OCTA

ソフトマテリアルのための統合化シミュレータ

GOURMET Primer

チュートリアル

Version 1.1

OCTA ユーザーズグループ

DEC. 25 2003

執筆者

Part I　西尾裕三, 株式会社 日本総合研究所, 土井正男, 名古屋大学
Part II　土井正男, 名古屋大学

プログラム開発者

西尾裕三, 株式会社 日本総合研究所

謝辞

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

本プログラムの改良は、独立行政法人 科学技術振興機構の補助を受け、「多階層的バイオレオシミュレータの研究開発」（研究代表者土井正男名古屋大学教授）における「バイオレオシミュレータ用プラットフォーム」機能改良ソフト開発の下で行われたものである。

目 次

図 目 次

表 目 次

第I部

GOURMETスクリプティング

第1章　はじめに: 野球シミュレータプロジェクト

この小冊子は、「あなたが GOURMET でできること」を紹介するものです。GOURMET は OCTA プロジェクトで開発されたシミュレーション・プラットフォームです。OCTA プロジェクトで開発されたシミュレーション・エンジンを使用したければ、GOURMET の使い方を学習する必要があります。GOURMET を使えば、エンジンのための入力ファイルを作成し、シミュレーション・エンジンを操作し、出力ファイルをブラウズし、データ分析やプロットを行い、3D アニメーションなどを行うことができます。

GOURMET は、柔軟でカスタマイズ可能となるように設計されており、システムを変更する様々な方法を提供しています。あなたは、使用するエンジンへの自分のインターフェースを作成でき、さらにデータ分析とデータ参照のための自分の機能を付け加えることができます。

これらの GOURMET のサービスは、OCTA プロジェクトで開発されたエンジンだけを対象とするものではありません。GOURMET は任意のエンジンのためのインターフェースとして使用することができます。そのための唯一の条件は、UDF(user definable format) と呼ばれるテキストファイル・フォーマットで入出力ファイルを書くことです。GOURMET はあなたが新しいエンジンを開発するのに有用なはずです。実際、GOURMET を使用すると、データ編集、ブラウジング、プロット、3D グラフィックスを行うような単純なプログラムを容易に作成することができます。

第 1 部では、小さなプロジェクト（野球シミュレータを作るプロジェクト）を実施することによりこれまで述べたことを実証したいと思います。私たちは、空気中で投げられたボールの軌道を計算するシミュレータを作り、GOURMET にインプリメントされている様々な機能を示します。

図 1.1 は、第 2 章を読むだけで作ることができる単純なシミュレータの出力を示しています。ここで描かれるものは、空気力学の効果が完全に無視される場合のボールの軌道です。実際のピッチングの駆け引きでは空気力学の力を利用します。次の章ではボールの回転と、地面でボールが弾むことをを考慮に入れて、投げられたボールの軌道を計算するシミュレータを作ります。その出力は図 1.2 や図 1.3 に示されています。このシミュレータを修正すれば、ゴルフシミュレータ（図 1.4）や卓球シミュレータ（図 1.5）にすることができます。

どうぞお楽しみに。

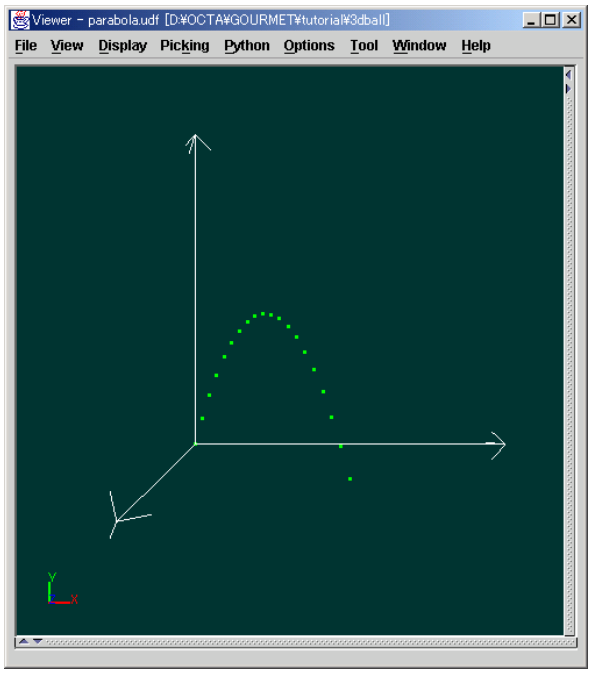

図 1.1: 放物線: 初めて使う GOURMET ビューワ

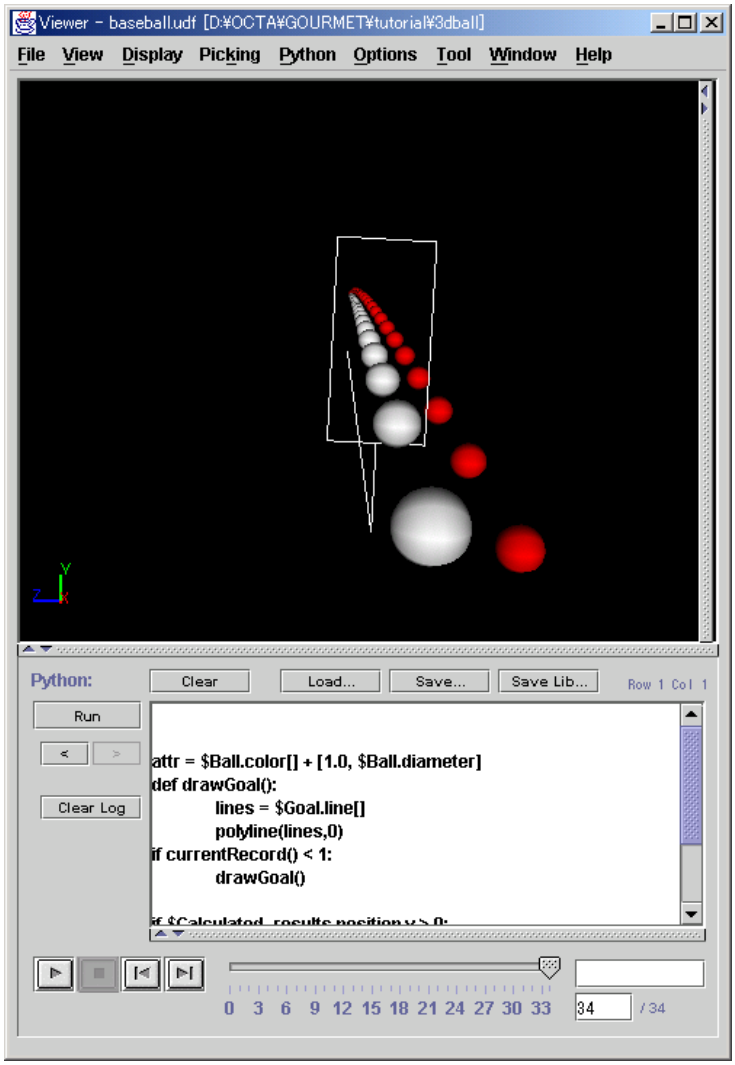

図 1.2: 速球とカーブの解析結果

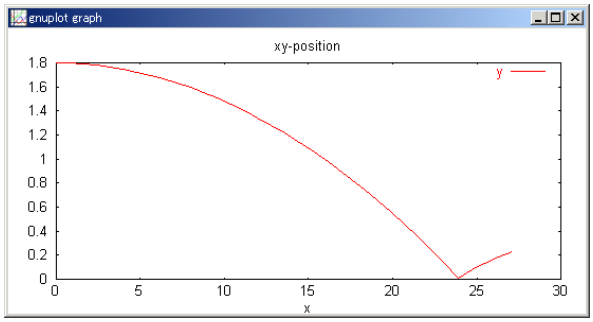

図 1.3: 速球の x-y プロット

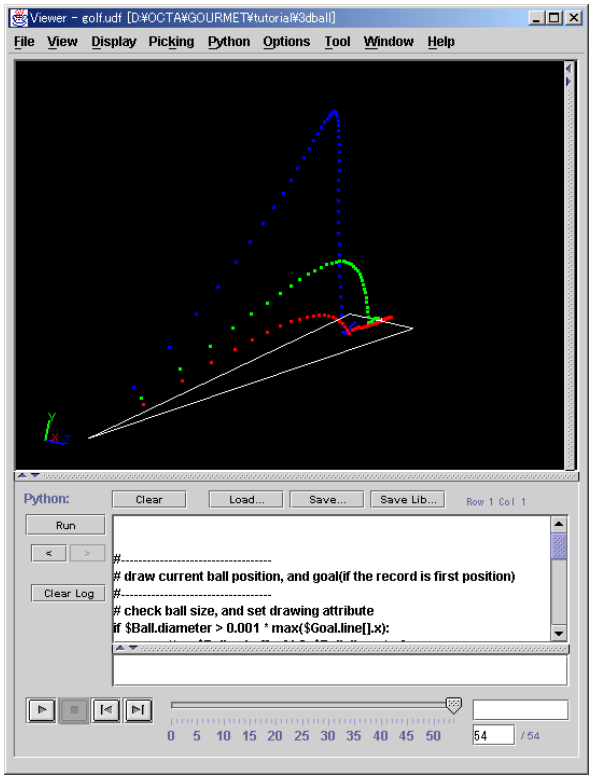

図 1.4: ゴルフ・シミュレータによる 3 次元解析結果

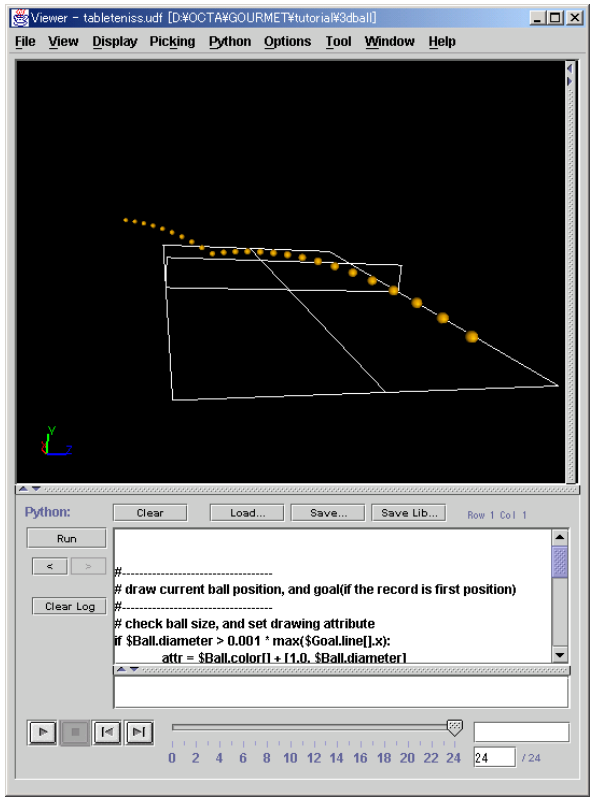

図 1.5: 卓球シミュレータによる３次元解析結果

第2章　初めてGOURMETを使う

2.1　GOURMETの起動方法

最初にGOURMETを起動して、野球プロジェクトを始めることにしましょう。Linuxをお使いの方は、次のコマンドを実行してください。

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

Windowsをお使いの方は、スタートメニューのOCTA2003-¿Start GOURMETメニューを選択してください。

または、次のコマンドを実行してください。

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

このとき、OCTAのインストール先ディレクトリの絶対パスを使うことを忘れないように。

2.2　エディタを使う

GOURMETを起動すると、エディタの初期ウィンドウが表示されます。まずGOURMETにデータファイルを教えてください。**File/Open...**メニューを使用して、"GOURMET/tutorial/3dball/parabola.udf"というUDFファイルを開いてください。（この章で使うUDFファイルは、ディレクトリ"GOURMET/tutorial/3dball/"の下に準備されています。）

図2.1のようなウインドウが表示されましたね。

今、開いた"parabola.udf"はテキストファイルです。テキストエディタで開くと、表2.1の内容が確認できます。これがUDF(User Definable Format)ファイルです。

UDFには、データの定義部分（定義部）とデータの値の部分（データ部）があります。定義部は、`\begin{def}`から`\end{def}`までの間に書かれ、データの名前と単位を定義しています。

データ部は、`\begin{def}`から`\end{def}`までの間に書かれ、データの実際の値を示しています。

定義部の例を説明しましょう。たとえば、

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

MaxTは実数型の秒[s]を表します。この内容は、GOURMETのデータウインドウに表示されています。定義部の最後の部分 `"max of calculation time"` は、ヘルプメッセージです。マウスカーソルをMaxTに合わせると、"max of calculation time" がポップアップされます。

データ部は、実際の値を表します。定義部の `Matx:float [s]` と、データ部の `MaxT:2.0000` は両方で、`MaxT` が2.0秒であることを表しています。この内容も、GOURMETのウインドウに表示されています。

