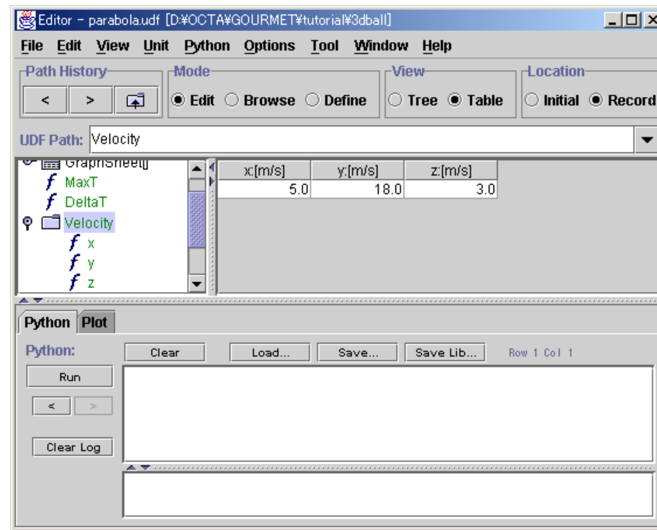


Figure 2.3: Parabola UDF editing by Table view

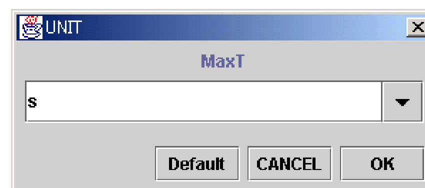


Figure 2.4: Unit dialog for MaxT

2.3 Running python script

At the bottom of the Edit window, there are **Python Scripting Window** and **Python Log Window**. You can run any python script in **Python Scripting Window** and see the result in **Python Log Window**. As a test,type the following script in Python Scripting Window and push the **Run** button.

```
a = 123.0 / 23
print a
```

You will see the result 5.34782608696 in Python Log Window.

Next, run the following script.

```
print $MaxT
```

You will see the result 2.0 in Python Log Window, if you did not modify the value. Here, `$MaxT` means the reference of the data "MaxT" in the UDF file which you have opened.

Also you can assign a value to any UDF data. Run the following script. Here, `$Velocity.z` is the data component "z" in the compound data "Velocity".

```
$Velocity.z = 4.0
```

Confirm the result by seeing it in the data view or executing `print $Velocity.z` in Python Scripting Window.

Now try the following script.

```
V= $Velocity
print V
print V[0], V[1]
```

You will see the print out

```
[5.0,18.0, 3.0]
5.0  18.0
```

Here `[5.0,18.0, 3.0]` is a way of writing a list of data in python. Each component in the list is referred to in the C++ manner, like `V[0]`,`V[1]`. Conversely, you can set the value of $Velocity by the follwoing srcipt.

```
$Velocity = [-1.0, -1.0, -1.0]
```

2.4 Using Viewer

Let us now try the Viewer of GOURMET. If you select **Window/Viewer** menu in Editor, you will see the blank Viewer shown in Fig. 2.5. Viwer has three windows; 3D Object Window, Python Scripting Window and Python Log Window.

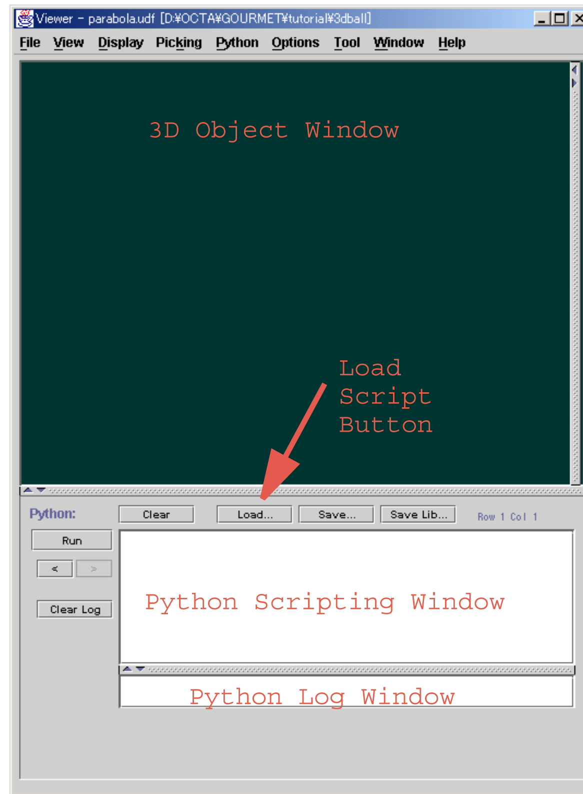


Figure 2.5: Opening 3D Viewer in GOURMET

Now, try the follwoing script in Python Scripting Window. You will see the window as it is shown in Fig. 2.6.

```
r0=[0,0,0]
r1=[1,0,0]
r2=[0,1,0]
r3=[0,0,1]
sphere (r0, 0)
arrow (r0, r1, 0)
line(r0,r3,0)
cylinder(r0,r2, 0)
text (r0, "Origin", 0)
```

The script `sphere (r0,0)` draws a sphere at the position `r0`. The last argument 0 is an attribute number which represents the attributes of the sphere, like radius and color. To draw spheres in different color, change the attribute number (try `sphere (r0,1)`). To draw a sphere with different diameter, see chapter 15.

The meaning of the other command will be obvious; `arrow (r0,r1,0)` draws an arrow starting from `r0` and ending at `r1` with the attribute number 0. `text(r0,"Origin",0)` add a text "Origin" in your viewing window starting at `r0` with the text attribute code 0. Other convenient commmands can be found in chapter15, and in **GOURMET Python Script Manual**.

Next, load the drawing script by **Load...** Button.
Choose the script "GOURMET/tutorial/3dball/script/parabola.py" in the file open dialog. You will see the loaded script in Python Scripting Window. The content of the script is shown in Table. 2.2.

The line starting with the character'#' is a comment line. This script draws three orthogonal axes from the origin (line 3 to 6), calculates the ball position at each time step(line 12 to 14), and shows the position (line 16) while the time is less than `MaxT`.

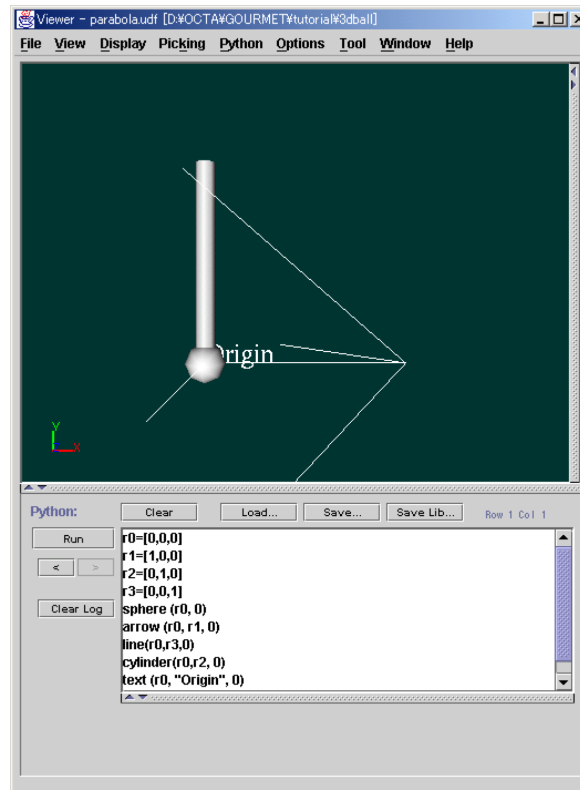


Figure 2.6: Simple drawing on 3D Viewer

Run the script by **Run** Button. You will see the 3D parabola as it is shown in Fig. 2.7. You can rotate the 3D object by **dragging with left mouse button**, and reset or set the view direction by **selecting View menu**. You can draw other parabolas by changing any parameters in "parabola.udf" by Editor and run the drawing script in the Viewer. Notice that the Editor and the Viewer are handling the same data:the change in one is reflected in the other. If you want to save the modified parameters, choose **File/SaveAs...** menu in Editor.

Now, you have learned the basic operations of GOURMET. Let's start to learn how to design and build our baseball simulator!

Table 2.2: Parabola.py

```
 1: # Python script file for parabola Viewer
 2: # draw axis
 3: max = 20
 4: arrow([0,0,0],[max,0,0], 0)
 5: arrow([0,0,0],[0,max,0], 0)
 6: arrow([0,0,0],[0,0,max], 0)
 7: #disk([0,0,0],[1,1,0,1,max,0,1,0])
 8: # draw ball
 9: t = 0.0
10: while t < $MaxT:
11: # calculate coodinates of the ball
12:     x = $Position.x + $Velocity.x * t
13:     y = $Position.y + ($Velocity.y  - 0.5*$Environment.gravity * t) * t
14:     z = $Position.z + $Velocity.z*t
15: # draw position
16:     point([x,y,z], 2)
17: # increment time
18:     t =  t + $DeltaT
19: #----------- end of script --------------
```

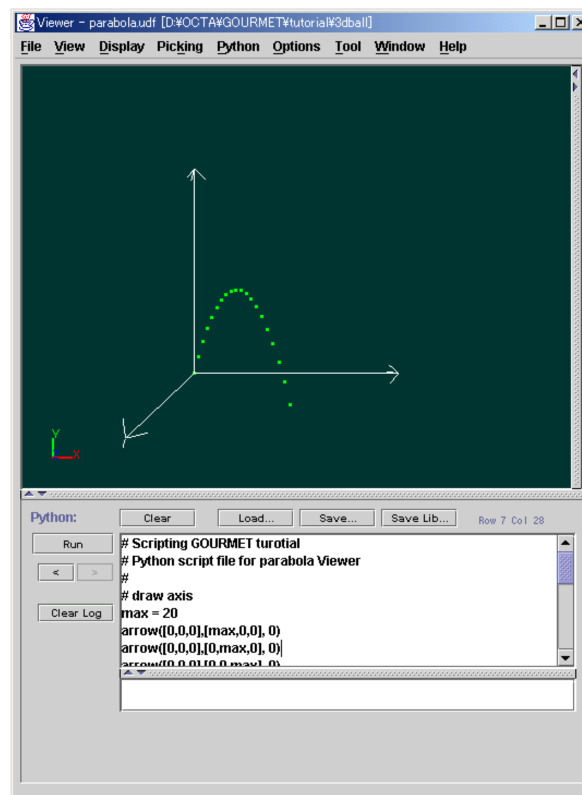


Figure 2.7: Parabola drawing

Chapter 3

Theoretical background

Let us now summarizes the theoretical background of our baseball simulator.

Consider a ball of mass m and radius a thrown in the air. Let us take a cartesian coordinate as it is shown in Fig. 3.1: y axis is taken in the vertical direction, and x axis is taken so that the initial velocity of the ball is in the x-y plane, and z axis is taken normal to x and y axes.

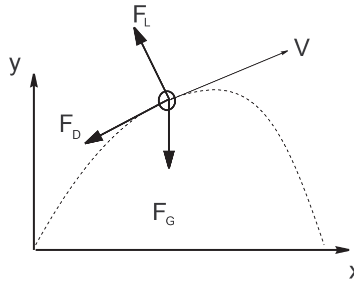


Figure 3.1: Forces committed on a ball

The velocity of the ball $\boldsymbol{V}$ is determined by the following equation of motion.

$$m\dot{\boldsymbol{V}} = \boldsymbol{F} - mg\boldsymbol{e}_y \tag{3.1}$$

where $\boldsymbol{F}$ is the aerodynamics force acting on the ball, and the second term represents the effect of gravity.

We use a simple approximation for the the aerodynamic force. First, consider a simple situation that the ball is moving in the x-direction with the speed V. The air exerts a drag on the ball, which can be written as

$$F_D = \frac{1}{2} C_D \rho V^2 A \tag{3.2}$$

where C_D is the drag coefficient, and $A = \pi a^2$ is the cross section of the ball. The drag force is acting in the negative direction of x axis.

If the ball is rotating around the z-direction with angular velocity ω, the upper part is moving with the speed $V + a\omega$, while the bottom part is moving with the speed $V - a\omega$. Since the air pressure at the surface is proportional to the square of the surface speed, the difference in the surface speed creates a lift force, which is proportional to $(V + a\omega_z)^2 - (V - a\omega_z)^2 = 4Va\omega_z$. Thus we may write the lift force as

$$F_L = 2C_L \rho V a\omega_z A \tag{3.3}$$

where C_L is the lift force coefficient. The lift fore is actikng in the direction of y axis.

In general, if the ball velocity is $\boldsymbol{V}$, and if the angular velocity of the ball is $bm\omega$, the aerodynamic force is written as

$$\boldsymbol{F} = -\frac{1}{2} C_D \rho V \boldsymbol{V} A + 2C_L \rho \boldsymbol{V} \times a\boldsymbol{\omega} A \tag{3.4}$$

The rotation of the ball will change in time, but, we shall assume that it is constant in the present calculation.

Thus the equation of motion for the position $\boldsymbol{r}$ is written as

$$\dot{\boldsymbol{r}} = \boldsymbol{V} \tag{3.5}$$

$$m\dot{\boldsymbol{V}} = -\frac{1}{2}C_D\rho V\boldsymbol{V}A + 2C_L\rho\boldsymbol{V}\times a\boldsymbol{\omega}A - mg\boldsymbol{e}_y \tag{3.6}$$

The equation must be supplemented by the condition at the ground. We simply assume that the ball will be bounced back if it hits on the ground. This is written as

$$V_y' = -C_R V_y \qquad \text{when} \quad y < 0 \tag{3.7}$$

Here C_R is the reflection coefficient of the ball at the ground.

Chapter 4

Data structure

Now let us construct the UDF file for our baseball simulator. The final UDF definition in this tutorial is in "GOURMET/tutorial/3dball/3dball.def", but we encourage you to type in the definition using some text editor.

4.1 Unit definition

First, we have to define the unit used in our simulator. We use SI unit system in our simulator. Since SI is already implemented in GOURMET, there is no need to define it. However, for those who would like to see the distance in miles or yard, let us give a definition for them. The definition is shown Table 4.1.

Table 4.1: Unit definition

```
\begin{unit}
     PI=3.141592
     [rps]=2.0*PI[rad/s]
     [rpm]=1.0/60.0*[rps]
     [yard]=0.9144*[m]
\end{unit}
```

The definition is straightforward. As you can see in the definition of `[rpm]`, you can use the unit which has been already defined. The rule of defining the unit is given in **UDF Syntax Reference**.

4.2 Class definition

To represent vector quantities, such as position, velocity, force, it is convenient to define a 3-dimensional vector class. This can be done as it is shown in Table 4.2.

This class defines a quantity which has three components `x,y,z` all having the same unit specified by `[unit]` at the end of the definition. Once such class is defined, you can give the following definition:

```
velocity :Vector3D [m/s]
force    :Vector3D [kg m/s^2]
```

Notice that the [unit] attatched at the end of the class determines the unit of each data component.

4.3 Data definition part of baseball simulator

4.3.1 Definition for constant data

Let us now define the physical quantities which will appear in our baseball simulator. The data definition is written between `\begin{def}` and `\end{def}` or between `\begin{global_def}` and `\end{global_def}`.

Table 4.2: Class definition

```
\begin{def}
    class Vector3D:{
      x:float [unit]
      y:float [unit]
      z:float [unit]
    } [unit]
\end{def}
```

The distinction will be explained later. Crudely speaking, use `global_def` for the data which will remain constant throughout the calculation, and use `def` for the data which varies as calculation goes on. (In fact, `global_def` is introduced recently as an option for pursuing the data access speed. Your program wll work, even if you do not discriminate them.)

Here we define the quantities which remains constant throughtout the calculation in `global_def` The definition is given in Table 4.3.

Table 4.3: Definition of constant data

```
\begin{global_def}
 Ball:{
        mass:float [kg] "mass of the ball"
        diameter:float [m] "diameter of the ball"
      } "ball attributes"
 Environment:{
        gravity:float [m/s^2] "gravity of environment"
        rho:float [kg/m^3] "density of atmosphere"
        reflection_ratio: float "reflection ratio of ground"
   } "environment parameter"
 Parameter:{
     CD:float "drag force coefficient"
     CL:float "lift force coefficient"
   } "aerodynamic coefficients"
 Solver:{
        dt:float [s] "differential time"
        tmax:float [s] "time period for calculation"
       output_interval:float [s] "output interval for saving record"
   } "solver parameters"
 Initial_condition:{
     position:Vector3D [m] "initial position of the ball"
     velocity:Vector3D [m/s] "initial velocity of the ball"
     spin:Vector3D [m/s] "surface velocity caused by spin"
   } "initial condition of calculation"
\end{global_def}
```

The meaning of the definition will be clear.

- The data `Ball` stands for the mass and the diameter of the ball.
- The data `Environment` stands for the data concerning the environment of the ball. This includes gravity , air density, and the reflection coefficient of the ground.

- `Parameter` includes the aerodynamic coefficients C_D and C_L.
- `Solver` stands for the parameters needed in the numerical analysis such as the time step of integration, the time interval between the output etc.
- `Initial_condition` stands for the initial position, velocity, and spin of the ball. Notice that the definition is done in a simple way using the Vector3D class introduced in Subsection 4.2.

4.3.2 Definition for calculated results

The results of our baseball simulator is a time evolution of the position and the velocity of the ball. GOURMET offers you a service to move through the time evolution of the system. This is to write the system state for each "record". The "record" in the UDF file is a set of data corresponding to a certain time step (or a certain value of a parameter which is being varied, like the load in the stress analysis). To write a data of current state, you create a new record and output the current state, and then advance the time step. The data defined between `\begin{def}` and `\end{def}` are written separately for each record. On the other hand, the data defined between `\begin{global_def}`, and `\end{global_def}` is written in a area common to all record.

To be able to see the time evolution of the ball position, we have to place the `Calculated_result` between `\begin{def}`, and `\end{def}`. If you place it between `\begin{global_def}`, and `\end{global_def}`, you will overwrite the previous result.

Thus we define the data `Calculated_result` as it is shown in Table 4.4.

Table 4.4: Calculated_results object definition

```
\begin{def}
     Calculated_results:{
       time:float [s] "time from start"
       velocity:Vector3D [m/s] "velocity at the time"
       position:Vector3D [m] "position at the time"
       force:Vector3D [N] "force at the time"
     } "calculated record at each time"
\end{def}
```

4.4 Data part of baseball simulator

We have done the definition part. Let us now create the data part. Table 4.5 shows a complete UDF file. Here instead of writing the definition part explicitly, we read the definition part from the file "3dball.def" by the insert command at the first line. When you open the file "baseball.udf" by GOURMET, the included definition is analyzed, and it is treated as if the contents of "3dball.def" were in the "baseball.udf" file from the first.

Now you have completed your UDF file. Store the file with some name, and open it by GOURMET. GOURMET will give you error messages if you have made any mistakes. You can open data items, and check whether the data is properly structured, and the naming and popup message are appropriate.

The final UDF file in this section is stored as "GOURMET/tutorial/3dball/baseball.udf". You can read the text file directly or browse by GOURMET Editor. If you see some objects not defined in this section, those will be explained in Chapter 7.

Table 4.5: Baseball data

```
\include{"3dball.def"}

// header information ...(will be explained)

\begin{data}
Environment:{9.8, 1.3, 0.5}
Goal:{
  [
    {0.0, 0.0, 0.0}
    {18.44, 0.0, 0.0}
    {18.44, 0.4, 0.0}
    {18.44, 0.4, 0.217}
    {18.44, 1.3, 0.217}
    {18.44, 1.3,-0.217}
    {18.44, 0.4,-0.217}
    {18.44, 0.4, 0.0}
  ]
}
Ball:{
  0.1445, 7.15e-2,
  [1.0, 1.0, 1.0]
}
Solver:{1.0e-2, 0.7, 2.0e-2}
Parameter:{0.35, 0.25}
Initial_condition:{{0.0, 1.8, 0.0}{45.0, 0.0, 0.0}{0.0, 8.9, 0.0}}

\end{data}
```

Chapter 5

Calculation

5.1 Loading python script

Let us now start programming using python. Open the UDF file named "GOURMET/turorial/3dball/baseball.udf". Load the python script named "GOURMET/turorial/3dball/script/calc.py" using Load buttun in Python Panel.

5.1.1 Python script for calculation

The contents of "calc.py" is shown Table 5.1. The first half of this script assigns the python variables from UDF data, and the later half calculates coordinates of the ball using the equation shown in Chapter 3.
Note: The indent of line is quite important in python script. Python parser diecriminate the space made by tab key and that made by space keys. Use tab key rather than space key to make consistent indent.

Table 5.1: python script file for calculation

```
#=============================================
# import python package for sin, cos, etc.
#-------------------------------------------
from math import *

# set python variable used calculation from UDF data
#-------------------------------------------
g = $Environment.gravity
rho = $Environment.rho
CR = $Environment.reflection_ratio
Ux, Uy, Uz = $Environment.U
A = pi * $Ball.diameter * $Ball.diameter / 4
m = $Ball.mass

# set Hydrodynamic coeficient
#-------------------------------------------
CD = $Parameter.CD
CL = $Parameter.CL
CS = $Parameter.CS

# following elegant assignment is called "tuple assignment".
# $Initial_condition.position returns list value such as [1.8, 0.0, 0.0].
# same as x = $Initial_condition.position.x; etc..
#-------------------------------------------
```

```
x, y, z = $Initial_condition.position
vx, vy, vz = $Initial_condition.velocity
sx, sy, sz = $Initial_condition.spin

# set python variavles for calculation control
#------------------------------------------
# delta time
dt = $Solver.dt
iv = $Solver.output_interval
# output by each 'num' calculation
num = int(iv/dt)
# maximum time for calculation
tmax = $Solver.tmax
# total steps of calculation
tstep = int(tmax/dt)

# repeats tstep times
#------------------------------------------
for n in range(tstep):
    t = dt * n
    if t > tmax:
        break
# velocity of the ball
    V = sqrt(vx*vx+vy*vy+vz*vz)
# Hydrodynamic forces
# equation (3.4)
    fx = -0.5*CD*rho*V*vx*A  + 2.0*CL*rho*vx*sx*A
    fy = -0.5*CD*rho*V*vy*A  + 2.0*CL*rho*vx*sy*A
    fz = -0.5*CD*rho*V*vz*A  + 2.0*CL*rho*vx*sz*A

# output log and save reslts at each num step
#----------------------------------------------
    if (n % num) == 0:
        print 'time:', t, 'position:' , [x, y, z], 'velocity:',[vx, vy, vz],
   'force:',[fx, fy, fz]
        if jump(n/num) < 0:
            newRecord()
# save results to UDF data
        $Calculated_results.time = t
        $Calculated_results.velocity = [vx, vy, vz]
        $Calculated_results.position = [x, y, z]
        $Calculated_results.force = [fx, fy, fz]
#--------------- if block end ------------------

# equation of motion
# equation (3.6)
    vx = vx + fx/m * dt
    vy = vy + (fy/m-g) * dt
    vz = vz + fz/m * dt
# equation (3.5)
    x = x + vx * dt
    y = y + vy * dt
    z = z + vz * dt
# equation (3.7)
    if y < 0:
        y = -y
        vy = -CR*vy
```

```
#-------repeat block end -----------------
# set final time reached
$Calculated_results.time = t

# return to initial record
jump(0)
#============= script end ================
```

5.2 Running the script for calculation

Now run the python script using the Run buttun in Python Panel. You will see the calculated results in Python Log Window. The result is stored in `Calculated_results`. Open the data item "Calculated_results" in the Edit window. You can see time, velocity, position and force etc. You can see the time variation of these quantities by moving your record slidebar.

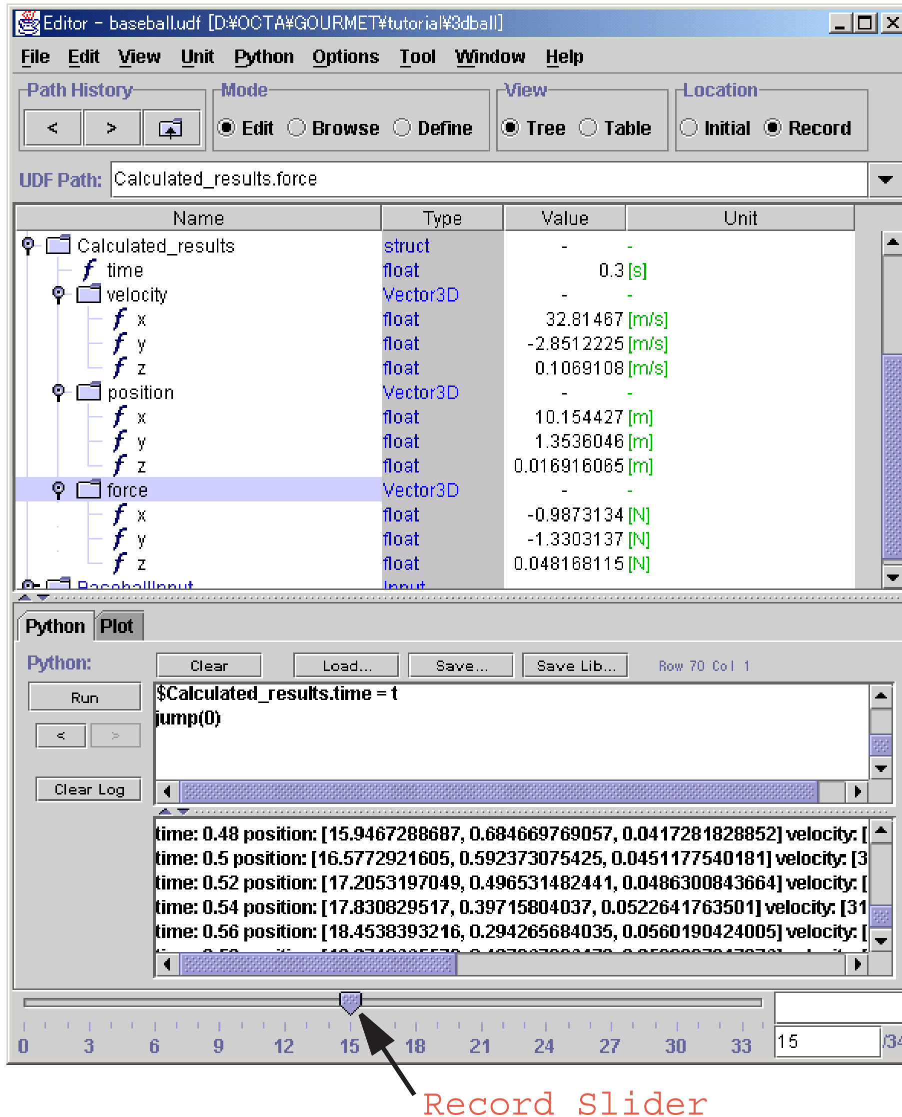


Figure 5.1: View of calculated results

5.3 Viewing results

Now you can do 3D drawing of the ball using Viewer. Select Viewer from the Window/Viewer menu, and load the python script named "GOURMET/turorial/3dball/script/show.py" in the Python Panel using the load button.

5.3.1 Python script for 3D object viewer

The content of "show.py" is shown Table 5.2. This script draws a snapshot of the ball position in the current record. The functions polyline(),sphere(),appendDraw() are the builtin drawing function of GOURMET.

Table 5.2: python script file for 3D object Viewer

```
# draw function for goal
#-------------------------------
def drawGoal():
    lines = $Goal.line[]
    polyline(lines,0)

# set drawing attribute(list of RGB color, transparency and size) for ball
#-------------------------------
attr = $Ball.color[] + [1.0, $Ball.diameter]

# set append drawing mode
# and draw goal if current record is initial
#-------------------------------
appendDraw()
if currentRecord() < 1:
    drawGoal()

# draw ball at current position
#-------------------------------
if $Calculated_results.position.y > 0:
    pos = $Calculated_results.position
    sphere(pos, attr)

#----------- end of script --------------
```

5.4 Running the script for 3D object drawing

Now run the python script using the Run buttun in Python Panel. Bring the record slider at the position 0, then you will see the strike zone and the initial position of ball. Now click the Start Animation button in the bottom left of the Viewer Window. You will get a stroboscopic view as it is shown in Fig. 5.2.

You can rotate the view by mouse dragging with the left button, and can do zooming with the right button. You can reset the view by View/Reset menu.

Note: Stroboscopic effect is made by appendDraw(append mode drawing) function.