"parabola.udf"では、さらに次のことも表しています。 `Velocity` は、xyzの3つの要素からなり、[m/s]の単位で表されます。GOURMETのウインドウが、図2.1の時に、`Velocity` の左の鍵アイコンをクリックするとホルダーが開き、`Velocity` のデータを見ることができます。例えば、"Velocity" のx成分は5.0[m/s]であり、これがまさに表2.1の内容となっています。図2.4は、これらの構造をすべて開いた状態を示しています。

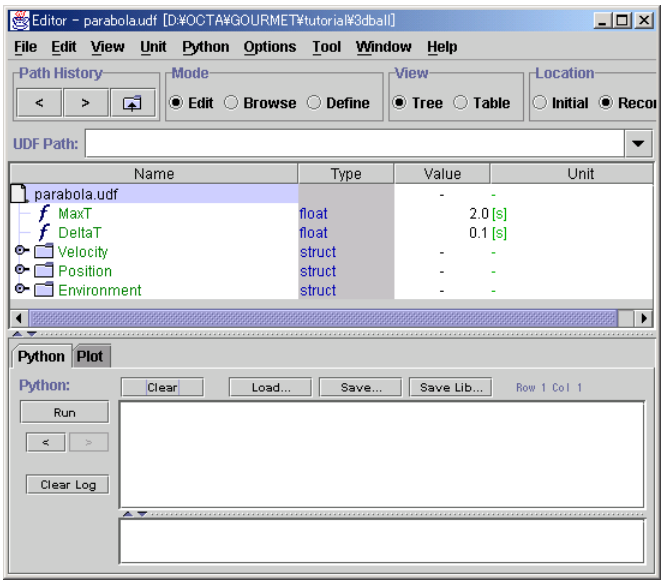

図 2.1: GOURMET エディタで開いた Parabola UDF

編集モードでは、データの値を編集できます。標準的なコピー（[Ctrl]+C)、ペースト（[Cntrl]+V）機能を使うことができます。データの数値か文字列を選択し [Ctrl]+C を押すと、その内容はクリップボードにコピーされ、[Cntrl]+V で他のアプリケーションにペーストできます。また逆に、クリップボードのテキストデータを GOURMET にペーストできます。エリアを指定して複数のデータを選択し、これをコピー・ペーストすることもできます。

次に、メニューバーの下の **Table** というラジオボタンをチェックしてください。こうすると、データを表形式で扱うことができます。（図 2.2）ここでもコピー・ペーストを試してみてください。

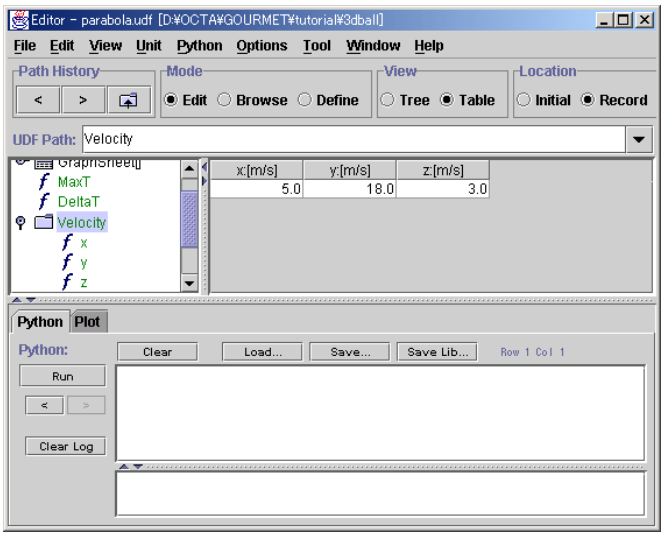

図 2.2: 表形式で表示された Parabola UDF

編集の結果を確認するために、**File** メニューの **Save As** を使って別の名前（例えば”parabola_1.udf”）で保存してください。そのファイルを他のテキストエディタで開き、変更した値を確認してみてください。

編集モードではデータ定義を変更することはできないようになっています。データ定義を変更するには、UDF ファイルをテキストエディタで直接編集するか、GOURMET の定義モードを使う必要があります。しかし、UDF と GOURMET をもっと理解してからこれらを使うようにしてください。

１つのデータを異なった単位で見ることもできます。GOUMET の Tree ビューで、MaxT の Unit 欄を右クリックすると、図 2.3 のダイアログボックスが現れます。

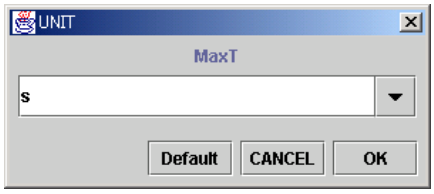

図 2.3: MaxT の Unit ダイアログボックス

このダイアログボックスの右にあるプルダウンリストから、例えば [ms] を選ぶと、MaxT の値が 2000.0 [ms] と表示されます。

このような単位の変更は、画面上の表示のみに使われ、内部では定義部で定義されている単位でデータの値が保存されています。これを確認するには、MaxT の値を 2000.0 [ms] から 4000.0 [ms] へ変更し、別ファイルに保存してみてください。データ部の値が `MaxT:4.0` となっているはずです。

GOURMET は [J] (ジュール) から [erg]（エルグ）などの単純な単位変換方法を知っています。また、[cal]、[atomic_mass]、[eV] のような単位を定義して使うことができます。これについての詳細は、**GOURMET Operations Manual** を参照してください。

ノート: ここまでに変更した UDF ファイルの内容を、最初の状態に戻すには、**File/Open** メニューで”parabola.udf” を開きなおしてください。

2.3 Python スクリプトを動かす

図 2.4 に示されているように、エディタの下部に **Python Scripting Window** と **Python Log Window** があります。**Python Scripting Window** ではどんな Python スクリプトでも実行できます。また、その結果が **Python Log Window** に表示されます。Python Scripting Window に次のスクリプトをタイプして、**Run** ボタンを押してみてください。

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

計算結果 5.34782608696 が Python Log Window に表示されましたか？

では次に、次のスクリプトを実行してみてください。

```
print $MaxT
```

その結果は、（あなたが変更していなければ）2.0 と表示されます。

さらに、UDF データに値を割り当てることもできます。次のスクリプトを実行してください。ここで、$Velocity.z は、割り当て対象が「速度」の「z」成分であることを意味します。

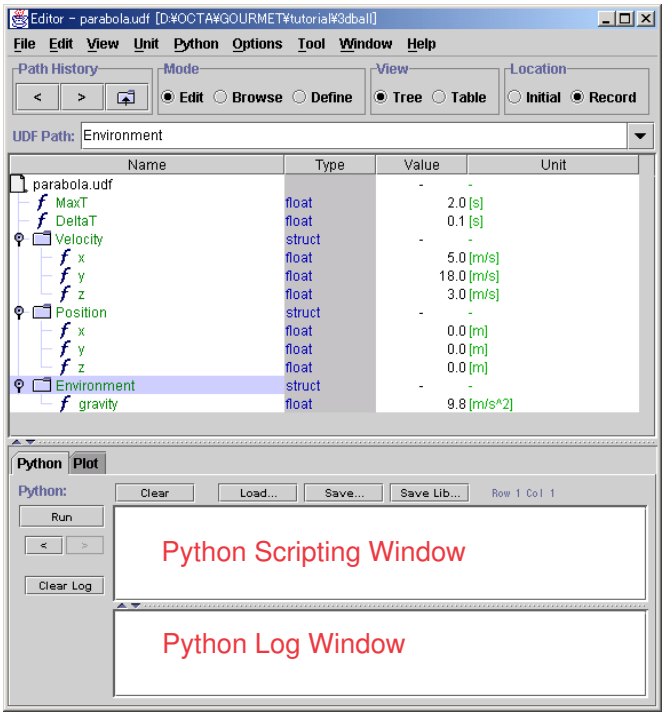

図 2.4: データ構造を開いた Parabola UDF

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

エディタの中で Velocity を開くか、Python Scripting Window で `print $Velocity.z` を実行して、結果をぜひ確認してみてください。

さて、次のスクリプトを実行してみてください。

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

次のような結果が出力されます。

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

ここで、 `[5.0,18.0, 3.0]` は Python でのリストの表示フォーマットです。リストの各要素は、C++言語と同じように `V[0],V[1]` などとして参照されます。また、次のようにして$Velocity の値を設定することもできます。

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

2.4　ビューアの使い方

さあ、ビューアの中で放物線描画スクリプトを実行しましょう。エディタの **Window/Viewer** メニューを選択すると、図 2.5 のように空白のビューアが表示されます。ビューアは次の 3 つのウィンドウを持っています。**3D Object Window**、**Python Scripting Window**、**Python Log Window** です。

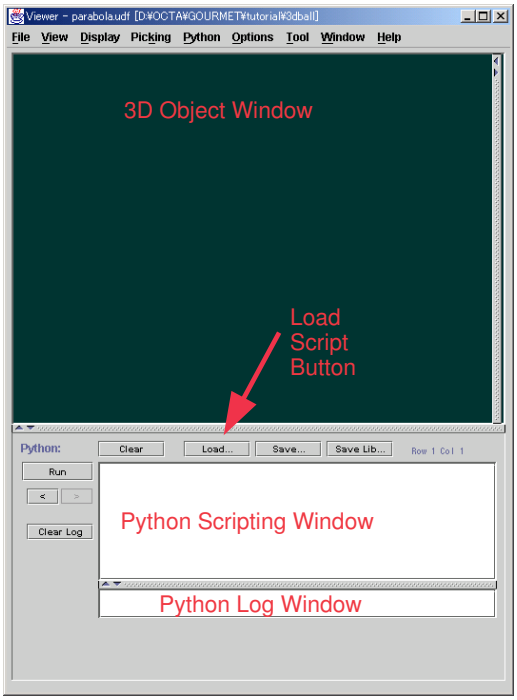

図 2.5: GOURMET の 3D ビューアを開いたところ

では、Python Scripting Window で、次のスクリプトを実行してください。図 2.6 のようなウインドウが表示されるはずです。

```
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)
```

スクリプト `sphere (r0,0)` は座標 `r0` に球体を描画します。最後の引数 0 は、半径や色などの球体の属性を表現する属性番号です。違う色で描画してみるには、`sphere (r0,1)` のように、属性番号を変えてみてください。違う半径で描画する方法については、付録 B を参照してください。

他のコマンドの意味については、わかりますね。`arrow (r0,r1,0)` は、`r0` から始まり、`r1` で終わる矢印を、属性番号 0 で描画します。`text(r0,"Origin",0)` は、”Origin”という文字列を、`r0` を先頭として属性番号 0 で描画します。他にも便利なコマンドが、付録 B や、**GOURMET Python Script Manual** で説明されています。

次に、**Load...** ボタンで描画スクリプトをロードしましょう。ファイル・オープンダイアログで、
”GOURMET/tutorial/3dball/script/parabola.py” というスクリプトファイルを選んでください。
ロードしたスクリプトが Python Scripting Window に表示されます。この内容を、表 2.2 に示しています。

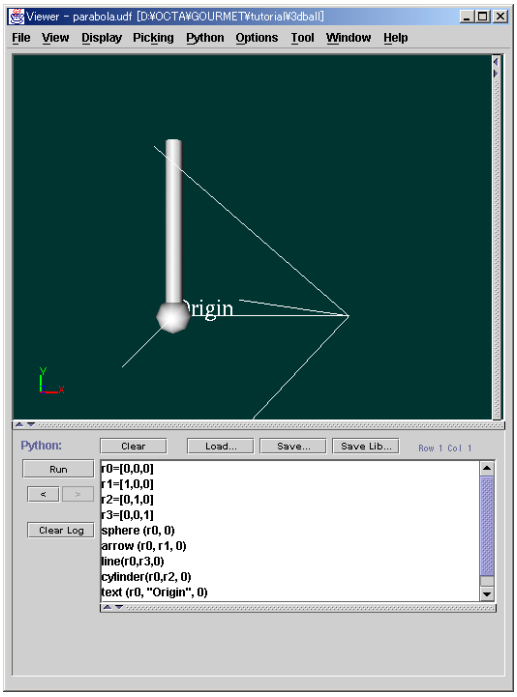

図 2.6: 3D Viewer での簡単な描画

'#' で始まる行は注釈行です。このスクリプトは 3〜6 行目で座標軸を 3 本の矢で描き、12〜14 行目でボールの座標を計算し、10〜18 行目をループにすることにより、時間が「MaxT」の値未満である間、その座標を描きます。

このスクリプトを **Run** ボタンで実行してください。3D Object Window で、図 2.7 のような 3D 放物線が描かれます。**左のマウスボタンでドラッグする**と 3D オブジェクトを回転させることができ、**View** メニューで視界方向をリセットしたり、指定した方向に設定することもできます。

エディタで parabola.udf のパラメタを変更し、ビューアで描画スクリプトを実行すると、どんな放物線でも描くことができます。ぜひ試してみてください。エディタとビューアが 1 つの UDF ファイル内容に対して連携していることがわかりますね。

あなたが変更したパラメタを保存したい場合は、エディタの **File/SaveAs...** メニューで別名で保存できます。

さて、GOURMET の基礎的な操作方法はわかりましたね。私たちの野球シミュレータを、設計し構築する方法を学習し始めましょう!