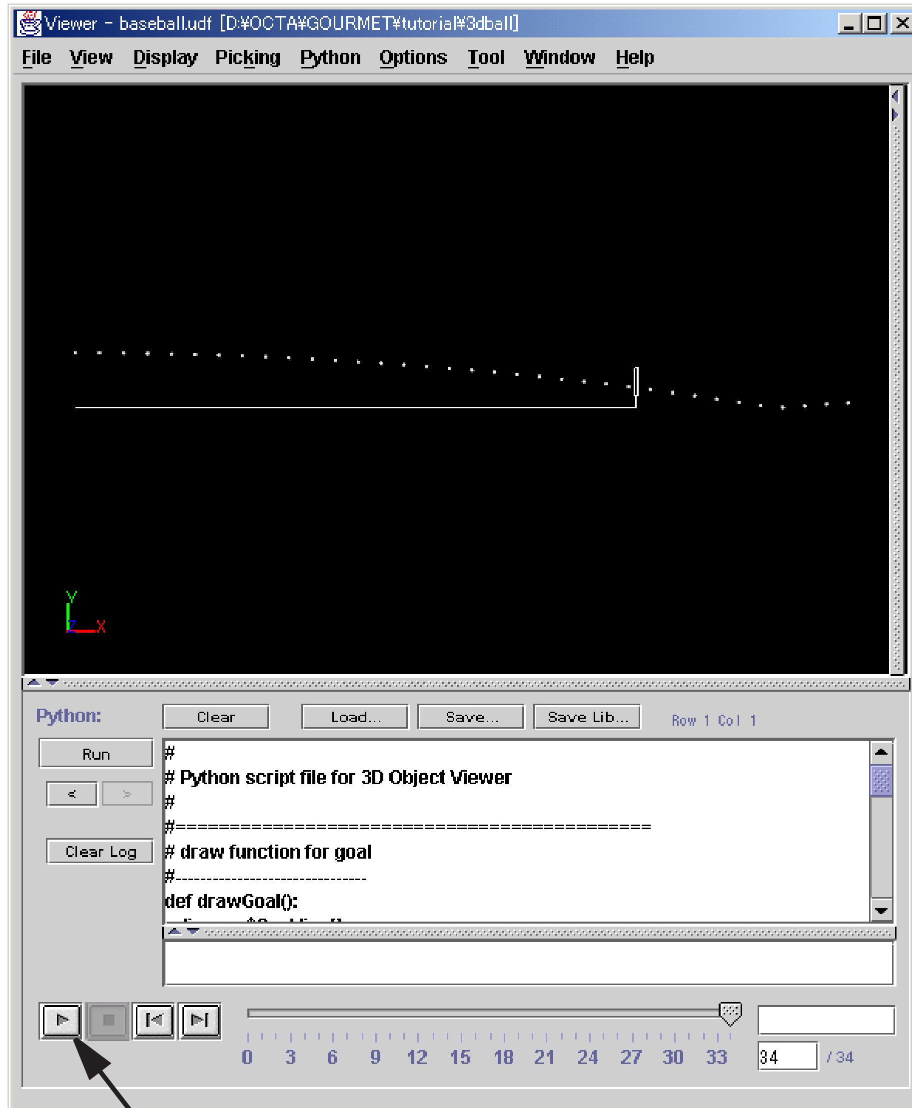


Figure 5.2: View of 3D Object drawing

Chapter 6

Drawing graphs

6.1 Using GraphSheet

You can plot your data in 2D and 3D graphs using gnuplot application. To do this, you have to make a table of the data to be plotted first. GOURMET helps you to make a table of plot data by the special worksheet named GraphSheet in **Table View of Editor**. To utilize this, go back to Editor, and load the python script named "GOURMET/turorial/3dball/script/addplot.py". The content of "addplot.py" is shown in Table 6.1. This script adds a column named 'x' to GraphSheet Object, and adds the data of the x-coordinate calculated by all records to the column.

Run the script and see that the content of the **GraphSheet** has been changed as it is shown in Fig. 6.1.

Table 6.1: python script file for adding time series data to GraphSheet

```
colname='x'
dataname='Calculated_results.position.x'

#-----------------------------------
# add a column to GraphSheet Object
#-----------------------------------
createSheetCol(getSheetColSize(), colname, totalRecord())

#-----------------------------------
# add the data in all records to the column of GraphSheet Object
#-----------------------------------
for rec in range(totalRecord()):
    jump(rec)
    setSheetData(colname, rec, dataname)
```

Similarly, add the data for y-coordinate and z-coordinate to GraphSheet Object,by changing 'colname' and 'dataname' in the script. Now you are ready to use the plot tool.

6.1.1 Using the plot tool

First press the **Plot Tab** next to the **Python Tab** at the bottom of the Edit window. This opens a plot pane. Next press **Make button** in **Plot Panel**. You will see the text in **Plot Scripting Window** as shown Fig. 6.2. The text is a script in gnuplot. Press the **Plot button**, then you will see a plot shown in Fig. 6.3. What has happend is that the table in the data window is sent to a file "plot.dat" in the directory(GOURMET/bin), and the gnuplot command shown in the plot scripting window is executed. You

Figure 6.1: Adding time series data to GraphSheet

can open the file "plot.dat", and see how the data in the window is transferred to "plot.dat". Note that the first column written as `[0], [1],[2]...` is exported as 0,1,2 ... and form the first column in "plot.dat".

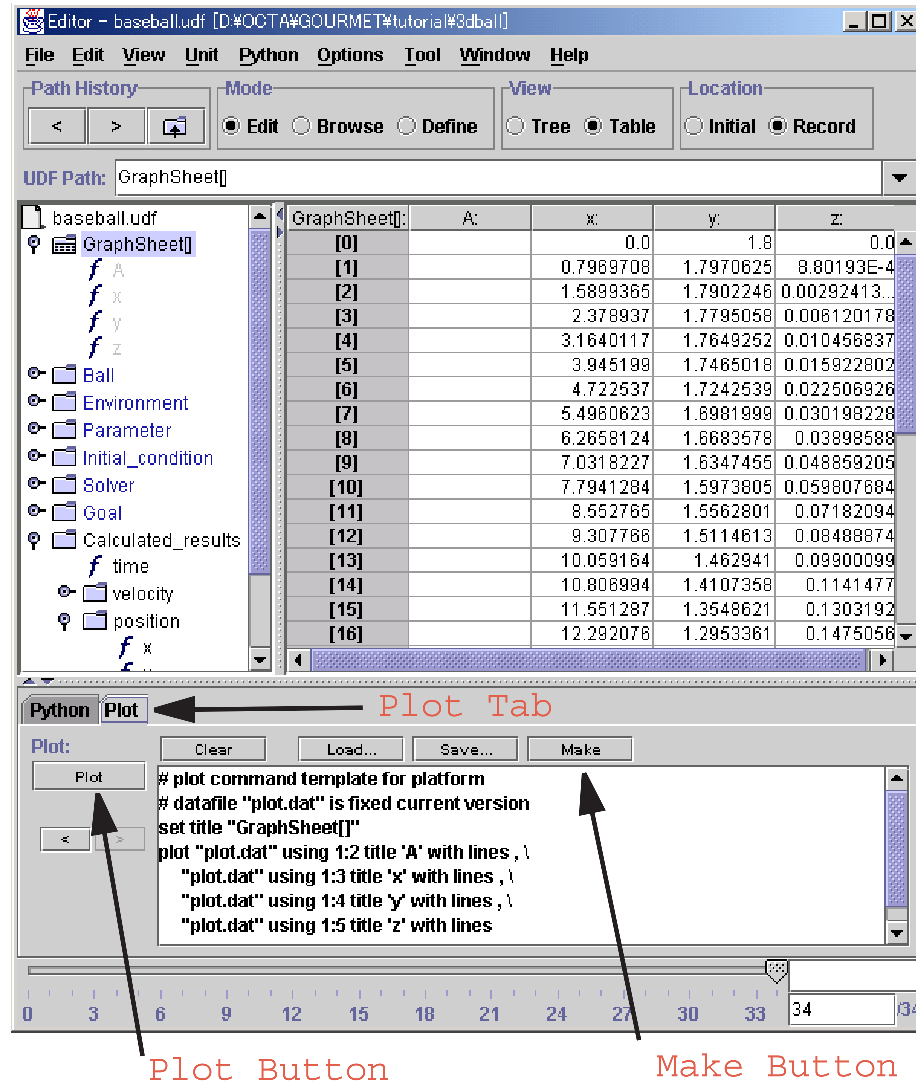
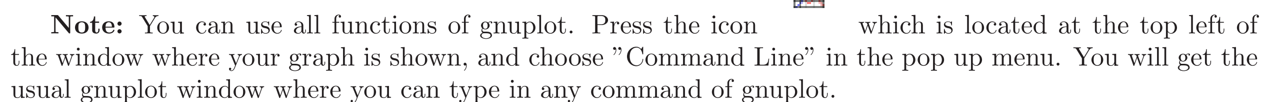


Figure 6.2: Making plot command from GraphSheet

If you want to see the plot of y against x, edit the plot script as follows.

```
plot "plot.dat" using 3:4 title 'y' with lines , \
     "plot.dat" using 3:5 title 'z' with lines
```

You will get gnuplot window as shown Fig. 6.4.



Note: You can use all functions of gnuplot. Press the icon which is located at the top left of the window where your graph is shown, and choose "Command Line" in the pop up menu. You will get the usual gnuplot window where you can type in any command of gnuplot.

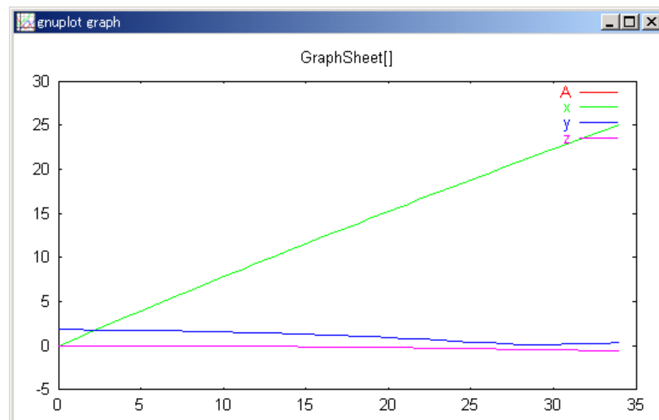


Figure 6.3: gnuplot window executed by plot tool simply

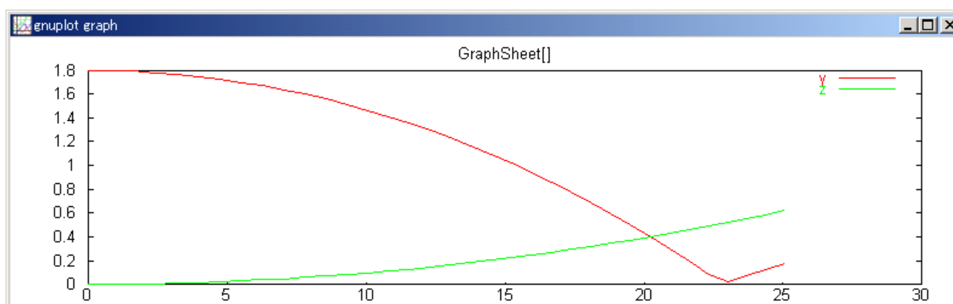


Figure 6.4: gnuplot window executed by plot tool

6.2 Plotting graphs by python

You can do the same plotting in the python scripting window. Return to the Python Panel, and load the script named "GOURMET/tutorial/3dball/script/plotlib.py", and run it. You will get the result immediately as shown Fig. 6.5.

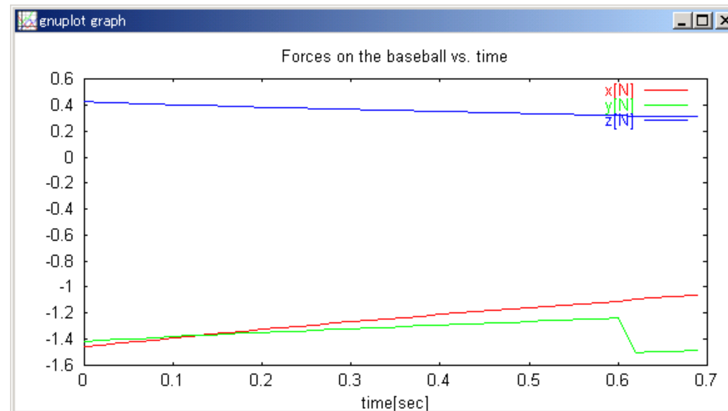


Figure 6.5: gnuplot window executed by python script using plot library

The content of the script is shown in Table 6.2. It utilizes a python modulue called "gnuplot". To plot a data, you first set the data to be plotted in an array (like `axis[], x[],...`), and then execute the command `gnuplot.plot`. The command sends the data to the file "plot.dat", and execute the plot command of gnuplot with the option in the argument.

Table 6.2: python script file for plotting Forces vs. time

```
# import gnuplot interface library
import gnuplot

axis = []
x = []
y = []
z = []

# append time serise data to list
for rec in range(totalRecord()):
    if $Calculated_results.time > $Solver.tmax:
        break
    jump(rec)
    axis.append($Calculated_results.time)
    x.append($Calculated_results.force.x)
    y.append($Calculated_results.force.y)
    z.append($Calculated_results.force.z)

# start gnuplot application with prepared data
gnuplot.plot(data=[axis,x,y,z],labels=['time[sec]','x[N]','y[N]','z[N]'],
             title='Forces on the baseball vs. time')
```

Chapter 7

Action

7.1 What is action

In this chapter, we will explain another way of executing the python script on GOURMET. It is called "Action". Action is a python script attatched to a data object in the UDF file, and can be called by clicking the data object with the right mouse button. You can use Action for various purpose, for example, to fill the input data automatically, to start a statisitical analysis of the data, to plot the array data, and to view the data in 3D graphics etc. Action is a tool for you to make your customization and expansion of GOURMET.

7.2 Kicking action

First let us see how to use Action.

Open the UDF file named "GOURMET/tutorial/3dball/baseball.udf" by Editor. Now select the data `Ball` and click it with the right mouse button. You will get the popup menu **setColor**. Press **setColor** and choose the color. You can see that the values of `Ball.color[0]`,`Ball.color[1]`, and `Ball.color[2]` have been changed.

Let us try other action. The following is a procedure of doing a curve ball simulation:

1. Select **BaseballInput** object by the right mouse button and choose **setSimpleCondition** action from the popup menu.
2. Choose **CurveBall** in Action Argument Dialog.
3. Select **Solver** object by the right mouse button and choose **calculate** action from the popup menu.
4. Select **baseball.udf** object(root object) by the right mouse button and choose **show** from popup menu.
5. Run animation by **Start** button in Python Panel.

You need only 5 actions to get the result shown Fig. 7.1. You can try other balls and see whether you miss the goal or not.

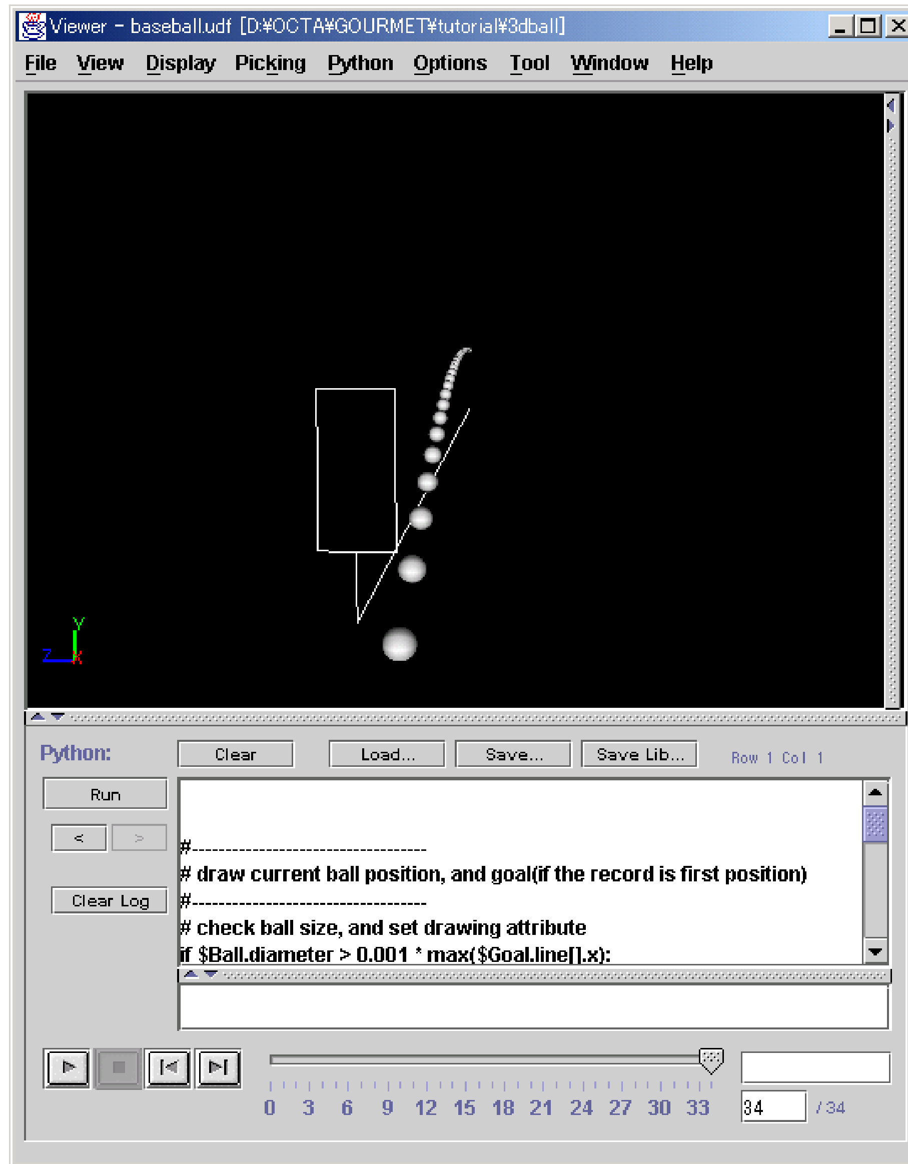


Figure 7.1: View of simulated Curve ball by Action

7.3 Defining action

Actions are defined in one or more action files and are linked by the UDF header data. Choose File/Header menu in Editor. You will get the current UDF header data shown in Fig. 7.2.

Note: The root path of action files directory is defined "GOURMET/action/" as default, and you can add another directory by using Environment variables "UDF_ACTION_PATH".

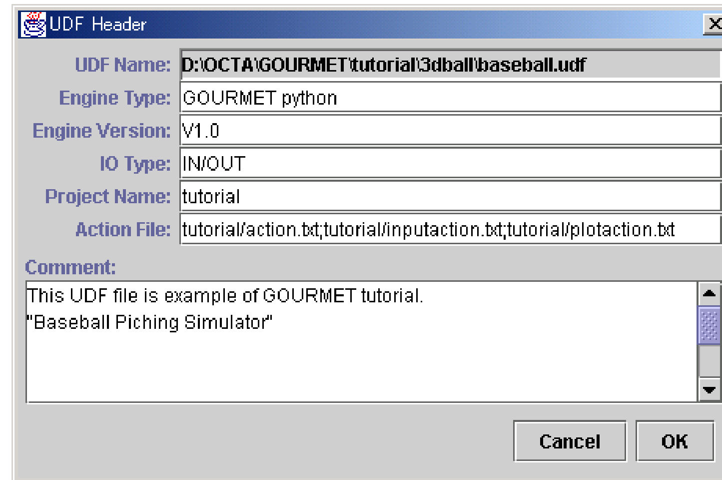


Figure 7.2: UDF header dialog

Here, "baseball.udf" is linked to 3 action files, listed in the following.

- **tutorial/action.txt**
  basic actions (which will be explained in 7.3.1 and 7.4)
- **tutorial/inputaction.txt**
  several actions for making input data(which will be explained in 7.4.2 and 7.4.3)
- **tutorial/plotaction.txt**
  many actions for plot the simulated results(explore by yourself)

7.3.1 Example of defining actions

The action file "GOURMET/action/tutorial/action.txt" defines following actions.

- : initialize()
- : clearDraw()
- GraphSheet[] : initialize()
- GraphSheet[] : addColumn(colname="newColumn",rows=0)
- Ball : setColor(BallColor="white | blue | green | red")
- Solver : calculate ()
- : show()

The following is the definition of the `setColor` action attatched to the `Ball` data.

```
action Ball: setColor(BallColor="white|blue|green|red") : \begin
#-----------------------------------
# set RGB color of Ball
#-----------------------------------
if BallColor == 'white':
    $Ball.color[] = [1,1,1]
elif BallColor == 'blue':
    $Ball.color[] = [0,0,1]
elif BallColor == 'green':
    $Ball.color[] = [0,1,0]
elif BallColor == 'red':
    $Ball.color[] = [1,0,0]
\end
```

The first line

```
action Ball: setColor(BallColor="white|blue|green|red") : \begin
```

defines the interface of the action. The arguments appears in the action pop up window. The syntax `BallColor="white|blue|green|red"` indicates that `BallColor` should be `"white"` or `"blue"` or `"green"` or `"red"`. The part starting from `\begin` and ending at `\end` is the python script which is executed when the OK button in the Action Argument Dialog is pressed. You can see the simple mechanism of action.

In general the Action is defined in the following way:

action target_object : action_name(argument1, argument2,...) : **\begin** python_script ... **\end**

When this action is invoked, the pop up window of the Action Argument Dialog shows a table consisting of name of argument and its value, and prompt users to enter the value of the argument. If the default value is a string and if it is separated by the character '|', it is regarded as a definition of the selection menu as it is in the case of `setBallColor` example. When the OK button in the Action Argument Dialog is pressed, the argument in the python script is replaced by its value, and the python sript is executed.

"Autorun" is a special action executed when the UDF file is loaded. It is defined as

```
autorun : action_name(argument1, argument2,...) : \begin
     ....python\_script ...
\end end
```

7.4 Actions of baseball simulator

In the following, we shall explain the action implemented in Baseball simulator.

7.4.1 Basic actions

Table 7.1: Basic actions

```
autorun : initialize() : \begin
#-----------------------------------
# autorun action for UDF file
# this action executed when the UDF file is opened or reloaded.
# You can import python modules or define python functions here.
#-----------------------------------
import gnuplot
from math import *
```

```
\end

action GraphSheet[]: initialize() : \begin
#-----------------------------------
# initialize GraphSheet Object
#-----------------------------------
# clear current GraphSheet data
$GraphSheet[] = []

# remove added columns from last
for i in range(getSheetColSize()-1,0,-1):
    deleteSheetCol(i)
\end

action GraphSheet[]: addColumn(colname="newColumn",rows=0) : \begin
#-----------------------------------
# add named column to GraphSheet Object, and add rows
#-----------------------------------
createSheetCol(getSheetColSize(),colname,rows)
\end

action : clearDraw() : \begin
#-----------------------------------
# clear current contents of 3D object window
#-----------------------------------
clearDraw()
jump(0)
\end

action Ball: setColor(BallColor="white|blue|green|red") : \begin
#-----------------------------------
# set RGB color of Ball
#-----------------------------------
if BallColor == 'white':
    $Ball.color[] = [1,1,1]
elif BallColor == 'blue':
    $Ball.color[] = [0,0,1]
elif BallColor == 'green':
    $Ball.color[] = [0,1,0]
elif BallColor == 'red':
    $Ball.color[] = [1,0,0]
\end

action Solver : calculate () : \begin
############################################################
```

Same as the contents of Table 5.1

```
############################################################
\end

action : show() : \begin
############################################################
```

Same as the contents of Table 5.2

```
############################################################
\end
```

7.4.2 Input action-simple

Our baseball engine needs the following data as initial condition which was explained in Table 4.3.

```
Initial_condition:{
  position:Vector3D [m] "initial position of the ball"
  velocity:Vector3D [m/s] "initial velocity of the ball"
  spin:Vector3D [m/s] "surface velocity caused by spin"
} "initial condition of calculation"
```

When you choose menu in setSimpleCondition Argument Dialog, this action selects predefined data and sets them to Initial_condition object as it is shown in Table 7.2.

Table 7.2: setSimpleCondition action

```
action BaseballInput : setSimpleCondition(throw="FastBall|CurveBall|ScrewBall|FalkBall")
 : \begin
$self.type = throw
if $self.type =='FastBall':
    $Initial_condition.position = [0,1.8,0]
    $Initial_condition.velocity = [45,-2,0]
    $Initial_condition.spin = [0,8.9,0]
elif $self.type =='CurveBall':
    $Initial_condition.position = [0,1.8,0]
    $Initial_condition.velocity = [40,0,0]
    $Initial_condition.spin = [0,-2.2,6.78]
elif $self.type =='ScrewBall':
    $Initial_condition.position = [0,1.8,0]
    $Initial_condition.velocity = [40,0,0]
    $Initial_condition.spin = [0,0,-6.78]
elif $self.type =='ForkBall':
    $Initial_condition.position = [0,1.8,0]
    $Initial_condition.velocity = [35,0,0]
    $Initial_condition.spin = [0,0,-1]
\end
```

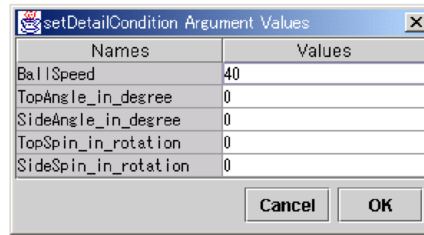
Note: The keyword 'self' is replaced to the target object name before the action is executed. In this example, '$self.type' will be replaced to '$BaseballInput.type'.

The reason why we define the input action at BaseballInput object will be explained in next subsection.

7.4.3 Input action-detailed

We have defined more general action which is used to set the Initial_condition in detail.

When you execute the action (Initial_condition : setDetailCondition), you will get the Argument Dialog shown as Fig. 7.3.


setDetailCondition Argument Values

Names	Values
BallSpeed	40
TopAngle_in_degree	0
SideAngle_in_degree	0
TopSpin_in_rotation	0
SideSpin_in_rotation	0

Cancel OK

Figure 7.3: setDetailCondition Argument Dialog

The content of setDetailCondition action is shown Table 7.3. Please be care that the value of Initial_condition.spin is the velocity at the ball surface. But, almost users need to input the value by rotation rate(rps etc.). So we have defined Input class as following and BaseballInput object using this class.

We have defined Input.alpha and Input.beta to calculate Initial_condition.velocity as following by the same reason.

```
class Input:{
  type:select{"Golf","Baseball","Teniss","TableTeniss"} "ball type selection"
  U0:float [m/s] "initial throw velocity"
  alpha:float [degree] "initial throw angle to y-axis"
  beta:float [degree] "initial throw angle to z-axis"
  rotation:Vector3D [rps] "initial rotation rate in each axis"
} "input tool for initial condition"
```

Note: Unit conversion of angle data is done using unit definition of [rps](See Table. 4.1) and built-in unit [rad]. What you need to convert unit is to define the unit used input, and to get the value by using get method.

Table 7.3: setDetailCondition action

```
action BaseballInput : setDetailCondition
   (BallSpeed=40,TopAngle_in_degree=0,SideAngle_in_degree=0,
    TopSpin_in_rotation=0,SideSpin_in_rotation=0) : \begin
from math import *
$self.U0 = BallSpeed
$self.alpha = TopAngle_in_degree
$self.beta = SideAngle_in_degree
$self.rotation.y = TopSpin_in_rotation
$self.rotation.z = SideSpin_in_rotation

# convert unit
alpha_in_rad = get("self.alpha","[rad]")
beta_in_rad = get("self.beta","[rad]")

$Initial_condition.velocity.x = $self.U0 *cos(alpha_in_rad)*cos(beta_in_rad)
$Initial_condition.velocity.y = $self.U0 *sin(alpha_in_rad)
$Initial_condition.velocity.z = $self.U0 *cos(alpha_in_rad)*sin(beta_in_rad)
print $Initial_condition.velocity

$Initial_condition.spin.x = 0.0
$Initial_condition.spin.y = $Ball.diameter * pi * $self.rotation.y
$Initial_condition.spin.z = $Ball.diameter * pi * $self.rotation.z
print $Initial_condition.spin
\end
```

Chapter 8

More and more

Having made the baseball simulator, you can try to make other simulators such as golf simulator and table tennis simulator by modifying the baseball simulator. Here are some tips.

8.1 Golf simulator

The UDF file of Golf simulator is "GOURMET/tutorial/3dball/golg.udf". GolfInput object has two actions shown in Table 8.1. The setDetailCondition action of golf simulator is slightly different from that of baseball simulator: in the golf simulator, the initial angle and the spin of the ball is determined by loft of club used.

The initial speed of the ball is determined by the speed of club head and the type of the club such as Persimmon and Iron used in the shot. Here, we have used the coeffcient of Meet Ratio to consider this effect.

You can use these two type of action for input for simple input and detail input. These dialogs of the action are shown Fig. 8.1 and Fig. 8.2.