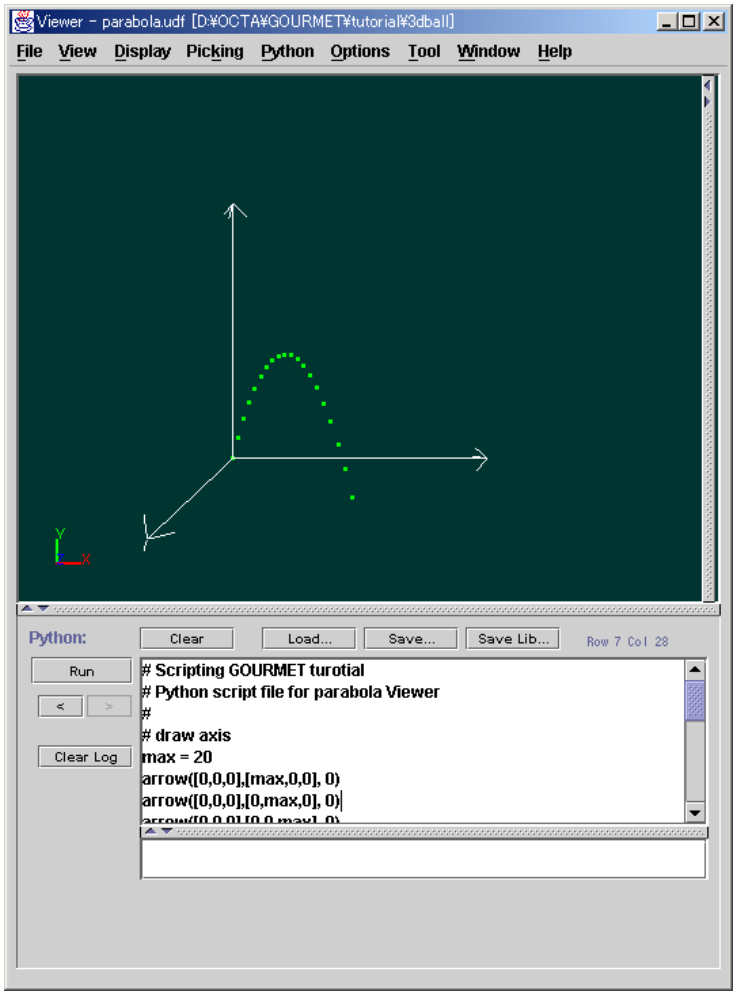

図 2.7: 放物線の描画結果

表 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}
```

表 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 --------------
```

第3章　理論的背景

私たちのボール・シミュレータの理論的な背景は以下のように要約されます。

空気中で投げられた質量 m、半径 a のボールを考えます。図 3.1 のようなデカルト座標を使います。Y 軸が地面との垂直方向、X 軸はボールの初期方向が X-Y 平面となるように取り、Z 軸は X、Y 軸に対して通常の方向に取ります。

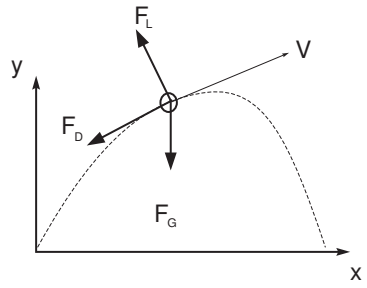

図 3.1: ボールに働く力

ボールの速度 $\boldsymbol{V}$ は、以下の運動方程式で決定されます。

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

ここで、$\boldsymbol{F}$ はボールにかかる空気力学的な力であり、第 2 項は重力の効果を表します。空気力に対しては単純な近似を行います。第一に、x 方向に速度 V で動くボールの単純な状況を考えましょう。空気がボールに対して、以下のような抵抗力を及ぼします。

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

ここで、C_D は抗力係数であり、$A = \pi a^2$ はボールの断面積です。抵抗力は進行方向 ($\boldsymbol{V}$) に対して負の方向に働きます。

ボールが z 軸の周りに ω の角速度で回転している場合、上部は $V + a\omega$ で移動するのに対し、下部は $V - a\omega$ で移動します。表面に働く空気の圧力は、表面速度の 2 乗に比例するため、表面速度の差が揚力を生成し、それは $(V + a\omega_z)^2 - (V - a\omega_z)^2 = 4Va\omega_z$ に比例します。このようにして、揚力は次のように書くことができます。

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

ここで、C_L は揚力係数です。揚力は、y 軸に対して正の方向に働きます。

一般に、ボールの速度が $\boldsymbol{V}$ であり、角速度が $bm\omega$ の時、空気力は、次のように書くことができます。

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

ボールの回転は時間経過で変化するでしょうが、我々の計算では一定と仮定します。

このようにして、　位置 $\boldsymbol{r}$ に関する運動方程式は次のように記述されます。

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

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

この方程式は、地面の条件により補足されなければなりません。我々は、ボールが地面に当たった時単純に跳ね返ると考えます。これは、次のように記述できます。

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

ここで、C_R は地面に対するボールの反発係数です。

第4章　データ構造

さあ、我々の野球シミュレータの UDF ファイルに取り掛かりましょう。このチュートリアル中の最終の UDF 定義は、”GOURMET/tutorial/3dball/3dball.def”にありますが、テキストエディタで定義をタイプすることを推奨します。

4.1　単位の定義

最初に、我々のシミュレータで使用する単位を定義しなければなりません。私たちはシミュレータの中で SI（国際単位系）を使用することにします。GOURMET の内部単位系は SI なので、この場合定義する必要はありません。でも、距離を「マイル」や「ヤード」で見たい人のために、これらを定義してみましょう。定義は表 4.1 のようになります。

表 4.1: 単位の定義

```
\begin{unit}
     PI=3.141592
     [rps]=2.0*PI[rad/s]
     [rpm]=1.0/60.0*[rps]
     [yard]=0.9144[m]
     [mile]=1760 [yard]
\end{unit}
```

簡単でしょう。 [rpm] の定義でわかるように、前に定義した単位を使うことができます。単位定義のルールについては、**UDF Syntax Reference** で記述されています。

4.2　クラスの定義

座標、速度、力などに使用される、3 次元のベクトル・クラスを表 4.2 のように定義します。

このクラス定義は、３つの要素　x、y、z　を持つ量を定義しており、最後にある [unit] によって、これらの３要素が同じ単位を持つことを表しています。

このクラスが定義されていれば、次のような定義を行うことができます。

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

クラス名の後ろに添えられた [unit] が、各データ要素の単位を決定していることに注意してください。

表 4.2: クラスの定義

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

4.3　野球シミュレータの定義部

4.3.1　定数データの定義

野球シミュレータに現れる物理量を定義することにしましょう。データの定義部は、\begin{def}から\end{def}まで、もしくは、\begin{global_def}から\end{global_def}までの間に書かれます。これら 2 つの記述方法の違いは後で説明することにしましょう。簡単に言ってしまえば、変化しない定数データは global_def で定義し、解析が進むと共に変化するデータは def で定義すればよいでしょう。(global_def は、データアクセス速度を向上させるために最近導入されたもので、あなたのプログラムには影響しません。)

ここでは、私たちは解析中の定数となっている物理量を global_def に定義することにします。この定義内容は、表 4.3 に示されています。

定義内容は明らかでしょう。

- データ Ball は、ボールの質量や直径で構成されています。
- データ Environment は、ボールの環境を表しており、重力、空気密度、地面の反射係数で構成されています。
- Parameter は、空気力学係数 C_D 、C_L を含んでいます。
- Solver は、時間積分の数値解析や解析結果の出力間隔に必要なパラメータからなっています。
- Initial_condition は、ボールの初期の座標、速度、スピンからなっています。前節 4.2 で導入した Vector3D クラスを使用することにより定義が簡単になっていることに注意してください。

4.3.2　解析結果データの定義

我々の野球シミュレータの出力結果は、ボールの座標と速度の時間発展です。GOURMET は、システムの時間発展の間を通じて動くサービスを提供します。そのサービスとは、システムの状態を各レコードに書き込むことです。UDF ファイルのレコードはある時間ステップ（または応力解析の負荷のように変化する入力パラメータの値）における関連したデータの集合です。時間発展データを出力するには、新しいレコードを生成して現在のデータを出力し時間ステップを進めます。 \begin{def} から \end{def}までに定義されているデータは各レコードに書かれます。一方、 \begin{global_def} から \end{global_def}までに定義されているデータは全てのレコードに共通な 1 つの領域に書き込まれます。

ボールの座標の時間発展を見ることのできるように、 Calculated_result は、 \begin{def}から \end{def}の間に定義する必要があります。もしも、これを \begin{global_def} から \end{global_def}で定義した場合、前の時間ステップの結果を書き直してしまうことになります。

表 4.3: 定数データの定義

```
\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}
```

このようにして、データ `Calculated_result` は、表 4.4 のように定義されます。

4.4 野球シミュレータのデータ部

定義部を作り終えたので、データ部の作成に移りましょう。表 4.5 は、完全な UDF ファイル”baseball.udf”の内容を示しています。ここでは、定義部をそのまま書く替わりに、最初の行にある挿入コマンドを使って、ファイル”3dball.def”から定義内容を読み込んでいます。これで GOURMET で”baseball.udf”ファイルを開いたとき、include された定義は分析され、”3dball.def”の内容が、”baseball.udf”ファイル内にもともとあったかのように扱われます。

これで、あなたの UDF は完成しました。適当な名前でファイルを保存して、GOURMET で開いてみてください。もしあなたがなにか間違っていたら、GOURMET はエラーメッセージを表示します。データ項目を開いたりして、データ構造が思ったように表示されているかをチェックしてみてください。また、名前やポップアップされるコメントが正しいかどうかチェックしてください。

このセクション中の最終の UDF ファイルは、”GOURMET/tutorial/3dball/baseball.udf”にあります。このテキストファイルを直接に読むか、あるいは GOURMET の Editor によってブラウズすることができます。こ

表 4.4: 解析結果データの定義

```
\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}
```

のセクションに定義されていないデータに関しては、第 7 章の中で説明することにしましょう。

表 4.5: Baseball.udf

```
\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}
```

第5章　計算するには

5.1　Python スクリプトをロードするには

さあ、Python プログラミングを始めましょう。UDF ファイル”GOURMET/turorial/3dball/baseball.udf”を開いてください。Python パネルの Load ボタンで Python スクリプト”GOURMET/turorial/3dball/script/calc.py”をロードしてください。

5.1.1　計算用スクリプトの内容

計算用スクリプト”calc.py”の内容を、表 5.1 に示しています。

このスクリプトの前半部品は、UDF データからの python 変数を割り当てます。また、後の半分は、第 3 章の中で示された方程式を使用して、ボールの座標を計算します。

注意: python スクリプトにおいて行の字下げは大変重要です。Python パーサは、タブとスペースを区別します。整合性のあるインデントを入力するためにはスペースではなく、タブを使用してください。