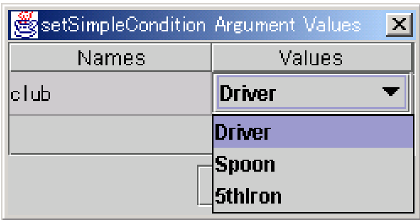


Figure 8.1: setSimpleCondition Argument Dialog of GolfInput

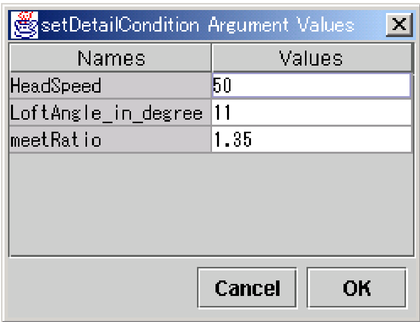


Figure 8.2: setDetailCondition Argument Dialog of GolfInput

Table 8.1: Actions of GolfInput

```
action GolfInput : setSimpleCondition(club="Driver|Spoon|5thIron") : \begin
$self.type = club
if $self.type =='Driver':
    $Initial_condition.position = [0,0,0]
    $Initial_condition.velocity = [65,12.5,0]
    $Initial_condition.spin = [0,9.5,0]
elif $self.type =='Spoon':
    $Initial_condition.position = [0,0,0]
    $Initial_condition.velocity = [61.5,18.5,0]
    $Initial_condition.spin = [0,14.5,0]
elif $self.type =='5thIron':
    $Initial_condition.position = [0,0,0]
    $Initial_condition.velocity = [48.5,30.0,0]
    $Initial_condition.spin = [0,26.5,0]
\end

action GolfInput : setDetailCondition
       (HeadSpeed=50,LoftAngle_in_degree=11,meetRatio=1.35) : \begin
from math import *
$self.type = "Golf detail input"
$self.alpha = LoftAngle_in_degree

# convert unit
alpha_in_rad = get("self.alpha","[rad]")
beta_in_rad  = get("self.beta","[rad]")

$self.U0 = meetRatio * HeadSpeed* cos(alpha)
$self.beta = 0.0

$self.rotation.y = $self.U0 *sin(alpha) / pi / $Ball.diameter
$self.rotation.z = 0.0

$Initial_condition.velocity.x = $self.U0 *cos(alpha_in_rad)*cos(beta_in_rad)
$Initial_condition.velocity.y = $self.U0 *sin(alpha_in_rad)
$Initial_condition.velocity.z = $self.U0 *cos(alpha_in_rad)*sin(beta_in_rad)
print $Initial_condition.velocity

$Initial_condition.spin.x = 0.0
$Initial_condition.spin.y = HeadSpeed *sin(alpha)
$Initial_condition.spin.z = 0.0
print $Initial_condition.spin
\end
```

Using Golf Simulator, you can determine which type of club you should use for a long shot as shown Fig. 8.3 , I hope. (^^;

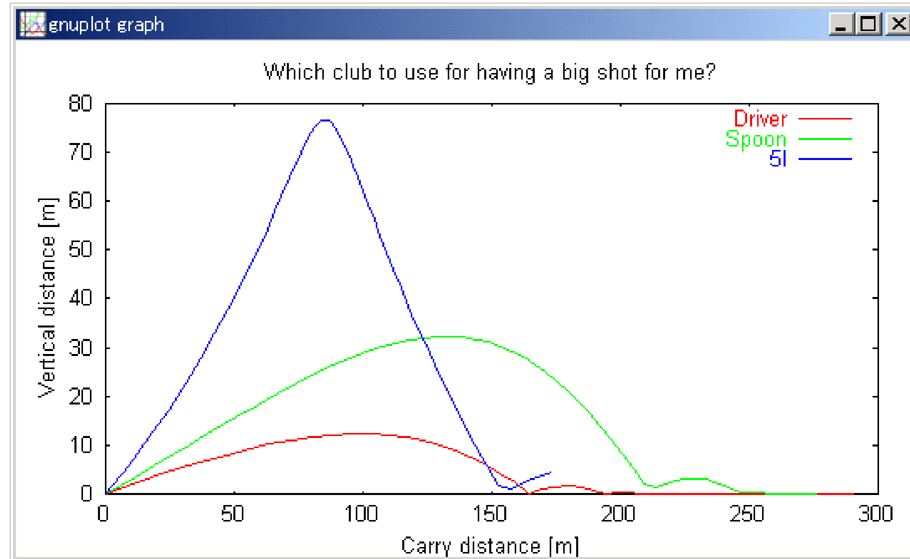


Figure 8.3: Which club to use?

8.2 Other simulator to be explored

8.2.1 Tabletennis simulator

Tabletennis is the sports which is strongly influenced by aerodrodynamic forces. The UDF file is "GOURMET/tutorial/3dball/tabletennis.udf". By the same way as the Baseball pitching simulator, you can build some input actions for servicing types such as Drive/Cut etc.

8.2.2 Tennis simulator

In tennis, the bouncing of the ball at the ground is quite important. In Chapter 3, we did not consider the effect of the ball spinning on the rebouncing of the ball. So, this simulator will be quite unsatisfactory for tennis players. Try your own modification, and let us know if you suceed to make a one satisfying them.

8.3 What now

I hope that you have enjoyed this tutorial. GOURMET has many functions to make your program easy to use. Please refer to the User's manual for more details.

- **GOURMET Operations Manual:** User's manual for operation of GOURMET

If you are going to develop new engine, the following documents will be useful.

- **Programming UDF:** Part 2 of this document
- **UDF Syntax Reference:** Refference of UDF syntax
- **GOURMET Python Script Manual:** User's manual for python scripting in GOURMET
- **libplatform:** Reference of Platform Interface Library for C++ programmer

Enjoy, and join our GOURMET community!

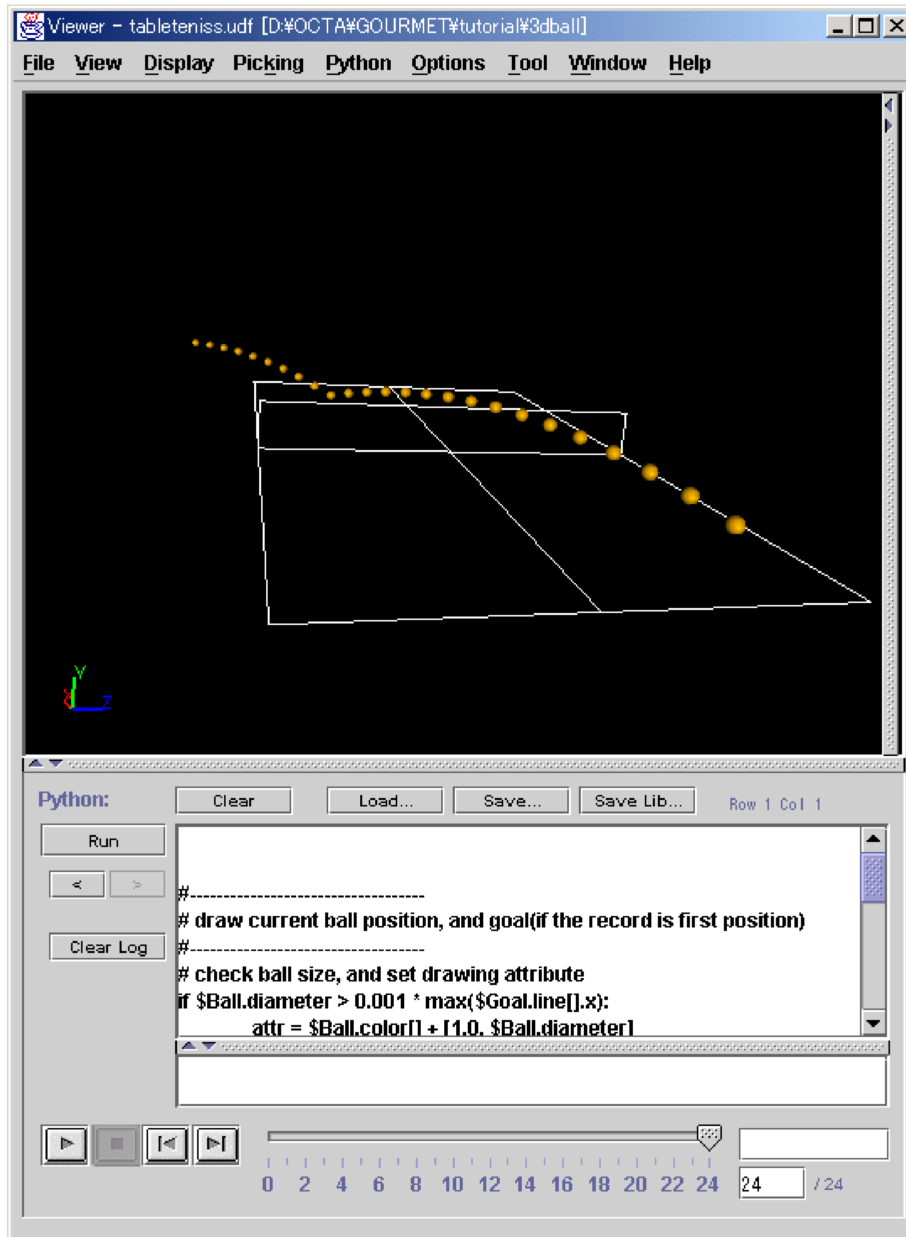


Figure 8.4: 3D result for Tabletennis servicing

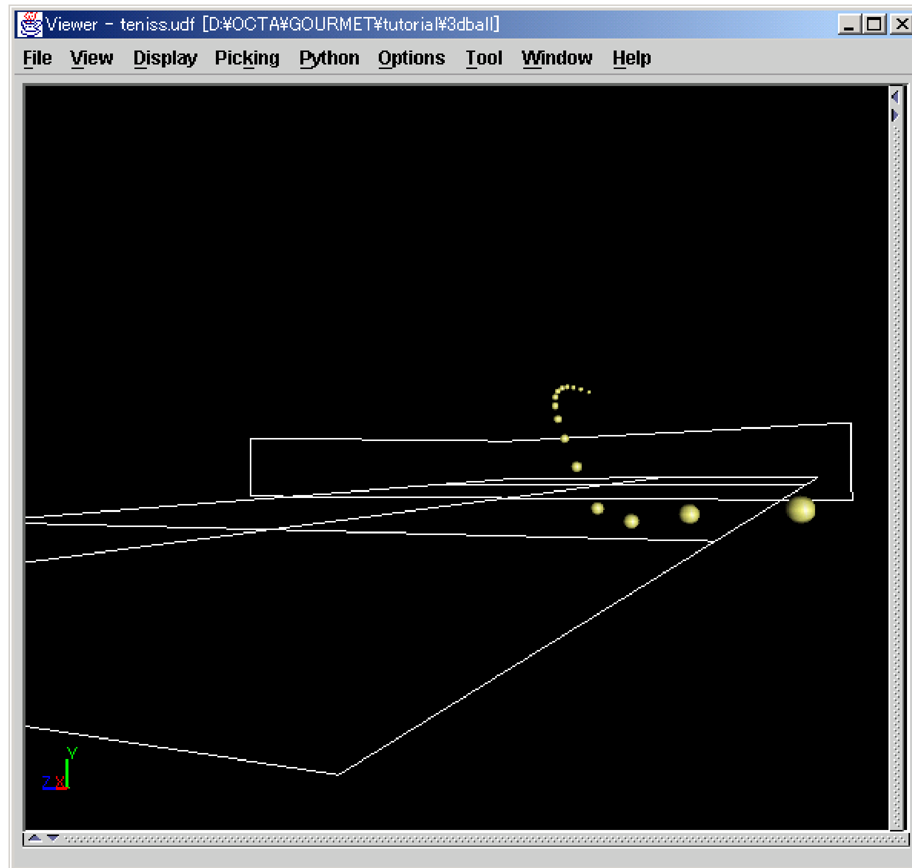


Figure 8.5: 3D result for Tennis servicing

Part II

Programing UDF

Chapter 9

Introduction

In this chapter, we will explain how to read and write the UDF file in C++ program. This is to use the UDF file as an input/output file of your simulation program. Simple programs, like data transformation or data analysis, can be done by GOURMET using the Python script. However, Python is not fast and uses a large memory. Any serious program where speed and memory is essential has to be written by C/C++ or Fortran.

Using the UDF file is advantageous since your program can be immediately connected to GOURMET, and can be used like other simulation programs running on GOURMET. Using the standard user interface of GOURMET, you can make your program easy to use for the potential users of your program. Suppose you have your own program and want to utilize the UDF file, you have two choices.

(1) You can leave the original program as it stands, and write a C++ program to transform the data from your file to the UDF file. Or you can utilize the function of the file transformation offered by GOURMET.

(2) You can rewrite the original program so that it can read and write the UDF file directly.

Reading and writing the UDF file is easy. All you need to learn is summarized in chapter 10. In fact you will see that reading and writing the UDF file is easier than the sequential file since any data in the UDF file can be accessed in a random order like a data in the data base file.

Chapter 11 and 12 is dedicated for those who want to start writing a C++ program using the UDF file as a input and output file. There we shall discuss how to map the data object(class and pointer) of the C++ to the data in the UDF file.

In chapter 13, we discuss briefly how to handle the UDF file by python script. This chapter does not describe the whole service you can get in GOURMET using the python script. The subject there is how to read and write the UDF file by the python script. The python script can be run in the command line of python or in the python window of GOURMET. It will be useful for the job, which needs to be done quickly.

Chapter 10

Basic operation

10.1 Simple data

10.1.1 Simple example of UDF file

UDF consists of two parts, the definition part, and the data part. In the definition part, you define the name and other attribute of your data(type, unit etc). In the data part, you give the actual data. The definition part is written between `\begin{def}` and `\end{def}`, while the data is written between `\begin{data}` and `\end{data}`. A simple example of UDF file is given below:

```
// "file1.udf"
\begin{def}  // definition part
   title   : string   //title of the project
   Nmax: int          // maximum size of the array
   average: double    // maximum value of the random number
   x[]: double        // list of random numbers
\end{def}

\begin{data}  // data part
   title:  "simple statistics"
   Nmax: 10
   x[]: [ 2.4  3.2  2.5]
\end{data}
```

In the definition part, the data name and its type must be specified. In this example, the definition part states that `title` is a data of string type, `Nmax` is a data of integer type, and `average` is a data of double type. Finally `x[]` is an array of data of double type.

(1) Basic data types are the same as in C++. They are integer type (int, short, long), floating-point type (float, double, single) and string type (string). There are some other data types(select, ID,KEY), which will be explained later (see chapter 12). User can define own data type combining these data type (see chapter 11).