表 5.1: 計算に使う python スクリプト

```
#============================================
# 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　計算用スクリプトの実行

さて、Python パネル中の Run ボタンを押すと、このスクリプトが実行されその結果は Python ログウインドウ表示されます。結果データは `Calculated_results` に保存されているので、エディタで時刻、速度、座標、力などの結果を見ることができます。また、Record スライダーを動かして時間変化を見ることができます。

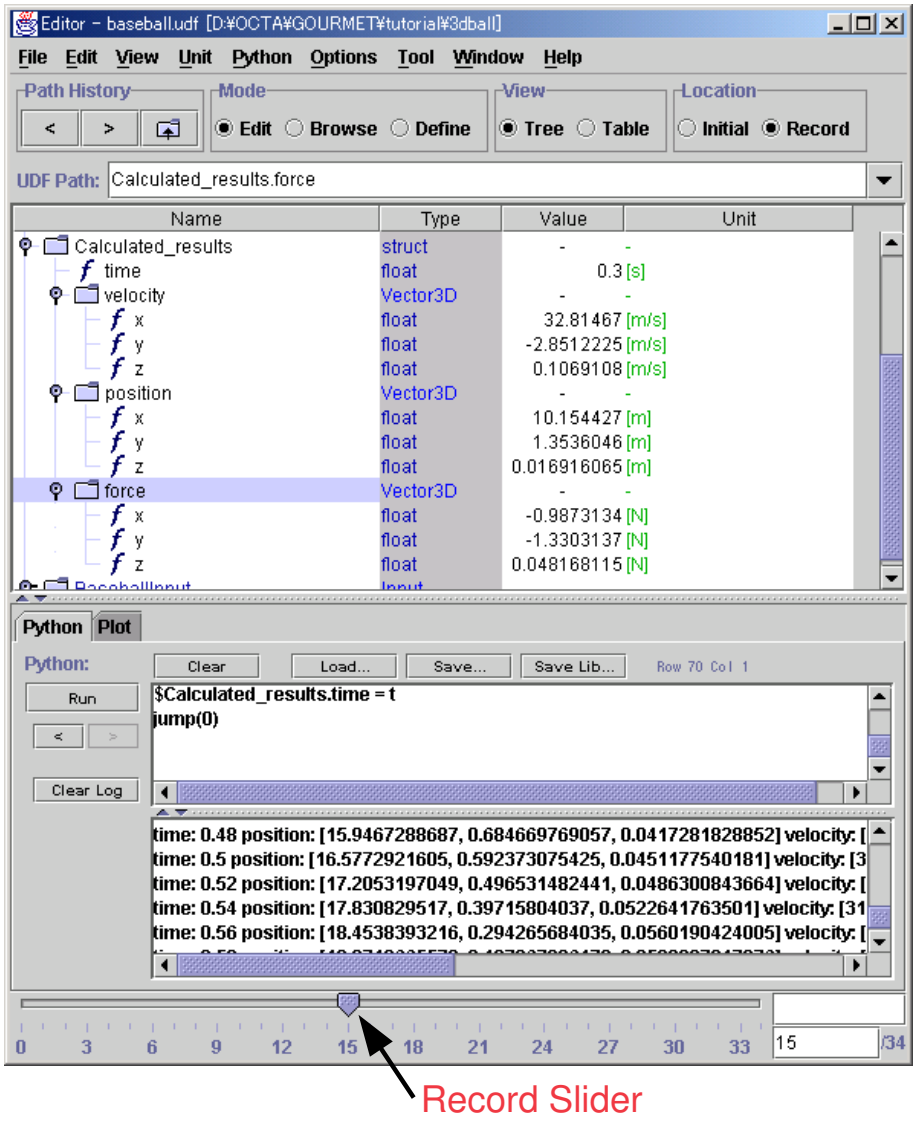

図 5.1: 計算結果の参照

5.3　計算結果を 3D で見るには

さて、これで計算結果をビューワで 3 次元表示できるようになりました。エディタの Window/Viewer メニューから Viewer を選択すると、ビューワが開きます。ビューワの Python パネル中の Load... ボタンで、”GOURMET/turorial/3dball/script/show.py” という Python スクリプトファイルをロードしてください。

5.3.1 3D 描画用の python スクリプト

”show.py”の内容が、表 5.2 に示されています。このスクリプトは、現在のレコードにおけるボールの位置のスナップショットを描画します。polyline(),sphere(),appendDraw() 等は、GOURMET に組み込みの描画関数です。

表 5.2: 3D 描画用の python スクリプト

```
# 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　3D 描画スクリプトの実行方法

さて、レコードスライダーを 0 に設定してから、Python パネルの Run ボタンを使って、ロードされている python スクリプトを実行してみてください。

最初に、ストライクゾーンおよびボールの初期位置が表示されます。
ビューア・ウィンドウの左下中のスタート・アニメーション・ボタンをクリックしてください。図 5.2 のように、ストロボ写真のような結果が得られます。
マウスの左ボタンでドラッグして視界を回転させたり、右ボタンでドラッグしてズームしたりしてみてください。また、View/Reset メニューで視界をもとにもどすことができます。

ノート: このストロボ写真のような効果は、appendDraw 関数により実現されています。

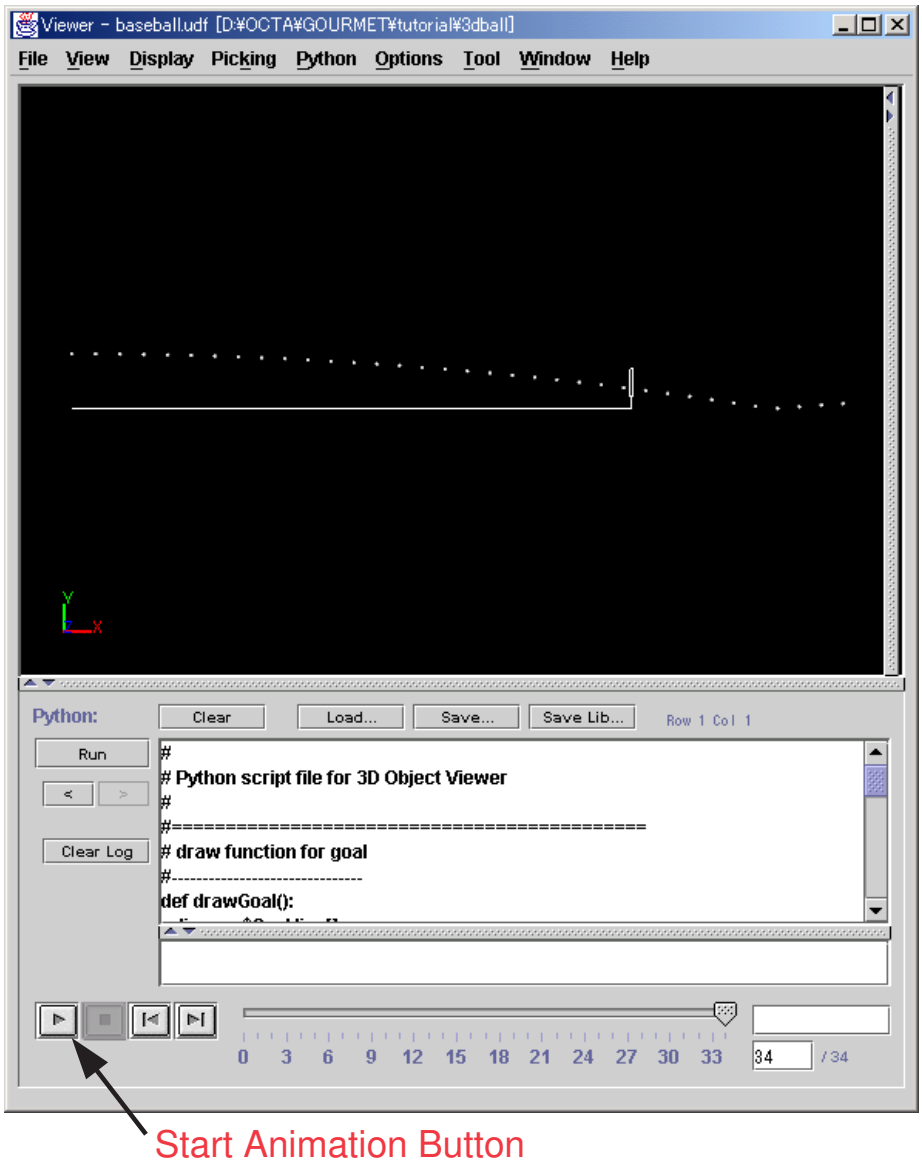

図 5.2: 3D ビューでの描画結果

第6章　グラフをプロットするには

6.1　Using GraphSheet

gnuplot アプリケーションを使えば、データを 2D や 3D のグラフでプロットすることができます。これを行うには、まずプロット・データを作っておく必要があります。GOURMET は、**エディタの Table ビュー**中の **GraphSheet** という特別のワーク・シートからプロット・データを作り、gnuplot と連携することにより、あなたを支援します。

6.1.1　GraphSheet に時系列データを追加する

さっそく、この機能を使って見ましょう。エディタに戻ってから、**エディタの python パネル**の中のロード・ボタンを使用して、
”GOURMET/turorial/3dball/script/addplot.py” という名前の python スクリプトをロードしてください。
”addplot.py”の内容が、表 6.1 に示されています。このスクリプトは、GraphSheet オブジェクトに’x’ という名前のカラムを加えて, 計算されたすべてのレコードの x 座標のデータをそのカラムへ付け加えます。

表 6.1: GraphSheet に時系列データを追加する python スクリプト

```
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)
```

スクリプトを実行し, **GraphSheet** を開いて内容を見てください。図 6.1 のように、時系列データが GraphSheet に加えられていますか？

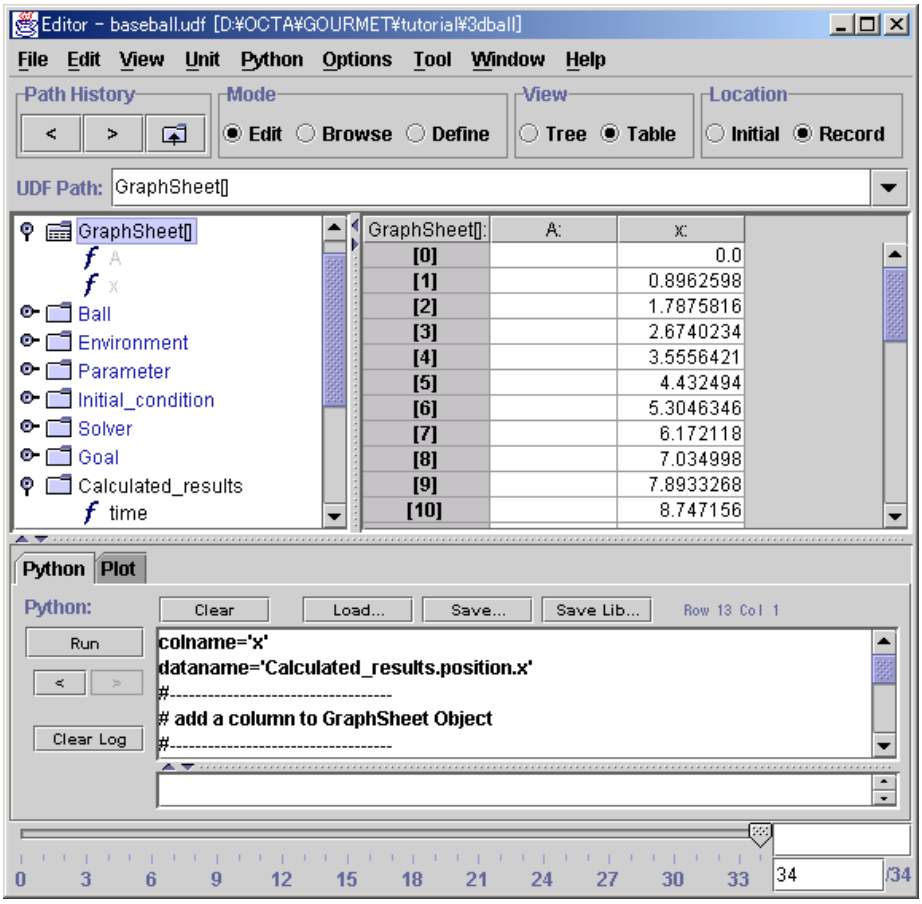

図 6.1: GraphSheet に時系列データを追加するには

このスクリプトの中の'colname' および'dataname を変更することにより、GraphSheet Objec に y-座標および z-座標用のデータを加えてみてください。

これでプロットのための時系列データが作成できたので、次の節で説明するプロット・ツールを使用ことができます。

6.1.2　プロットツールを使う

最初に、**Python Tab** の隣の **Plot tab** を押してください。これで、Plot パネルが開きました。次に、Plot パネルの **Make ボタン**を押すと、図 6.2 の様に生成されたプロット・スクリプトが、**Plot Scripting Window** に表示されます。このスクリプトは、gnuplot のコマンドです。このまま **Plot ボタン**を押すと、図 6.3 のようなプロットが行われます。GraphSheet のデータが実行ディレクトリ（GOURMET/bin）のファイル"plot.dat" に書き出され、Plot Scripting Window のプロット・スクリプトが gnuplot に実行されたわけです。テキストエディタで"plot.dat" ファイルを開けば、データがどのように変換されたかを見ることができます。最初のカラム、[0], [1],[2]... が"plot.dat"では、0,1,2 ... のように書き出されているのに注意してください。