(2) In the definition part, the text after "//" is regarded as a comment, and has no effect in the definition.

(3) The size of an array is determined by the number of the data given in the data part, and need not be specified in the definition part.

In the data part, the data is written after the data name and the character ':'. Numerical data (int, short, long, float, double and single) are written in the same way as in C++ such as (-2.4, 5.0E-12 etc). Unlike C++, however, the string data in the data part has to be sandwiched by " or '. If a string is not sandwiched by these characters, it is regarded as a comment in the data part and is ignored.

The data name may appear in any order independently of the order in the definition part. There is no need to give values to all data defined in the definition part. In the above example, the value of the data `average` is not given in the data part since it is an output of the program.

The array data must start with the character '[' and end with the character ']'. The data may be separated by space, comma, tab, new lines or any non-data object.

As a result of these rules, there is a considerable freedom in the writing of the data part. For example the following is a valid example of the data part corresponding to the above definition.

```
\begin{data}
  Name of the project
    title:  "simple statistics"
  Maximum number of data points
    Nmax: 10
  Generated random numbers
  x[]: [
       first      2.4
       second     3.2
       third      2.5
           ]
\end{data}
```

In this example, the keyword `title:`, `Nmax:`, and `x[]:` are the data names, and cannot be eliminated. On the other hand, the text strings such as "Name of the project" or "Generated random numbers" or "first" etc. are comments and can be omitted.

10.1.2 Reading a UDF file in C++ program

Reading and writing the UDF file is done by UDFManager in the C++ class. An example of reading the above UDF file is shown.

```
#include "udfmanager.h"
main(){
  UDFManager uf("file1.udf");
  string title;
  uf.get("title", title);
  int Nmax;
  uf.get("Nmax", Nmax);
  int n= uf.size("x[]");
  if ( n> Nmax) {cerr << "Error" << endl; exit;}
  double* x = new double(n);
  uf.getArray("x[]", x );
}
```

(1) To handle a UDF file, you first create an UDFManager. Here `UDFManager uf("file1.udf");` indicates that you create an UDFManager named uf to handle the UDF file "file1.udf".

(2) To read a data in the UDF file, send "get" message to the UDFManager with two arguments, the data name in the UDF file, and the C++ variable to which the data should be stored. The interface of the get method is:

```
bool UDFManager::get(const string &data_name, int &value) etc
```

The data name in the UDF file is specified by a string. In this example the sentence `uf.get("Nmax", Nmax);` reads the data named "Nmax" in the UDF file, and store it to `Nmax` in the C++ program. The same get method can be used to read the data of int, double, string etc. as they are differentiated by the type of the second argument in the get method.

(3) The size of an array can be obtained by the "size" method.

(4) "getArray" method is used to read an array data. To use the method, a memory space must be allocated first. This is done in

```
int n= uf.size("x[]");
double* x = new double(n);
```

Reading the whole array is done by the "getArray" method.

```
uf.getArray("x[]", x );
```

If you do the memory allocation using the vector in STL (standard template library), you can read the array data even more simply like:

```
vector<double> x;
uf.getArray("x[]", x);
```

The interface of the "getArray" method is

```
bool UDFManager::getArray(const string &data_name, double*  array) const;
bool UDFManager::getArray(const string &data_name, vector<double> &array) const;
```

Of course you can read each component of the array data as follows:

```
double x0,x1;
uf.getArray("x[0]", x0); uf.getArray("x[1]", x1)
```

UDFManager offers other method to read a data in the UDF file. The above program may be written in the following way.

```
string tile = uf.stringValeu("title");
int Nmax = uf.intValue("Nmax");
vector <double> x = uf.doubleArray("x[]");
```

In this case, the method to read string data and that of integer data cannot be the same: they are distinguished by the name of the method. The methods "intValue", "doubleValue", "stringValue" etc and also "intArray", "doubleArray" etc are available. The above style is simple and easy to use. However, you should be aware that the method requires a data copying and is not economical if `"x[]"` is a huge array.

10.1.3 Writing a UDF file in C++ program

Writing the output of your C++ program into a UDF file is as easy as reading. In the following, a program to calculate the average of the `x[]` is shown:

```
  UDFManager uf("file1.udf");
// data reading
  int n= uf.size("result[]");
  double* x = new double(n);
  uf.getArray("x[]", x );

// calculating average
  double sum=0;
  for (int i=0; i< n; i++) sum +=x[i]
  double average = sum/n

// data writing
  uf.put("average", average);
  uf.write("file1_out.udf");
```

When this program is executed, the content of the file1_out.udf will be:

```
\begin{data}
   title:  "test of random number"
   Nmax: 10
   average:2.7
   x[]: [ 2.4  3.2  2.5]
\end{data}
```

As it is shown here, data writing is done by the put method. The interface is the same as the get method for data reading: the first argument of the put method is the data name which is defined in the definition part of the UDF file, and he second argument is the C++ variable where the data is stored for output.

UDFManager handles the UDF file very much like a data base program. UDFManager can read any data by the get method, and can change the value of any data by the put method. The change of the data becomes effective when "write" method is executed. The sentence `uf.write("file1_out.udf")` writes the definition part and the data part to the file specified by the argument. If the argument is not specified, UDFManager will write the data to the file it is handling (in this case "file1.udf").

10.2 Structure type

10.2.1 Definition of structure type

In C, structure type is a data type, which consists of several data components. UDF has an equivalent to the structure type in C. The definition of a data of structure type is simple.

```
// "file2.udf"
\begin{def}
  vector:{x:double, y:double, z:double}
  simulation_condition:{
       temperature: double
       box:{Lx:double, Ly:double, Lz:double}
  }
\end{def}
```

Here `vector`, and `simulation_condition` are the data of structure type (hereafter abbreviated as structured data). A structured data is defined by the list of data components written between { and }. Like C++, each component of a structured data can be referred to using "." such as `vector.x` and `simulation_condition.box.Lx`.

10.2.2 Data part of structured data

In the data part, the data of a structure type is written between { and }. Like an array, each component of the structured data can be separated by space, comma, tab, new lines or any non-data object such as:

```
// "file2.udf"
\begin{data}
  vector:{ 1.0  3.0   -1.0}
  simulation_condition:{ 4.0 , { 1.0  1.0  3.0}  }
\end{data}
```

Notice that only the name of the top-level object needs to be given: the names of the sub-level objects need not be given as they are defined in the definition part. Of course it is not harmful to put the names of the sub-level object as comments. The following is an example of valid data part.

```
\begin{data}
  vector:{ x    1.0
           y    3.0
           z   -1.0
        }
  simulation_condition:{
            temperature  4.0
```

```
            box_size    Lx   Ly   Lz
                     { 1.0   1.0  3.0  }
                }
\end{data}
```

10.2.3 Reading structured data in C++

Reading the structured data is straightforward if you do not mind of calling the names of all data you need. For example, you can get each data item of the above file ("file2.udf") as follows:

```
UDFManager uf("file2.udf")
double vx, vy, xz, temp, Lx, Ly, Lz;
uf.get("vecor.x", vx);
uf.get("vecor.y", vy);
uf.get("vecor.z", vz);
uf.get("simulation_condition.temperature", temp);
uf.get("simulation_condition.box_size.Lx", Lx);
uf.get("simulation_condition.box_size.Ly", Ly);
uf.get("simulation_condition.box_size.Lz", Lz);
```

You can make the program shorter by introducing a string variable that represents the current UDF path.

```
UDFManager uf("file2.udf")
double vx, vy, xz, temp, Lx, Ly, Lz;
string current ="vector.";
uf.get(current+"x", vx);
uf.get(current+"y", vy);
uf.get(current+"z", vz);
current="simulation_condition.";
uf.get(current+"temperature", temp);
current= current+"box_size.";
uf.get(current+"Lx", Lx);
uf.get(current+"Ly", Ly);
uf.get(current+"Lz", Lz);
```

Though this method is easy to understand, it requires a rather expensive manipulation of strings to read each data item. To avoid this, Location class has been introduced. The usage of this class is demonstrated in the following example.

```
UDFManager uf("file2.udf")
double vx, vy, xz, temp, Lx, Ly, Lz;
Location loc;
loc.seek("vector");
uf.get(loc.sub("x"), vx);
uf.get(loc.sub("y"), vy);
uf.get(loc.sub("z"), vz);
loc.seek("simulation_condition");
uf.get(loc.sub("temperature"), temp);
loc.down("box_size");
uf.get(loc.sub("Lx"), Lx);
uf.get(loc.sub("Ly"), Ly);
uf.get(loc.sub("Lz"), Lz);
```

This program utilizes the following method of the UDFManager class.

```
bool UDFManager::get(const Location& loc, double value);
```

and the following methods of the Location class.

```
Location Location::seek(const string& location)
Location Location::sub(const  string& component);
```

The method `Location::sub(const string& component)` returns the location of the component data specified by the argument (see Fig.10.1). The method `+Location::seek(const&`string)+ changes the current location to the location specified by the string.

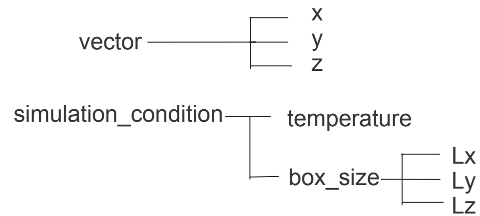


Figure 10.1: UDF data-hierarchy in the structured type variable

10.2.4 Writing structured data in C++

Writing the structured data to a UDF file can be done in a way similar to reading. An example is shown below.

```
UDFManager uf("file2.udf");
uf.put("vector.x", 1.0);
uf.put("vector.y", 0.0);
 ... all data is set
uf.write("file2_out.udf")
```

So far, the structured data has been handled for each data component. However, it is better to handle the structured data as a single entity. In chapter 11, we shall discuss how to read and write the structured data as a single entity, and how to map the structured data in a UDF file to an object in C++.

10.3 Unit

You can state the unit for each data item in the definition part. An example is given in the following.

```
\begin{def}
  pendulum_length : double [cm]
  atomic_radius: double [nm]
\end{def}
\begin{data}
  pendulum_length : 50.0
  atomic_radius: 0.2
\end{data}
```

T he unit can be put to any number of floating point type (float, double, single), and must be placed after the type. Basic units (such as those used in SI and cgs unit system) are implemented. User can define own unit as it will be discussed later.

Unit is introduced for the readability of the UDF file. Notice that the data part is not changed. The above UDF file means that `pendulum_length` is 50.0[cm] and `atomic_radius` is 0.2 [nm], but when the data is passed to C++ programs, these units are ignored. Thus `uf.doubleValue("pendulum_length")` gives 50.0 and `uf.doubleValue("atomic_raidus")` gives 0.2. Therefore the unit of the physical quantities is used by GOURMET only, and does not affect the simulation program.

GOURMET offers a service of showing the data in other unit, but the data is always recorded in the UDF file as a number written in the unit defined in the definition part. For example, when GOURMET,

`pendulum_length`, reads the above UDF file is shown as `50.0[cm]`. User can change the unit for `pendulum_length` from the default unit `[cm]` to `[m]`. If he does it, `pendulum_length` is shown as `0.5[m]`. Suppose the user change `pendulum_length` to `3[m]`, then the data part of the stored UDF file becomes `pendulum_length:300` , meaning that `pendulum_length` is 300[cm], and `uf.doubleValue("pendulum_length")` gives 300.0.

New units can be defined between `\begin{unit}` and `\end{unit}`. Consider the following example.

```
\begin{unit}
  [eV]=1.60E-19 [J]
  [L]=3[m]
  [T]=2[s]
  [M]=5[kg]
  [E]=[M * L^2 / T^2]
\end{unit}
\begin{def}
  pendulum_length:[L]
  kinetic_energy:[E]
\end{def}
\begin{data}
  pendulum_length:0.5
  kinetic_energy: 0.1
\end{data}
```

In this example, `pendulum_length` is `0.5[L]=1.5[m]`, and `kinetic_energy` is `0.1[L]=0.1*5*3*3/(2*2)[kg m^2/s^2] = 1.125[J]`. As it is seen in this example, new unit can be defined using the predefined unit. In this definition, user can see the value of `pendulum_length` in the default unit ([L]) or in the other standard unit such as [m] or [cm]. Also he can see the value of `kinetic_energy` in the default unit ([E]) or in the unit of [J] or [erg].

The following syntax allows users to change the factors needed in the unit transformation:

```
\begin{unit}
  [L]={$reference.length}[m]
  [T]={$reference.time}[s]
  [M]={$reference.mass}[kg]
  [E]=[M * L^2 / T^2]
\end{unit}
\begin{def}
  reference:{ length: double [m], time:double [s], mass:double [kg] }
  energy:double [E]
\end{def}
```

Here the units `[L], [T], [M]` are defined in reference to the quantity given in `reference`. In default, the `energy` is shown in a unit of `[E]`. However, if users give a value to `reference`, he can see the `energy` in `[J]` or `[erg]`.

This is a typical style of stating units in scientific simulations. Scientific simulations are usually done in a non-dimensional form. Programmers define their reference quantities, and make all physical quantities dimensionless with respect to these reference quantities. The above definition allows programmers to follow their traditions of using non-dimensional form, while it allows end users to see the physical quantities in SI or cgs units.

The unit is introduced to enhance the readability and transferability of the data file. Many people are accustomed to write the simulation code first by making all quantities appearing in the problem dimensionless using suitable reference quantities. If the reference quantities are not stated explicitly, it is not possible to transfer the information of such file for other usage. Stating unit for all physical quantities is strongly advised: it improves the readability of the file, and also increases the possibility of the file to be used by other applications.

10.4 Record

UDFManager reads all text data written by UDF, and starts the service to a user. However, when the output of a simulation program is very big, it is impossible to read all data and to develop on memory.

For example, although the position of all atoms, speed, strength, etc. are outputted to adequate timing by molecular dynamics, if the number of times to output becomes what 10,000 times, it is impossible to read all data in memory. Usually, it is sufficient if all the data for every time step to output are developed on memory. Also in the program of a finite element method, if data, such as a speed place of the system of each moment and displacing space, are developed on memory, it will be sufficient. The record unit of the data that UDFManager can develop and treat on memory is called record. Probably, in the case of large-scale simulation, a record may be called record of the condition of a certain instantaneous system.

UDFManager offers the following methods to handle a record.

1. string UDFManager::newRecord(string recordName="")
   The current record is closed, a new record is created and a record label is returned.

2. unsigned int UDFManager:: totalRecord ()
   The total record count is returned.