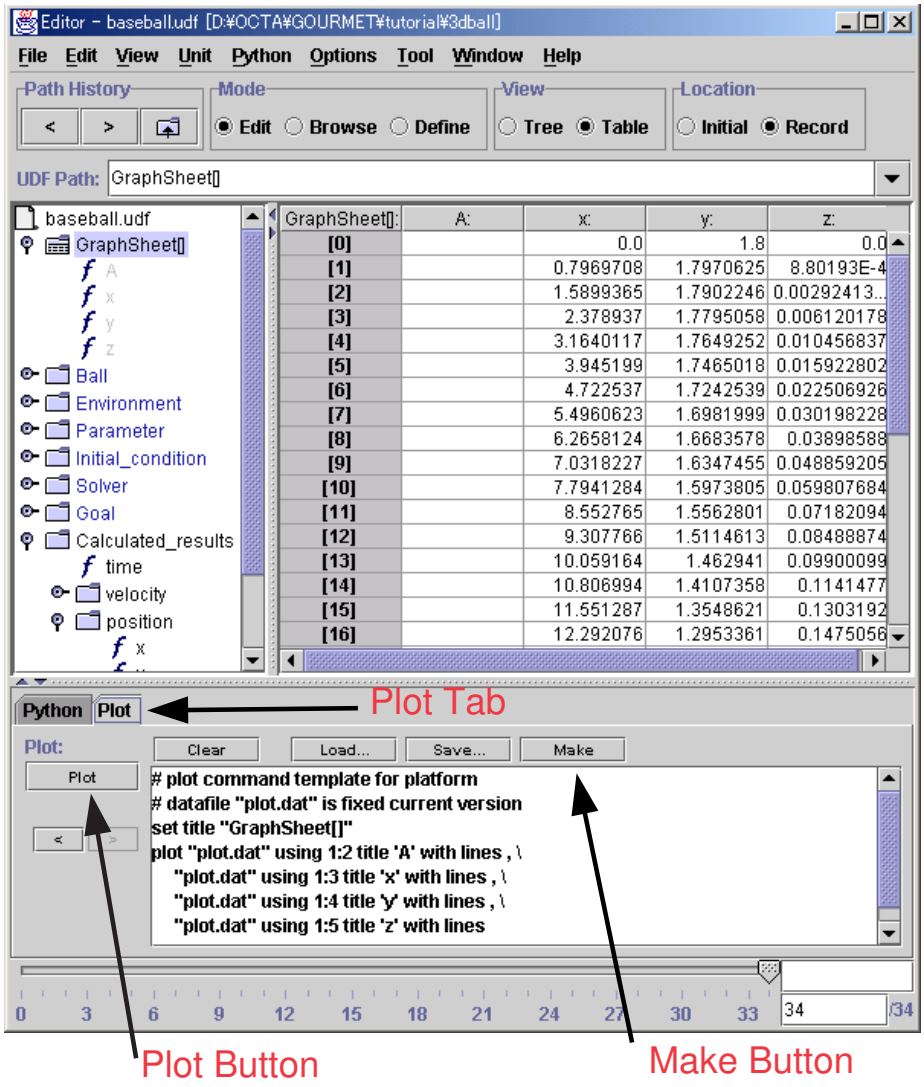

図 6.2: GraphSheet からプロットコマンドを作る

x に対し y と z をプロットしてみたければ、以下のようにプロット・スクリプトを編集してから、**Plot ボタン**を押してください。

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

図 6.4 のような gnuplot ウィンドウが表示されます。

ノート: gnuplot ウインドウの左上の gnuplot アイコン  のポップ・アップメニュー中の"Command Line"の選択により、gnuplot コマンド・ウィンドウを使用して、gnuplot の機能をすべて使用することが可能です。

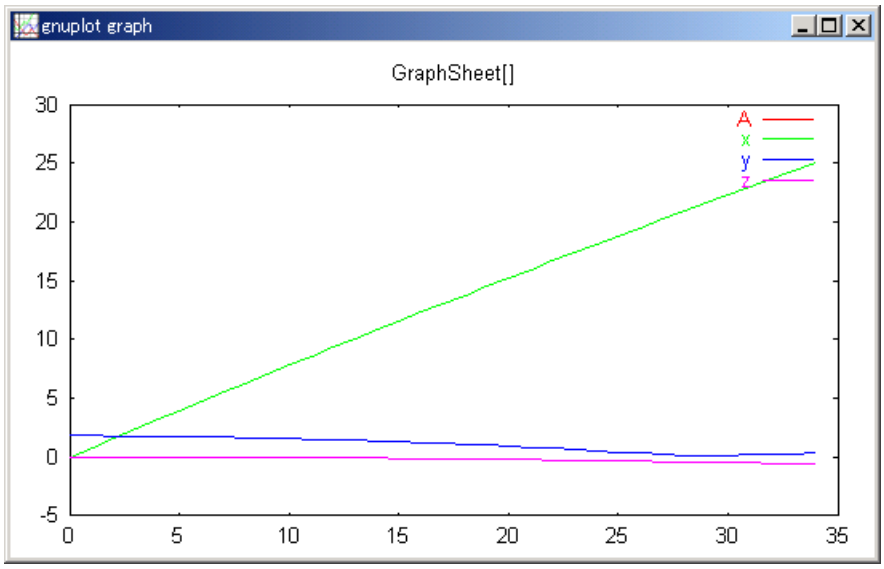

図 6.3: プロット・ツールをそのまま使うと...

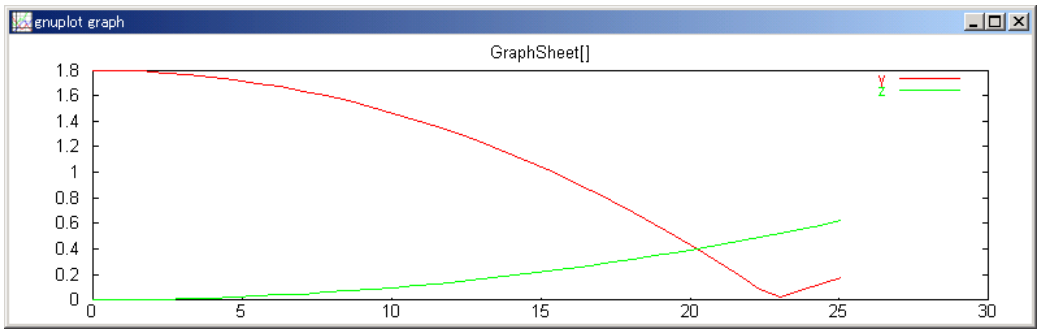

図 6.4: 編集後のプロット・スクリプトから起動された gnuplot ウインドウ

6.2 プロットライブラリの使い方

同じプロット処理を、python scripting window でも行うことができます。Python Panel に戻って、”GOURMET/tutorial/3dball/script/plotlib.py” というスクリプトをロードして、実行してみてください。これだけで、すぐに図 6.5 に示すような結果を得ることができます。

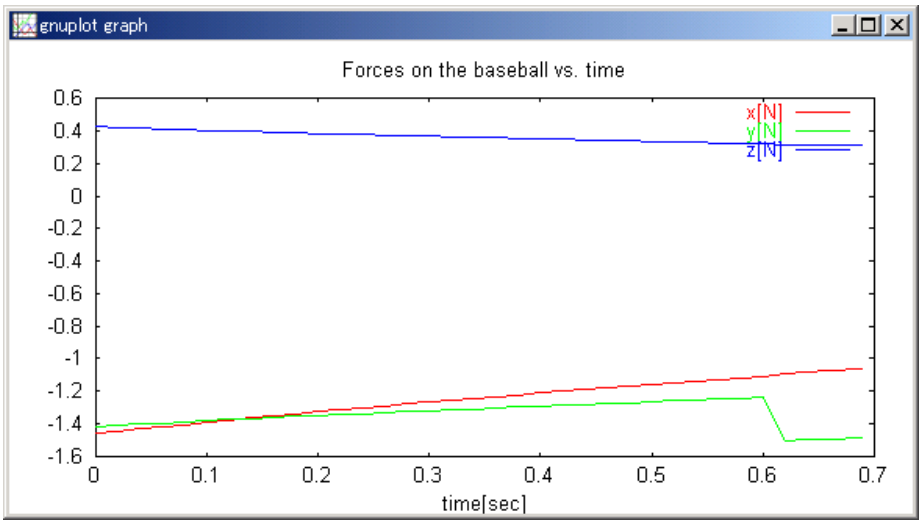

図 6.5: プロット・ライブラリを使って起動された gnuplot ウインドウ

このスクリプトの内容を、表 6.2 に示しています。

これは、”gnuplot” という python モジュールを使っています。データをプロットするには、先にデータを `axis[], x[],...` のようなリストに入れておき、関数 `gnuplot.plot` を呼び出します。この関数は、データをファイルに書き出し、引数で与えた gnuplot のコマンドを実行します。

表 6.2: 力の時間変化をプロットする python スクリプト

```
# 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')
```

第7章　アクション

7.1　アクションとは

本章では、GOUMET において Python スクリプトを実行する別の方法を紹介しましょう。それは、”アクション”と呼ばれます。アクションは UDF ファイルのデータオブジェクトにアタッチされた Python スクリプトで、そのデータオブジェクトをマウスで右クリップすると起動します。いろんな目的でアクションを使うことができます。例えば、入力データを自動的に埋める、データの統計解析を始める、配列データのプロットを始める、データを 3D で見る、などです。アクションは、GOURMET をあなた用にカスタマイズするためのツールと言えるでしょう。

7.2　アクションの実行方法

まず、どうやって使うのかを見てみましょう。

”GOURMET/tutorial/3dball/baseball.udf”という UDF を、エディタで開いてください。 `Ball` データを選択し、マウスの右ボタンでクリックしてください。表示されたポップアップメニューの **setColor** を選択し、ダイアログで色を選んでください。その後、 `Ball.color[0]`、`Ball.color[1]` や `Ball.color[2]` を見ると、データが変更されているのがわかるでしょう。

もうひとつアクション実行を行ってみましょう。次の、操作はカーブをシミュレーションするものです。

1. **BaseballInput** オブジェクトをマウスの右ボタンで選択し、ポップアップメニューから **setSimpleCondition** アクションを選んでください。
2. アクション引数ダイアログで、**CurveBall** を選んでください。
3. **Solver** オブジェクトをマウスの右ボタンで選択し、ポップアップメニューから choose **calculate** アクションを選んでください。
4. (トップにある) **baseball.udf** をマウスの右ボタンで選択し、ポップアップメニューから **show** アクションを選んでください。
5. Python パネルの Start ボタンで、アニメーションを行ってください。

図 7.1 の結果を得るまでに、たった５回の操作しか必要なかったでしょう！他のボールも投げてみてください。ストライクを取れましたか？

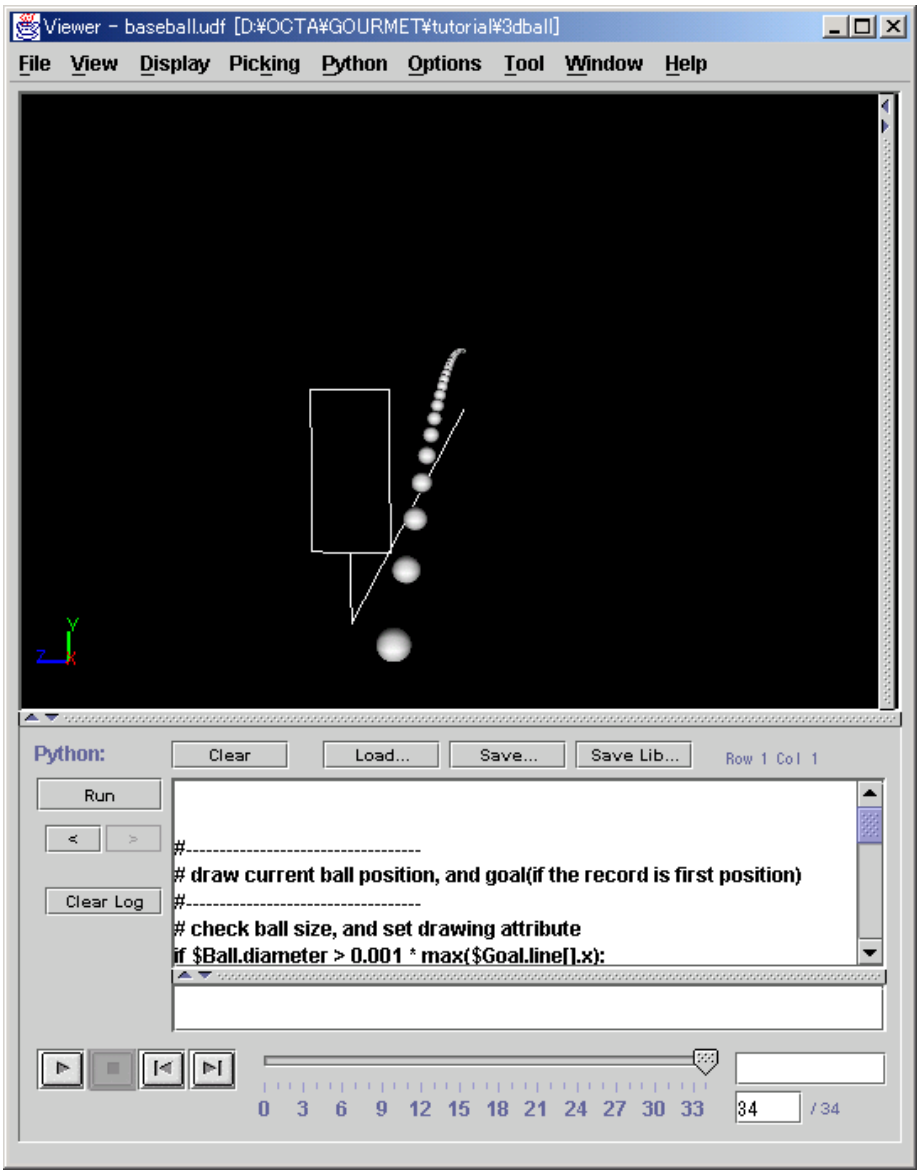

図 7.1: アクションから起動されたカーブの 3D 描画結果

7.3 アクションの定義方法

アクションは 1 つ以上のアクション・ファイルに定義され、UDF ヘッダ・データからリンクされます。エディタの中の File/Header メニューを選んでください。図 7.2 のように、現在の UDF ヘッダ・データを見ることができます。

ノート: アクションファイル･ディレクトリのルート･パスはデフォルト値として定義された”GOURMET/action/”です。また、環境変数 (”UDF_ACTION_PATH”) の使用により別のディレクトリを追加することができます。

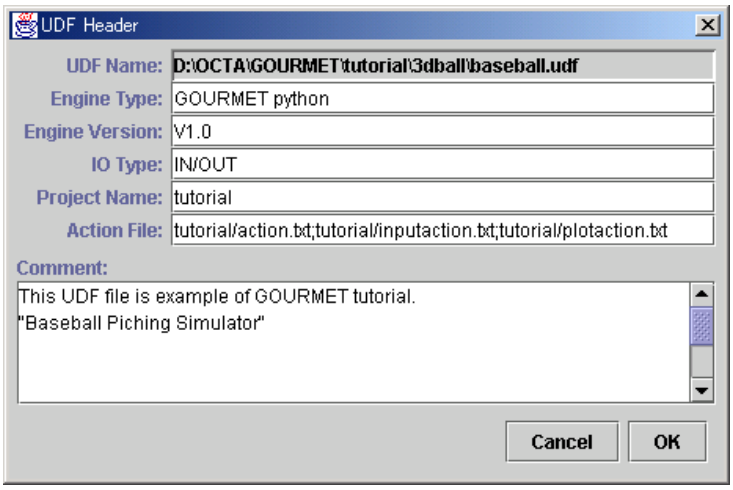

図 7.2: UDF ヘッダ・ダイアログ

この例では、baseball.udf は以下に示された 3 つのアクションファイルとリンクしています。

- **tutorial/action.txt**
  基本的なアクション (7.3.1 と 7.4.1 で説明されています)
- **tutorial/inputaction.txt**
  入力データを作成するアクション (7.4.2 と 7.4.3 で説明されます)
- **tutorial/plotaction.txt**
  たくさんのプロットアクション (ご自分で調べてください)

7.3.1 アクションの定義例

アクションファイル”GOURMET/action/tutorial/action.txt” は次のアクションを定義しています。

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

- Solver : calculate ()
- : show()

次の定義は、 Ball データにアタッチされた setColor アクションの中身です。

```
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
```

最初の行：

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

は、アクションのインタフェースを定義しています。引数はアクション引数ダイアログで入力される値となります。BallColor="white|blue|green|red" は、 BallColor が "white" または "blue" または "green" または "red" であることを示しています。 \begin から \end までが Python スクリプトであり、アクション引数ダイアログで OK ボタンを押した時に実行されます。アクションのメカニズムをもうすこし説明しましょう。一般的には、アクションの記述ルールは以下の通りです。

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

アクションが起動されたとき, アクション引数ダイアログが引数の名前と値をテーブル形式で表示します。引数が文字列であり文字'|' で区切られている場合、前の setBallColor の例のように選択メニューの定義とみなされます。アクション引数ダイアログで OK ボタンを押した時に Python スクリプトの中の引き数名と同じテキストが全て引数の値と置き換えられて実行されます。

"autorun"は、UDF ファイルが開かれた時およびリロードされた時に実行される特別のアクションで、次のように記述されます。

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

7.4 野球シミュレータのアクション

この節では、野球シミュレータ用に実装してあるアクション内容のポイントを説明することにしましょう。

7.4.1 基本的なアクション

表 7.1 に、”action.txt”の内容を示しています。このアクション・ファイルの終わりにある show アクションと culculate アクションを見てください。その python スクリプト部分は、第 5 で説明したものと全く同じです。つまり、既に作ってある python スクリプトを使用して、新しいアクションを簡単に定義することができるわけです。

表 7.1: 基本的なアクション

```
autorun : initialize() : \begin
#-----------------------------------
# autorun action for UDF file
# this action executed when the UDF file is opened or reloaded.
# You can define python functions here.
#-----------------------------------
# import modules
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 簡易設定アクション

私たちの 3D ボール・エンジンは、表 4.3 で説明した初期条件として次のデータを必要とします。

```
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"
```

表 7.2 に示されるように、アクション引き数ダイアログでメニューを選んだ時、このアクションはあらかじめ定められたデータを選択し、Initial_condition オブジェクトに設定します。

表 7.2: setSimpleCondition アクション

```
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
```

ノート: アクションが実行される前に、キーワード’self’ はターゲット・オブジェクト名に置換されます。この例では、’$self.type’ は’$BaseballInput.type’ に置換されます.

ここでわざわざ BaseballInput オブジェクトを使って入力アクションを定義した理由は次のサブセクションで説明することにしましょう。

7.4.3 詳細設定アクション

ここでは初期条件を詳細に設定するために使用される、より一般的なアクションを定義してみます。

アクション (Initial_condition : setDetailCondition) を実行すると、図 7.3 のようなアクション引数ダイアログが現れます。

setDetailCondition アクションの内容は、表 7.3 に示してあります。Initial_condition.spin の値がボール表面での速度であることに注意してください。でも、ほとんどのユーザは回転率 (rps など) によって値を入力する必要があるでしょう。したがって、以下のような Input クラスとを定義し、さらにこのクラスを使用して（野球の入力専用の）BaseballInput オブジェクトを定義したわけです。

同様な理由で Initial_condition.velocity を計算するために Input.alpha と Input.beta を定義しています。

```
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"
```

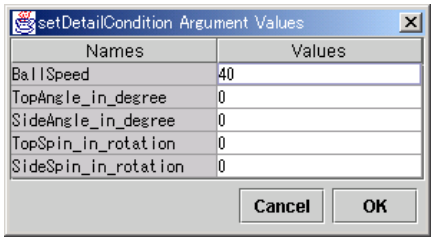

図 7.3: 詳細設定アクションダイアログ

```
  beta:float [degree] "initial throw angle to z-axis"
  rotation:Vector3D [rps] "initial rotation rate in each axis"
} "input tool for initial condition"
```

ノート: 角度の単位変換は、[rps] の単位定義（表 4.1）およびビルトイン単位 [rad] を使用して行われます。単位を変換するためにあなたがしなければならないのは、入力値の中で使用される単位を定義し、get 関数によって（変換先単位を指定して）値を取り出すことです。

表 7.3: setDetailCondition アクション

```
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
```

第8章　もっともっと

野球シミュレータができたら、これを変更してゴルフ・シミュレータや卓球シミュレータのような別のシミュレータを作ることができます。ここでは、いくつかのヒントをお教えしましょう。

8.1　Golf simulator

ゴルフ・シミュレータの UDF ファイルは”GOURMET/tutorial/3dball/golg.udf”にあります。

GolfInput オブジェクトは、表 8.1 のように 2 つのアクションを持っています。ゴルフ・シミュレータの setDetailCondition は、野球シミュレータと少し異なっていることに注意してください。つまり、ゴルフ・シミュレータではボールの初期角度とスピンは使用されるクラブのロフトによって決定されます。

ボールの初期の速度は、クラブ・ヘッドの速度とショットの中で使用されるパーシモンやアイアンのようなクラブのタイプによって決定されます。ここでは慣性モーメントを考慮したより複雑な計算を使用する代わりに、この効果を考慮するために、打撃係数を使用しています。

表 8.1: GolfInput のアクション定義