3. int UDFManager:: currentRecord ()
   It returns what the number of current records is.

4. const string& UDFManager:: getRecLabel ()
   The label attached to the current record is returned.

5. void UDFManager:: setRecLabel (const string& recordLabel)
   The label of the current record is changed.

6. bool UDFManager::jump(const string& recordLabel)
   It moves to the first record that has the record label mached with specified recordLabel.

7. bool UDFManager:: jump (int timeStep)
   It moves to the record specified by timeStep.

8. bool UDFManager::nextRecord();
   It moves to the next of the current record.

There are two kinds of records. They are the portion (this is across the board to all time) which described conditions, a physical constants, etc. of a simulation, and the portion (this changes each time) that described time increase of the system. The former is called global record and the latter is called time series record.

```
\begin{unit}
    \textbf{Unit definition}
\end{unit}
\begin{def}
        UDF class definition
        UDF instance definition
\end{def}
\begin{global_def}
        The definition of global record
\end{global_def}

The description of an initial state using global record

\begin{record}["step1"]
The description of the data at 1st. step
\end{record}
\begin{record}["step2"]
The description of the data at second step
\end{record}
```

(1) Global record is defined between `\begin{global_def}` and `\end{global_def}`.

(2) Time series records are defined between `\begin{record}` and `\end{record}`.

(3) In advance of a time series record, global record is restricted once and written.

(4) You may write one or more time series record. The variable currently written to each record may change each time.

(5) UDFManager manages both global record and the time series record of one batch on memory.

(6) The portion surrounded between [] after `\begin{record}` is the label attached to the record. This label can be used jumping to a certain record.

As an example, the complete UDF file for the baseball simulator is shown in Table 10.1.

Table 10.1: Baseball simulator definition and data

```
\begin{def}
\begin{unit}
PI=3.141592
[rps]=2.0*PI[rad/s]
[rpm]=1.0/60.0*[rps]
[yard]=0.9144*[m]
\end{unit}
\begin{def}
class Vector3D:{
  x:float [unit]
  y:float [unit]
  z:float [unit]
} [unit]
class Input:{
  type:select{"Golf",  "Baseball",  "Tennis",  "TableTennis"} "ball type selection"
  U0:float [m/s] "initial throw velocity"
  alpha:float [degree] "initial throw angle to y-axis"
  beta:float [degree] "initial throw angle to z-axis"
  rotation:Vector3D [rps] "initial rotation rate in each axis"
} "input tool for initial condition"

\end{def}
\begin{global_def}
Ball:{
  mass:float [kg] "mass of the ball"
  diameter:float [m] "diameter of the ball"
  color[]:float "color attribute of the ball"
} "ball attributes"
Environment:{
  gravity:float [m/s^2] "gravity of environment"
  rho:float [kg/m^3] "density of atmosphere"
  U:Vector3D [m/s] "velocity of background flow"
  reflection_ratio: float "reflection ratio of ground"
} "environment parameter"
Parameter:{
  CD:float "drag force coefficient"
  CL:float "lift force coefficient"
} "aerodynamic coefficients"
Initial_condition:{
```

```
  position:Vector3D [m] "initial position of the ball"
  velocity:Vector3D [m/s] "initial velocity of the ball"
  spin:Vector3D [m/s] "surface velocity caused by spin"
} "initial condition of calculation"
Solver:{
  dt:float [s] "diviation time"
  tmax:float [s] "time period for calculation"
  output_interval:float [s] "output interval for saving record"
} "solver parameters"
Goal:{
  line[]:Vector3D [m]
} "goal shape for show 3D result"
\end{global_def}
\begin{def}
Calculated_results:{
  time:float [s] "time from start"
  velocity:Vector3D [m/s] "velocity at the time"
  position:Vector3D [m] "position at the time"
  force:Vector3D [N] "force at the time"
} "calculated record at each time"
\end{def}

\begin{global_def}
BaseballInput:Input "input tool for initial condition"
\end{global_def}

\begin{data}
Environment:{9.8, 1.3,{0.0, 0.0, 0.0} 0.5}
Goal:{
    [
         {0.0, 0.0, 0.0}
         {18.44, 0.0, 0.0}
         {18.44, 0.4, 0.0}
         {18.44, 0.4, 0.217}
         {18.44, 1.3, 0.217}
         {18.44, 1.3,-0.217}
         {18.44, 0.4,-0.217}
         {18.44, 0.4, 0.0}
    ]
}
Ball:{
    0.1445, 7.15e-2,
    [1.0, 1.0, 1.0]
}
Solver:{1.0e-2, 0.7, 2.0e-2}
Parameter:{0.35, 0.25}
Initial_condition:{{0.0, 1.8, 0.0}{45.0, 0.0, 0.0}{0.0, 8.9, 0.0}}
BaseballInput:{"CurveBall",40.0, 0.0, 0.0, {0.0, 0.0, 30.0}}
\end{data}

\begin{record}{"timestep 0"}
\begin{data}
Calculated_results:{0.0,{40.0, 0.0, 0.0}{0.0, 1.8, 0.0}{-1.4615154,-1.6457667,-0.4246746}}
\end{data}
\end{record}

\begin{record}{"timestep 1"}
```

```
\begin{data}
Calculated_results:{2.0e-2,{39.79,-0.2274,-5.863e-2}{0.7969,1.796,-8.801e-4}
{-1.448, -1.6363,-0.4204}}
\end{data}
\end{record}
```

Chapter 11

Class

11.1 Introduction

Modern languages, C,C++, Pascal and Fortran90, support the structured data, which regard a group of data as a single object. As we discussed in the previous section, UDF supports the structured data as well. In the previous chapter, however, reading and writing the UDF file was done for each data component of the structured data. It is preferable to treat the structured data as a single entity, and make a direct connection between the structured data in the UDF file and the data object in the programming language. In this section, we shall discuss how to do such mapping taking C++ as an example.

11.2 UDF class

UDF class is a type of a structured data, which can be used in the definition part. The usage is illustrated in the following example.

```
// file3.udf
\begin{def}
  class Vector3d:{ x: double [unit], y:double[unit], z:double[unit]} [unit]
  point: Vector3d [m]
  class atom: { pos: Vector3d [nm],  vel: Vector3d [nm/s]}
  atom[]:Atom
\end{def}
\begin{data}
  point:{ 1.0  3.0   -1.0}
  atom[]:[    position          velocity
       {   { 1.0  1.0  3.0}    {-1.0, 2.0,-1.0}   }
       {   { 2.0  1.0  1.0}    {-2.0, 2.0,-2.0}   }
       {   { 3.0  1.0  2.0}    {-3.0, 2.0,-4.0}   }
        ]
\end{data}
```

Here the first line:

```
  class Vector3d:{ x: double [unit], y:double[unit], z:double[unit]} [unit]
```

is the definition of a new UDF class Vector3d. It states that Vector3d has three components x, y, z each being double type, and having the same unit specified [unit]. The syntax of the definition of the unit will become clear in the next line:

```
  point: Vector3d [m]
```

This definition is equivalent to the definition:

```
  point: {x:double [m], y:double [m], z:double[m] }
```

But the definition using the UDF class is simpler and easier to understand. The third and the fourth lines give further illustration of the usage of the UDF class. One can immediately see that the structure of the data file is as it is shown in Fig. 11.2.

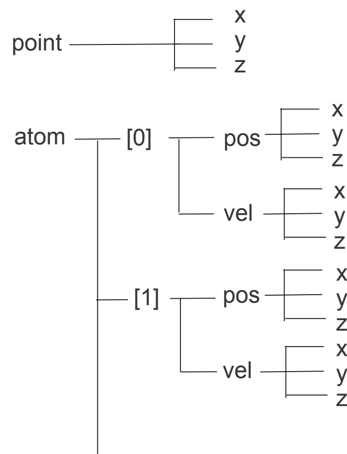


Figure 11.1: The data structure for the file "file3.udf"

11.2.1 Reading the structured data into C++ class object

In the following, we shall discuss how to map such UDF class to the class in C++. First, a C++ class, Vector3d, is defined as follows.

```
class Vector3d {
  public:
     double x;
     double y;
     double z;
     get( const UDFManager& uf, const Location& loc );
     put( const UDFManager& uf, const Location& loc);
}
```

The two methods "get" and "put" are added to read and write a structured data of `Vector3d` in the UDF file. The actual implementation of these methods are given below:

```
void
Vector3d::get( const UDFManager& uf, const Location& loc) {
uf.get(loc.sub("x"), x)
uf.get(loc.sub("y"), y)
uf.get(loc.sub("z"), z)
}
void
Vector3d::put( const UDFManager& uf, const string& loc) {
uf.put(loc.sub("x"), x)
uf.put(loc.sub("y"), y)
uf.put(loc.sub("z"), z)
}
```

Similarly, one can define the class Atom in the following way:

```
class Atom {
  public:
     get( const UDFManager& uf, const Location& loc);
```

```
     put( const UDFManager& uf, const Location& loc);
  private:
     pos: Vector3d;
     vel: Vector3d;
}
Atom::get( const UDFManager& uf, const Location& loc){
    uf.get(loc.sub("pos"), pos);
    uf.get(loc.sub("vel"), vel)
}
Atom::put( const UDFManager& uf, const Location& loc){
    uf.put(loc.sub("pos"), pos);
    uf.put(loc.sub("vel"), vel)
}
```

If the classes Vector3d and Atom are defined in such a way, one can write the following program to read and write the UDF file ”file3.udf”.

```
void
main(){
  UDFManager uf("file3.udf");
// reading
  Vector3d point(uf, Location("point") );

  int n= uf.size("atom[]");
  Vector3d* atom = new Atom(n);

  Location loc("atom[0]");
  for (int i=0; i< n; i++) {
       atom[i].get(uf, loc);
       loc.next();
  }
//writing
  point.put(uf, Location("point") );
  loc.seek("atom[0]");
  for (i=0; i< n; i++) {
       atom[i].put(uf, loc);
       loc.next();
  }
  uf.write("file3_out.udf);
```

Chapter 12

Pointer expression

12.1 Introduction

We shall now discuss how to map the pointer in C/C++ to a UDF file. As an example, let us consider the project of drawing many spheres in space. We assume that each sphere is characterized by the position of the center of mass, and the sphere type. The sphere type is a data set denoting the size and the color of the sphere, and has a name such as "big_blue", "small_green" etc. Then the class definition of our program will be:

```
struct SphereType {
   string name;
   double radius;
   int    color_code;
}
struct Sphere {
   Vector3d pos;
   SphereType* pTyepe
}
SphereType sphereType[5];
Sphere     sphere[100];
```

Here `sphere[i]` has a pointer `pType` that points the sphere type.

To write such data into a text file, we would like to replace the pointer by some readable quantity such as integer or string. One of the available methods is to use the index: if `sphere[i].pType` is equal to `&sphereType[p]`, we can record the index `p` to identify the sphere type. Then the definition part of our UDF file will be:

```
\begin{def}  // definition 1
  class SphereType:{ name:string,  radius:double, color_code:int}
  class Sphere: { pos: Vector3d,   type_code:int}
  sphereType[]: SphereType
  sphere[]:Sphere
\end{def}
```

An example of the data part will be

```
\begin{data}
  sphereType[]:[{"big_blue",    3.0,  2}
              {"small_green", 1.0,  1}
              {"medium_red",  2.0   0}]
  sphere[]:[       position     type_code
            {  { -1.0  0.0  0.0}    1  }
            {  {  0.0  0.0  0.0}    0  }
            {  {  1.0  0.0  0.0}    2  }
          ]
\end{data}
```

The data shows that `sphere[0]` is `small_green`, `sphere[1]` is `big_blue` and `sphere[2]` is `medium_red`. This mapping is simple and straightforward, but the fact that `type_code` of the class `Sphere` is a replacement of a pointer in C++ (i.e., `type_code` is used to identify other data object) is not apparent in this definition. You can add comments to explain the logical structure for other users, but you cannot make such comments understandable for computers.

In order to describe such logical structure in the file, UDF has two data types to represent a pointer, KEY and ID. KEY replaces the pointer to a string, and ID replaces the pointer to an integer. They can appear only in the class definition, and are used as a unique name for the data object of that class. They are explained in the following sections.

12.2 KEY

KEY is a string used as a name of the data object in the class. Using KEY, above UDF file can be rewritten as follows.

```
//file 2
\begin{def}
  class Vector3d:{ x:float, y:float, z:float}
  class SphereType:{ name:KEY,  radius:double, color_code:int}
  class Sphere: { pos: Vector3d,  type:<SphereType,KEY> }
  sphereType[]: SphereType
  sphere[]:Sphere
\end{def}
\begin{data}
  sphereType[]:[{"big_blue",    3.0,  2}
                {"small_green", 1.0,  1}
                {"medium_red",  2.0   0}]
  sphere[]:[      position            type
            {  { -1.0  0.0  0.0}    "small_green"  }
            {  {  0.0  0.0  0.0}    "big_blue"     }
            {  {  1.0  0.0  0.0}    "medium_red"   }
          ]
\end{data}
```

Here `name` of `SphereType` is KEY. The value of the KEY must be unique within the class as it is used to identify the data object in the class.

On the other hand, `type` of `Sphere` is defined as `<SphereType,KEY>`. It means that `type` is the key to identify a data object in the UDF class `SphereType`.

12.3 ID

ID is the same as KEY except that ID is an integer. No two objects in a class should have the same ID. Using ID, you can rewrite the above data structure as:

```
//file 3
\begin{def}  //definition 3
  class Vector3d:{ x:float, y:float, z:float}
  class SphereType:{ name:ID,  radius:double, color_code:int}
  class Sphere: { pos: Vector3d,  type_code:<SphereType,ID> }
  sphereType[]: SphereType
  sphere[]:Sphere
\end{def}
\begin{data}
  sphereType[]:[{1, 3.0,  2}
                {2, 1.0,  1}
                {3, 2.0   0}]
  sphere[]: [ position            type_code
```

```
            {  { -1.0  0.0  0.0}     2 }
            {  {  0.0  0.0  0.0}     1 }
            {  {  1.0  0.0  0.0}     3 }
          ]
\end{data}
```

To see how it works, try the python script:

```
 print get(getLocation("SphereType",2) )
```

12.4 Reading in C++

The following is a C++ program, which reads "file2.udf".

```
struct SphereType;
map<string, SphereType*> sphereTypeTable;
struct SphereType {
     string name;
     double radius;
     int    color_code;
     void   get(const UDFManager& uf, const Location& loc);
  };
struct Sphere {
    double x,y,z;
    SphereType* type;
    void get(const UDFManager& uf, const Location& loc);
};
void SphereType::get(const UDFManager& uf, const Location& loc) {
     uf.get(loc.sub("name"), name);
     uf.get(loc.sub("radius"), radius);
     uf.get(loc.sub("color_code"), color_code);
     sphereTypeTable[name]=this;
  };
void Sphere::get(const UDFManager& uf, const Location& loc){
   uf.get(loc.sub("pos.x"), x);
   uf.get(loc.sub("pos.y"), y);
   uf.get(loc.sub("pos.z"), z);
   string type_code;
   uf.get(loc.sub("type_code"), type_code);
   type = sphereTypeTable[type_code];
};
main(){
     UDFManager uf("keyTest.udf");
     int nst=uf.size("sphereType[]");
     SphereType* sphereType= new SphereType[nst] ;
     Location loc("sphereType[0]");
     for (int i=0; i<nst; i++) {
          sphereType[i].get(uf,loc);
          loc.next();
     }
     int ns=uf.size("sphereType[]");
     Sphere* sphere= new Sphere[ns] ;
     loc.seek("sphere[0]");
     for (i=0; i<ns; i++) {
         sphere[i].get(uf, loc);
         loc.next();
     }
     for (i=0; i< ns; i++ ){
```

```
            cout << i << "  " << sphere[i].type->name << endl;
      }
}
```

In this example, to relate the key with pointer, the data type map in the standard template library is used. The program works only when all `SphereType` data is read prior to the data of `Sphere`.

If ID is used instead of KEY, the `sphereTypeTable` should be defined as `map<int, SphereType*>`.

12.5 Other functions related to ID and KEY

1. `vector<int> UDFManager::getIdList(const string &class_name)`

   This method returns the list of all IDs of the UDF object in `class_name`.

2. `const Location& UDFManager::getLocation(const string &class_name, int id)`

   This method returns the location of the UDF object which has the `id` in the class `class_name`.

3. `vector<string> UDFManager::getKeyList(const string &class_name)`

   This method returns the list of all the keys of the UDF data object in `class_name`.

4. `const Location& UDFManager::getLocation(const string &class_name, string key)`

   This method returns the location of the UDF object which has the `key` in the class of `class_name`.

Chapter 13

Handling UDF file by python

13.1 The $ prefix

The python script can handle the UDF file. If you open a UDF file on GOURMET, you can handle any data in the file using the $ option. For example, open the following UDF file ”python_test.udf” in GOURMET:

```
// "python_test.udf"
\begin{def}
   title: string
   x[]: double           // list of random numbers
   output:{
      average: double
      max:double
   }
\end{def}

\begin{data}
   title: "statistics"
   x[]:[ 2.4  3.2  2.5]
\end{data}
```

Now type the following python script in the python window, and press the Run button.

```
print $title, $x[0], $x[]
$output=[2.7, 3.2]
print "average=",$output.average
```

The output of the script will be

```
statistics  2.4 [2.4,3,2,2.5]
average= 2.7
```

As you can see, any data in the UDF file opened by the file menu of GOURMET can be accessed by calling the data name with the prefix $. Notice that the prefix $ is not a valid symbol for python parser. The above python script utilizes an extension of the python script, which is available only on GOURMET.

If you want to do the same thing in the command line of Python, you have to do the following:

```
from UDFManager import *
uf = UDFManager('python_test.udf')
print uf.get("title"),  uf.get("x[0]"),  uf.get("x[]")
uf.put([2,7,3,2], "output")
print "average", uf.get("output.average")
```

Here ”get” is the method of returning the value specified by the argument, and ”put” is the method of putting the value of the first argument, to the data specified by the second argument.

Of course you can run the above program in Python Scripting Window of GOURMET. You can use this method to manipulate data that exists in various files. For example suppose that you want to combine the data in "fileA.udf" with the data in "fileB.udf", to make the data in "fileC.udf", where the definition part are given by:

```
// fileA.udf
\begin{def}
 A[]:{ x: int, y:int}
\end{def}
   ....  // data of A is omitted

// fileB.udf
\begin{def}
 B[]:{ u :int,v :int }
\end{def}
   ...   // data of B is omitted

//fileC.udf
\begin{def}
 C[]:{ x:int, y:int, u:int, v:int }
\end{def}
```

You can do it by the following python script:

```
ua=UDFManager("fileA.udf")
ub=UDFManager("fileB.udf")
uc=UDFManager("fileC.udf")
N=ua.size("A[]")
if N > ub.size("B[]"): N = ub.size("B[]")
for i in range(N):
    uc.put( ua.get("A[]",[i])+ub.get("B[]",[i]), "C[]",[i])
```

Here get("A[]",[i]) is equivalent to get("A[i]").

Other methods of the UDFManager are summarized in chapter 15.

Chapter 14

Documents for GOURMET

Refer to the following documents for more details.

- **GOURMET Operations Manual:** User's manual for operation of GOURMET
- **UDF Syntax Reference:** Reference of the UDF syntax
- **Python Script Manual:** User's manual for python scripting in GOURMET
- **libplatform:** Reference of Platform Interface Library for C++ programmer

References

1) Python Language Website. http://www.python.org/

2) Josh Cogliati. Non-Programmer's Tutorial For Python. http://www.honors.montana.edu/ jjc/easytut/easytut/

3) Konrad HINSEN. Python for Science. http://starship.python.net/crew/hinsen/

Chapter 15

Minimum reference of Python in GOURMET

This section is a very brief description of how to use Python and how to write Python scripts. If you want to know about Python in more detail, please read the books in the bibliography of the last of this chapter.

15.1 Variable

Data objects handled by python are numbers (integer, long integer, floating point and complex), text strings, an array object and the object of other classes. They are defined as follows.

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

Notice that type declaration for variables is not needed. String is enclosed by single quotes (’), double quotes (”), triple single quotes (”’) or triple double quotes (”””). Operators + for string data means a connection of the strings. For numerical values, the usual arithmetic operators (+, -, *, /), power operator (**) and modular operator (%) can be used.

15.2 Tuple, list, dictionary

To represent the collection of data, python offers a data type of tuple, list and dictionary. They are written in the following way:

```
tuple_example = ( 'awk' , 'perl' , 'ruby' , 'python' )
list_example = [ 1, 2.3, [ 'awk',' perl' , 'ruby' ] ]
dict_example = { 'awk':1, 'perl':[2,3], 'ruby':('diamond','carbon'), 'python':1.5 }
print tuple_example[2], list_example, list_example[2], dict_example['ruby']
```

The tuples , list, and dictionary start with ”(”, ”

”, ”{”*andendwith*”)”, ”}”, *and*”

” respectively. Each element of the tuple and the list are referred in the C style, such as `tuple_example[0]`, `list_example[1]` etc. The element of tuple and list need not be of the same type as you can see in the above example of `list_example`: `list_example[0]` is an integer 1, `list_example[1]` is a floating point data 2.3, and `list_example[2]` is a list `[ 'awk',' perl' , 'ruby' ]` The element of the dictionary type is referred to by the key. For example `dict_example['awk']` is equal to 1, and `dict_example['perl']` is equal to `[2,3]` . You can see this in the output of the last statement.

Tuple is a constant object:you cannot change the value of the elements and the number of elements in tuple. On the other hand, you can change the contents of the list. For example,

```
array    =  [1,2,3]
array[2] = 0
print array
array = array + [4]
print array
```

15.3 Control statements

Python has usual control syntax; "if/else" statements for conditional branching, and "for" and "while" statements for loops. An example of the conditional branching is the following:

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

An example of loop syntax is the following;

```
n=0
while n<5:
     print n
     n=n+1
```

Notice that the "if/else" block and the while block are denoted by the indent. Python is very strict for indentation: the indent must be made by the same number of space or tab character.

The "for" statement of python is slightly different from the "for" statement in C. The following python script demonstrates this.

```
for i in [0, 1,"apple", "orange"]:
     print i
```

You can see that the output is the printout of the list element. The "for" statement in Python execute the "for" block replacing the control variable by all elements in the list. The above example of the "while" statement is rewritten by using "for" as:

```
for i in [0,1,3,4]:
     print i
```

This program can be written as:

```
for i in range(5):
     print i
```

The function `range(N)` is equivalent to `[0,1,...N-1]`. More generally, `range(N,M)` is equivalent to `[N,N+1,...M-1]`, and `range(N, M, k)` is equivalent to the list of numbers that increase from N to (M-1) with the increment k.

15.4 Function and class

Functions can be defined by the def statement.

```
def fact(n):
    if (n<=1): return 1
    return fact(n-1)*n
for i in range(5):
    print i, fact(i)
```

Python has an implementation of "class" similar to that in C++. The following is a simple example of the definition and the usage of the class.

```
class Vector3d:
    def __init__(self, x, y, z):
        self.x=x; self.y=y; self.z=z
    def norm(self):
        return self.x**2+self.y**2+self.z**2
    def multiply (self, c):
        return Vector3d(c*self.x, c*self.y, c*self.z)
    def list(self):
        return [self.x, self.y, self.z ]
v = Vector3d(2,3,4);
w = v.multiply(0.5);
print v.list(), v.norm()
print w.list(), w.norm()
```

Here four methods are defined for the class: `__init__` is constructor of the Vector3d object, `norm` return the sum of the square of the component, `multiply` returns a new vector obtained by multiplying the present vector by a factor `c`, and `list` returns the list of the component. The first argument of all functions is `self`, which represents the object itself (equivalent to *this in C++).

15.5 Module

In Python, the program modules (functions and classes) are loaded by the import statement. For example mathematical functions, sin, cos, log, exp etc are defined in the module called "math". To use it, you do the following:

```
import math
x= pi/2
print math.sin(x), math.cos(x)
```

The prefix `math` is needed before the math function. If you hate to use the math prefix, you can do the following:

```
from math import *
x= pi/2
print sin(x), cos(x)
```

You can create your own module. If you store the above function `fact(n)` in a file "factpy", you can call it in the following form.

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

15.6 UDFManager in GOURMET

You can handle the data of UDF file by using UDFManager class. Suppose you want to handle a file as following.

```
// "file1.udf"
\begin{def}
  a:{ b1:double
      b2[]: {c1:double
             c2[]:double
            }
    }
\end{def}
\begin{data}
 a:{   1.0
       [  { 2.0, [4.0, 5.0] }
```

```
        { 3.0, [6.0, 7.0] }
      ]
   }
\end{data}
```

You can read and write the file by the following python script.

```
from UDFManager import *
uf = UDFManager('file1.udf')
print uf.get("a.b2[]")
print uf.get("a.b2[0].c2[0]"), uf.get("a.b2[0].c2[1]")
print uf.get("a.b2[].c2[]",[0,0]),uf.get("a.b2[].c2[]",[0,1])
uf.put ([-1.0, -2.0], "a.b2[0].c2[]")
uf.write()
```

Here the script:

```
uf = UDFManager('file.udf')
```

creates a UDFManager that handles 'file.udf'. The methods `get(location)` returns the data at the `location`. The location can be specified by the string form such as:

```
print uf.get("a.b2[0].c2[1]")
```

or in the form specified string and a list of integers.

```
print uf.get("a.b2[].c2[]",[0,1])
```

15.6.1 Summary of UDFManager methods

The methods of the UDFManager class are listed below. The parameter **location** is a instance of Location class. The Location class is an auxiliary class for the string operations that allows to handle a UDF variable name easily.

- UDFManager(udffilename): The UDF file specified to udffilename is opened and an UDFManager object is created.
- get(location,unit=""): Returns a value at the specified location.
- getArray(location,unit=""):Returns an array at the specified location.
- put(value,location,unit=""):The value is set to the location.
- write([filename,record,mode]):Data is written to a file.
- totalRecord():The total record count in a UDF file is returned.
- currentRecord():The current-record position is returned.
- jump(record_no_or_label):The target record is moved to the record position specified with the integer number or the label string.
- nextRecord():The target record moves to the next record.
- newRecord(label=""):A new record with the name given with the label argument is appended at the last record and the name of the record is returned.
- seek(location):The reference position of data moves to the location specified by the 'location' argument.
- tell():The current location is returned.
- next():The rightmost index included in the location of an array is increased by one.
- prev():The rightmost index included in the location of an array is decreased by one.
- type(location):The UDF data type of the specified location is returned as a string.
- size(location):If the specified location is an array, the number of elements is returned.

15.6.2 Summary of Drawing methods

When the UDFManager extension module is used in GOURMET, you can draw a 3-dimensional object on the view screen of GOURMET.

The following are the meaning of the parameters.
coordinateN(N is integer) is either a list type data with 3-dimensional coordinate value such as [x, y, z], or a location string with $ prefixed, whitch indicates the coordinate data.
coordinate_list is a list type data with 3-dimensional coordinate, such as [[x1, y1, z1], [x2,y2,z2]...].
attribute_id is an integer value or a list of floating values, which specifies the display attributes for drawing.

The methods for the drawing are listed below.

- line(coordinate1,coordinate2,[r,g,b,t])
- point(coordinate,[r,g,b,t])
- polygon(coordinate_list,[r,g,b,t])
- polyline(coordinate_list,[r,g,b,t])
- disk(coordinate1,[r,g,b,t,radius,vx,vy,vz])
- ellipse1(coordinate1,[r,g,b,t,a,b,vx,vy,vz])
- ellipse2(coordinate1,coordinate2,[r,g,b,t,a,vx,vy,vz])
- cylinder(coordinate1,coordinate2,[r,g,b,t,radius])
- sphere(coordinate1,[r,g,b,t,radius])
- ellipsoid1(coordinate1,[r,g,b,t,a,b,c,vx,vy,vz])
- ellipsoid2(coordinate1,coordinate2,[r,g,b,t,a])
- tetra(coordinate1,coordinate2,coordinate3,coordinate4,[r,g,b,t])
- cube(coordinate1,length,[r,g,b,t])
- arrow(coordinate1,coordinate2,[r,g,b,t,radius,height])
- text(coordinate,contents,[r,g,b,t,size])
- clearDraw(): All the graphic objects before calling this are erased.
- appendDraw(): Within one Python execution, the drawings after calling this are overwritten.

15.7 Bibliography

(1) Guido van Rossum. ”Python Tutorial” http://www.python.org, March 22, 2000.

(2) Mark Lutz ”Programming Python” O’Reilly & Associates, Inc, October 1996.

(3) David M. Beazley ”Python Essential Reference” New Riders Publishing, 1999.

(4) Python Language Home Page
Python can be downloaded from http://www.python.org/