```
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
```

シンプルな入力と詳細な入力のためにこれらの 2 つのタイプのアクションを使用できます。これらのアクションダイアログを、図 8.1 と図 8.2 に示しています。

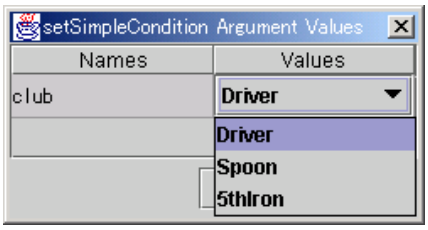

図 8.1: GolfInput の簡易設定アクションダイアログ

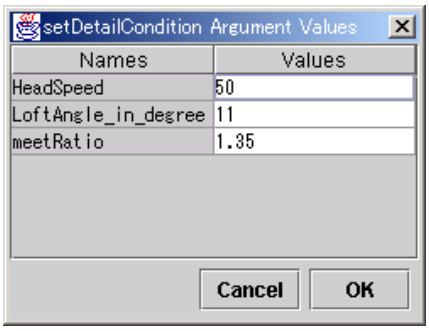

図 8.2: GolfInput の詳細設定アクションダイアログ

ゴルフ・シミュレータを使用すると、図 8.3 のように、ロング・ショットのためにどのタイプのクラブを使用するか決めることができるかも (^^;

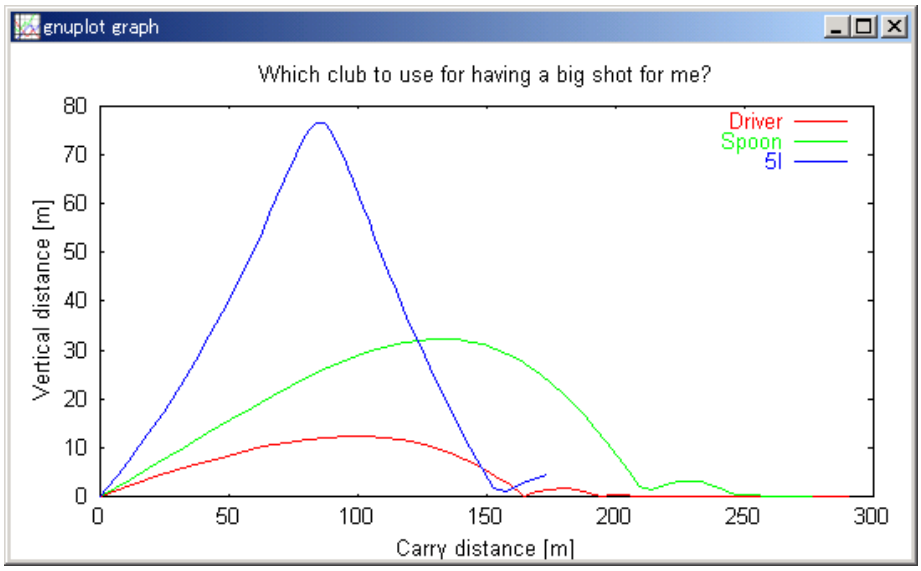

図 8.3: あなたはどのクラブを使うのが正解？

8.2 他にも

8.2.1 卓球シミュレータ

卓球は流体力によって強く影響を受けるスポーツです。

この UDF ファイルは、”GOURMET/tutorial/3dball/tabletennis.udf”にあります。野球シミュレータと同じ方法を使って、強打/カットなどのようなサービスを行うために入力アクションを構築してください。

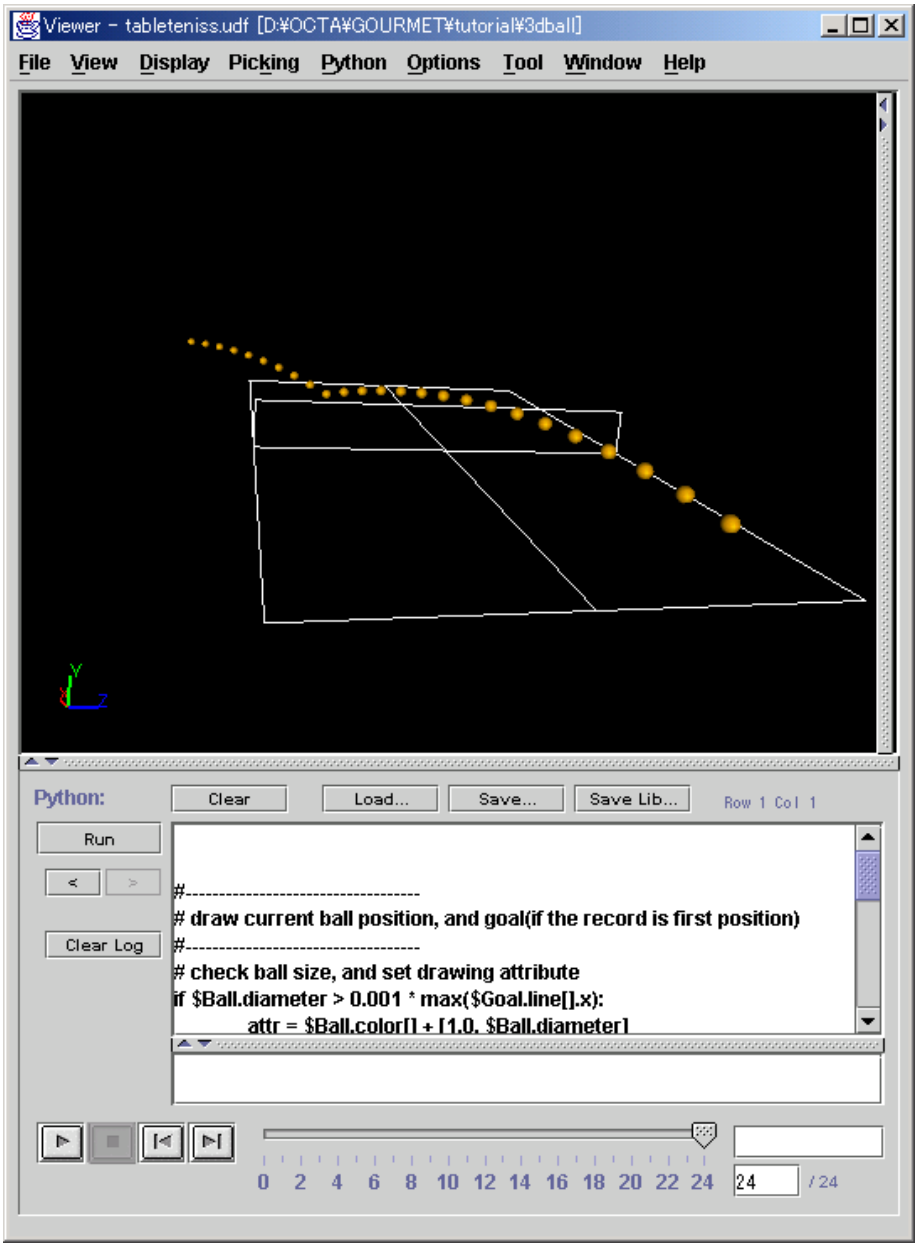

図 8.4: 卓球サーブの 3D 描画結果

8.2.2 テニス・シミュレータ

テニスと卓球の重要な違いの 1 つにボールとコートの表面の間の反発があります。

第 3 章では、ボールのスピンによって引き起こされる地面からの反動を考慮しませんでした。したがって、このシミュレータはテニス・プレーヤには絶対満足できないでしょう。

あなた独自の変更で、彼らを満足させるシミュレータができたら、ぜひとも教えてください。

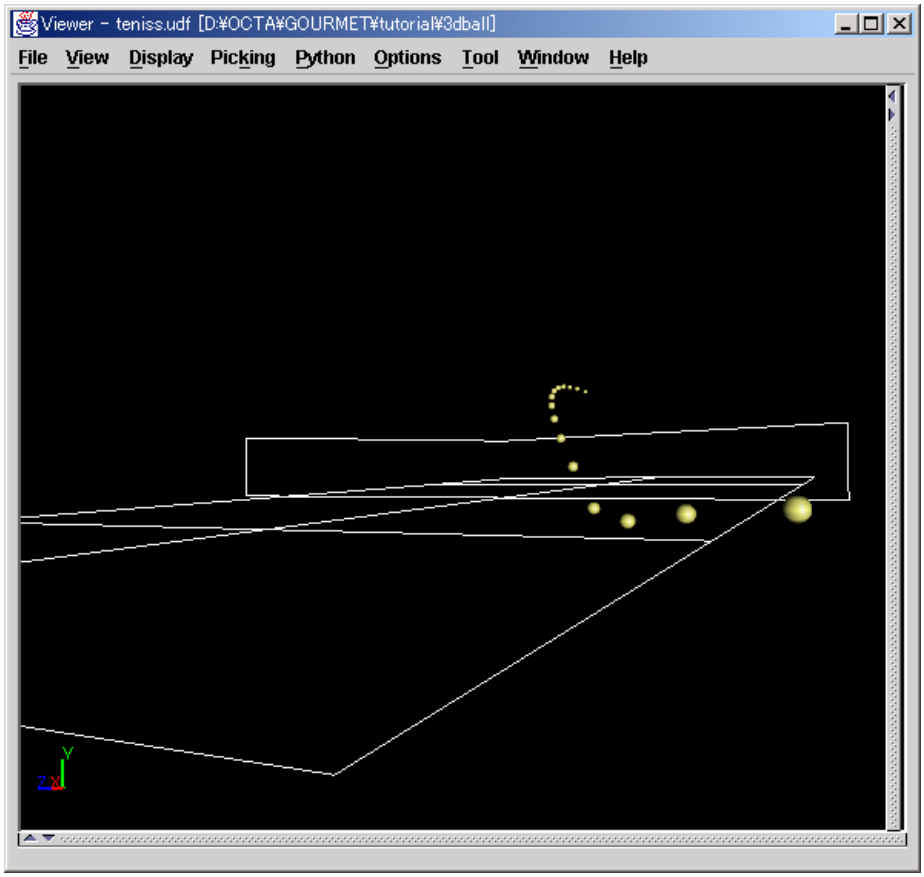

図 8.5: テニス・サーブの 3D 描画結果

8.3 さてこれから

このチュートリアルがお役に立ったなら幸いです。

GOURMET は、計算された結果を分析したり、新しいシミュレータを構築することを容易にするために様々な機能を持っています。より詳細に関してはユーザーズマニュアルを参照してください。

- **GOURMET Operations Manual:** GOURMET のユーザ向け操作マニュアル

新しいエンジンを開発するつもりならば、次の文書を読むべきです。

- **Programming UDF:** 本書の第 2 部。C++プログラマ向けのプログラミング方法の紹介
- **UDF Syntax Reference:** UDF の文法規約書
- **Python Script Manual:** GOURMET のユーザ向け python スクリプトマニュアル
- **libplatform:** C++プログラマ向けのライブラリ API リファレンス

まず楽しんでください、そして GOURMET コミュニティーに参加しませんか？

第II部

UDFプログラミング

第9章　はじめに

第二部では、C++プログラムの中でどのようにUDFファイルを読み書きするのかを説明しましょう。これは、あなたのシミュレーション・プログラムの入出力ファイルとしてUDFファイルを使用するために必要です。GOURMETはデータ変換やデータ分析のような単純なプログラムを、Pythonスクリプトを使用して行うことができます。しかしながら、Pythonは速くありませんしメモリも多く使います。速度とメモリが重要であるような真剣なプログラムはC/C++やFortranで書かれていなければなりません。

UDFファイルを使用すると、あなたのプログラムをGOURMETと直接接続することができ、GOURMETの上で走る他のシミュレーション・プログラムのように使用することができるのです。GOURMETの標準のユーザ・インターフェースを使用すると、あなたのプログラムの潜在的なユーザのために使いやすいプログラムにすることができます。もし自分のプログラムを持ち、UDFファイルを利用したければ、2つの選択を行うことができます。

(1) オリジナルのプログラムをそのまま残し、あなたのファイルからUDFファイルの形式にデータを変換するC++プログラムを書くか、あるいは、GOURMETに提供されているファイル変換機能を利用することができます。

(2) UDFファイルを直接に読み書きすることができるように、オリジナルのプログラムを書き直すことができます。

UDFファイルの読み書きは容易です。学習する必要のあることはすべて、第10章に要約されています。実際、データベース・ファイル中のデータのように任意の順でUDFファイル中のどんなデータにもアクセスすることができるので、UDFファイルの読み書きがシーケンシャル・ファイルより容易であることがわかるでしょう。

第11章および第12章は、入出力にUDFファイルを使用して、C++プログラムを書き始めたい方のために準備されています。そこでは、UDFファイル中のデータにC++のデータ・オブジェクト(クラスとポインタ)を写像する方法について議論します。

第13章では、pythonスクリプトによってUDFファイルを扱う方法について簡潔に議論します。そこでは、pythonスクリプトを使用してGOURMETから得ることができるサービスについて、完全には記述していません。この主題は、pythonスクリプトによってどのようにUDFファイルを読み書きするかです。pythonスクリプトは、pythonのコマンドラインの中で、あるいはGOURMETのpythonウィンドウの中で実行することができるので、すぐに実行すべき仕事に役立つでしょう。

第10章　UDFプログラミングの基礎

10.1　単純なデータ

10.1.1　UDF ファイルの例

UDF は、定義部とデータ部の二つの部分から構成されます。定義部では、あなたの使いたいデータの名前と他の属性 (タイプ、単位など) を定義します。データ部では、実際のデータを与えます。定義部を\begin{def} から \end{def}の間に記述し、データ部を\begin{data} から \end{data}の間に記述します。

UDF ファイルの簡単な例は、以下のようなものです。

```
// "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}
```

定義部分では、データ名とそのタイプが指定されなくてはいけません。この例では title は文字列タイプ、Nmax　は整数タイプ、そして、average は、倍精度浮動小数点タイプのデータです。最後の x[] は、倍精度浮動小数点タイプの可変長配列データです。

(1) 基礎的なデータ型は、C++言語とほぼ同じです。整数型 (int、short、long)、浮動小数点型 (float、double、single)、そして、文字列型 (string) があります。他にもいくつかのデータ型 (select、ID、KEY) があり、後で説明することにします。(第 12　章参照) ユーザーは、これらのデータを組み合わせて自分のデータ型を定義することができます。(第 11　章参照)

(2) 定義部において ”//” の後に書かれたものはコメントです。コメント以外としての意味はありません。

(3) 配列の大きさはデータ部にあるデータ数により決められます。定義部で指定する必要はありません。

データ部では、データ名と ’: ’ の後にデータが書かれます。数値データ (int、short、long、float、double、single) が、C++の出力と同様に書かれます。(例えば、-2.4, 5.0E-12 のように) しかしながら、C++と違って、文字列データは ” か ’ で囲まれる必要があります。もし文字列がこれらの文字によって囲まれていなければ、それはコメントとみなされ無視されます。

データ名は、定義部分での順序とは無関係にどんな順序でも現れてかまいません。定義部分で定義されたすべてのデータに値を与える必要性もありません。上記の例、 average の値は、プログラムの出力であり、データ部で与えられるのではありません。

配列データは文字 '[' で初め、']' で終わる必要があります。各データ要素は、スペース、コンマ、タブ、改行またはどんな非データ・オブジェクトによって区切られる可能性があります。

これらのルールにより、データ部の記述はかなり自由になります。次のものは、データ部に対する記述の有効な例です。

```
\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}
```

この例では、キーワード title:、Nmax:、x[]: は、データ名でであり省略はできません。また、"Name of the project"、"Generated random numbers"、"first" などのようなテキスト文字列は、コメントであり、取り去ることができます。

10.1.2 C++プログラムで UDF ファイルを読むには

UDF ファイルの読み書きは、C++クラスの UDFManager によって行われます。前節の UDF ファイルを読む例は、以下のようになります。

```
#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) UDF ファイルを処理するために、最初に UDFManager のインスタンスを作ります。ここで、UDFManager uf("file1.udf"); は、UDF ファイル"file1.udf"を処理する uf という UDFManager を作っています。

(2) UDF ファイルのデータを読み込むには、UDF ファイル中でのデータ名と格納先の C++変数の 2 つの引数付きで ”get”メッセージを、UDFManager へ送って下さい。このメッセージのインターフェースは以下の通りです。

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

UDF ファイル中でのデータ名は、文字列によって指定されます。この例では、文 `uf.get("Nmax", Nmax);` は、UDF ファイル中で”Nmax”と呼ばれるデータを読み、C++プログラムの変数 `Nmax` に格納します。同様に、int、double、string などのデータを読むために、第二引数の型で区別される get メソッドを使うことができます。

(3) 配列の大きさは size メソッドで得ることができます。

(4) 配列全体を読み込む時には getArray メソッドを用います。次の例に示すように最初に配列の大きさの領域を確保し、

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

次に getArray 命令で配列に読み込みます。

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

配列が STL （スタンダード・テンプレート・ライブラリ）の vector クラスで確保されている時には、次に示す例のように配列の大きさを確保しないでもっと簡単に読み込むことができます。

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

”getArray”メソッドのインターフェースは、以下のようなものです。

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

もちろん、以下のように、配列データの各要素を読むことができます。

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

UDFManager は、データを UDF ファイルに読み込むためのメソッドを他にも提供しています。上記のプログラムは、以下のように書くことができます。

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

この場合、文字列データと整数データを読むメソッドは、同じにはできないので、メソッドの名前によって区別して使います。メソッド ”intValue”、”doubleValue”、”stringValue”や、”intArray”、”doubleArray”などが利用できます。この方法は使いやすいですが、C++ではデータのコピーが行われているので、もし「`x[]`」巨大な配列の場合はこれらの方法を使うのはメモリーと計算時間の無駄となりますので注意が必要です。

10.1.3 C++プログラムで UDF ファイルを書くには

UDF ファイルへ C++プログラムの出力を書くのは、読むのと同じほど簡単です。下記に、 x[] の平均を計算するプログラムを示します。

```
  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");
```

このプログラムが実行されると、”file1_out.udf”の内容は、次のようになるでしょう。

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

ここにあるとおり、データの書き出しは、”put”メソッドによって行われます。そのインターフェースは、読み込みに使う”get”メソッドが同じであり、最初の引数が UDF ファイルの定義部分で定義されているデータ名、第 2 引数が出力するデータの格納されている C++変数です。

UDFManager はデータベース・プログラムと非常に似たように UDF ファイルを扱います。UDFManager は、get メソッドを使ってどんなデータでも読むことができ、put メソッドを使ってデータの値を変えることができます。データの変更結果は、”write”メソッドが実行された時点で有効になります。`uf.write("file1_out.udf")` という文が、引数によって指定したファイルに定義部分とデータ部を書きます。もしこの引数が無い場合、処理中のファイルへデータを書くことになります。(このケースでは”file1.udf”です)

10.2 構造体型

10.2.1 構造体型の定義

C 言語では、構造体型は、いくつかのデータ・コンポーネントから構成されるデータ型です。UDF は C 言語の構造体型と等価性を持っています。構造体型のデータの定義は簡単です。

```
// "file2.udf"
\begin{def}
  vector:{x:double, y:double, z:double}
```

```
  simulation_condition:{
       temperature: double
       box:{Lx:double, Ly:double, Lz:double}
  }
\end{def}
```

ここで、 vector と simulation_condition は、構造体型のデータです。(今後、構造型データと呼びます) 構造型データは、{ と } の間に書かれたデータ・コンポーネントのリストにより定義されます。C++言語と同じように、構造型データのそれぞれのコンポーネントは vector.x や simulation_condition.box.Lx のように、”.”で参照されます。

10.2.2 Data part of structured data

構造体型のデータは、データ部の { と } の間に書かれます。配列の場合と同じように、構造型データの各要素もスペース、コンマ、タブ、改行、または非データ・オブジェクトによって分けることができます。例えば以下の通りです。:

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

トップレベルのオブジェクトの名前だけが必要であることに注意してください。定義部分で定義されているので、サブ・レベルのオブジェクトの名前は、記述する必要はありません。もちろん、コメントとしてサブ・レベルのオブジェクトの名前を記入することは有害ではありません。下記のものは、有効なデータ部の例です。

```
\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 C++で構造型データを読むには

構造型データを読むことは、(データの完全な名前で読み込むのを気にしなければ) 単純です。例えば、上記のファイル (”file2.udf”) のそれぞれのデータ項目を以下のようにして得ることができます。

```
  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);
```

上記のプログラムは、”現在の UDF パス”を表す文字列変数を導入することによって、より短くすることができます。

```
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);
```

この方法は理解しやすいですが、それぞれのデータ項目を読むために、コストのかかる文字列操作を必要とします。これを避けるために、Location クラスが導入されました。このクラスの使い方は、以下の例の通りです。

```
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);
```

このプログラムでは、UDFManager クラスと Location クラスの以下のメソッドを使っています。

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

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

メソッド Location::sub(const string& component) が引数によって指定したコンポーネント・データのロケーションを返します。(図 10.1 参照) メソッド+Location::seek(const&string)+ は、現在位置を文字列で指定したロケーションに移動させます。

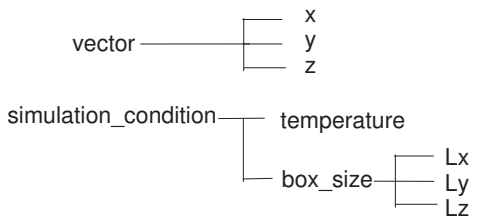

図 10.1: UDF の構造型データ階層

10.2.4 C++で構造型データを書くには

UDF ファイルへ構造型データを書くのは、読み込みに似た方法で行うことができます。以下の例を見てください。

```
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")
```

これまでのところは、構造型データは、それぞれのデータ・コンポーネントについて処理する必要がありました。しかし、構造型データをひとつの構成要素として処理する方がより良い方法です。第 11 章では、構造型データをひとつの構成要素として読み書きし、C++のオブジェクトに UDF ファイルの構造型データをマップする方法について議論することにしましょう。

10.3 単位

定義部分では、各データ項目の単位も記述することができます。以下に、例を示しています。

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

単位は、浮動小数点型 (float、double、single) ならばどれでも、その後に記述できます。基本単位 (例えば SI や cgs 単位系で使われるもの) は、実装されています。後で議論しますが、ユーザーは自分独特のものを、定義することができます。

単位は、UDF ファイルを読みやすくするために導入されました。データ部が変更されるのではないことに注意して下さい。上記の UDF ファイルは、 pendulum_length が 50.0[cm] であり、atomic_radius が 0.2[nm] だということです。しかし、データが C++プログラムに渡されるときには、これらの単位は無視されます。つまり、 uf.doubleValue("pendulum_length") は、50.0 を返し、 uf.doubleValue("atomic_raidus") は、0.2 を返します。このように物理量の単位は、単に GOURMET によって使われるのであり、シミュレーション・プログラムには影響しません。

GOURMET は、データを他の単位で表示するサービスを提供します。しかし常に、そのデータは定義部分で記述された単位での値が UDF ファイルに記録されます。その例を示しましょう。上記の UDF ファイルを GOURMET で読み込むと、 pendulum_length は、50.0[cm] と表示されます。ユーザーは、その単位をデフォルトの [cm] から [m] へ変更することができます。そうすると、 pendulum_length は、0.5[m] と表示されることになります。 ここで彼が、 pendulum_length を 3[m] に変更したとすると、UDF ファイルのデータ部には pendulum_length:300 （300[cm]）として格納されます。また、（更新された UDF に対しては） uf.doubleValue("pendulum_length") も 300.0 を返します。

新しい単位を、\begin{unit}と\end{unit}の間に定義することができます。以下の例を見てください。

```
\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}
```

この例で、 pendulum_length は、 0.5[L]=1.5[m] であり、 kinetic_energy は
0.1[L]=0.1*5*3*3/(2*2)[kg m^2/s^2] = 1.125[J] となります。新しい単位は、既に定義された単位を使って定義することができます。この定義では、ユーザーは pendulum_length の値を、デフォルトの単位 ([L])、やその他の標準の単位（例えば [m] あるいは、[cm]）で見ることができます。また、kinetic_energy の値を、デフォルトの単位 ([E]) や、[J] または [erg] で見ることができます。

さらに以下の構文を使うと、単位変換に必要となる係数をユーザーに変更させることも可能です。

```
\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}
```

ここでは、[L]、[T]、[M] が reference で指定された量を参照して、定義されています。デフォルトでは、energy は、[E] の単位で表示されます。しかし、もしユーザーが reference の値を入力すれば、[J] または [erg] で energy を見ることができます。

これは、科学シミュレーションで単位を指定するための典型的なスタイルです。科学シミュレーションは、無次元形式で通常行われます。プログラマーはリファレンスとなる量を定義し、そして、これらのリファレンス量を用いてすべての物理量を無次元化します。上記の定義は、プログラマーには従来の形式をそのまま使うことを可能にし，一方でエンド・ユーザーには、SI あるいは cgs 単位で物理量を見ることをできるようにします。

単位は、データ・ファイルの読みやすさと利用しやすさを改良するために、導入されるものです。全ての物理量を、適当なリファレンス量を使った問題のある無次元化量で表現するようなシミュレーション・コードを書くことに、多くの人々は慣れきっています。リファレンス量が明白に宣言されなければ、他の目的のためにファイルの情報を利用することができません。すべての物理量のための単位を記述することを、強く推奨します。それは、ファイルの読みやすさを改善し、さらにそのファイルが他のアプリケーションによって使われる可能性を増やすことができます。

10.4 レコード

UDFManager は UDF で書かれたテキストデータを全て読み込んで、ユーザに対するサービスを開始します。しかし、シミュレーションプログラムの出力が非常に大きな場合には、すべてのデータを読み込んでメモリ上に展開することは不可能です。例えば分子動力学では、適当なタイミングですべての原子の位置、速度, 力などを出力しますが、出力する回数が何万回となると、すべてのデータをメモリに読むことは不可能です。通常、出力するタイムステップごとのデータをすべてメモリ上に展開すれば十分です。有限要素法のプログラムにおいても、各瞬間の系の速度場、変位場などのデータがメモリ上に展開されれば十分でしょう。UDFManager がメモリー上に展開して扱うことのできるデータの記録単位をレコードといいます。大規模シミュレーションの場合、レコードは、ある瞬間の系の状態の記録といって良いでしょう。

UDFManager は、レコードを処理するために、以下のメソッドを提供しています。

1. string UDFManager::newRecord(string recordName="")
   現在のレコードが閉じられます。新しいレコードが生成されて、レコード・ラベルが返されます。

2. unsigned int UDFManager:: totalRecord ()
   全レコード数が返されます。

3. int UDFManager:: currentRecord ()
   現在のレコード番号を返します。

4. const string& UDFManager:: getRecLabel ()
   現在のレコードにつ付けられているラベルが返されます。

5. void UDFManager:: setRecLabel (const string& recordLabel)
   現在のレコードのラベルが変更されます。

6. bool UDFManager::jump(const string& recordLabel)
   現在のレコードから、指定されたラベル recordLabel とマッチした最初のレコードに移動します。

7. bool UDFManager:: jump (int timeStep)
   現在のレコードから、指定されたレコード番号 timeStep　のレコードに移動します。

8. bool UDFManager::nextRecord();
   現在のレコードの次のレコードに移動します。

レコードには 2 種類あります。シミュレーションの条件や物理定数などを記述した部分（これはすべての時間に共通です）と、系の時間発展を記述した部分（これは毎回変わります）です。前者をグローバルレコード、後者を時系列レコードと呼びます。

```
\begin{unit}
    \textbf{Unit definition}
\end{unit}
\begin{def}
        UDF クラス定義
        UDF データ定義
\end{def}
\begin{global_def}
        グローバルデータ定義
\end{global_def}
```

グローバルレコードの初期状態の記述

```
\begin{record}["step1"]
第 1 ステップのデータの記述
\end{record}
\begin{record}["step2"]
第 2 ステップのデータの記述
\end{record}
```

(1) グローバル・レコードは\begin{global_def} と \end{global_def}の間に記述されます。

(2) 時系列レコードは、\begin{record} と \end{record}の間に記述されます。

(3) グローバル・レコードは、時系列レコードの前に、一度だけ記述されます。

(4) 時系列レコードは何回書いても構いません。各レコードに書いてある変数が毎回変わっていても構いません。

(5) UDFManager はグローバル・レコードと 1 回分の時系列レコードの両方をメモリ上に展開して管理します。

(6) \begin{record} の後の［］の間に囲まれた部分はレコードにつけられたラベルです。あるレコードに飛ぶ時にはこのラベルを利用することができます。

最後に、完全な UDF ファイルの例として、野球シミュレーターの UDF ファイル内容を、表 10.1 に示します。

表 10.1: 野球シミュレータの UDF 定義とデータ例

```
\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","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"

\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}
```

第11章　クラス

11.1　はじめに

現代のプログラミング語、C、C++、Pascal 、Fortran ９０などでは、一群のデータをひとつのオブジェクトとみなす構造型データをサポートしています。前のセクションで議論したように、UDF も同様に構造型データをサポートしています。しかしながら、前の章では、UDF ファイルの読み書きは、構造型データの各データ・コンポーネント毎に行いました。望ましいのは、構造型データをひとつの構成要素のように扱って、UDF ファイルの中の構造型データの間での直接の関連付けを行うようなデータ・オブジェクトをプログラミングすることです。このセクションでは、ひとつの例として C++でマッピングを行う方法を議論してみましょう。

11.2　UDF クラス

UDF クラスは、構造型データの型であり、定義部分で使うことができます。以下の例のように使われます。

```
// 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}
```

最初の行、

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

は、新しい UDF クラスである Vector3d の定義です。Vector3d が、倍精度型の x、y、z の３つのコンポーネントを持っており、そして [unit] で指定された単位を持っていることを定義しています。
単位の定義構文は、次の行にあります。

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

この定義は、次の内容と同じ意味です。（実際にこのように定義することもできます）

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

でも、UDF クラスを利用した定義の方が、よりシンプルで理解しやすいはずです。3行目と4行目は、UDF クラスのより高度な使い方の例となっています。これらのデータ構造が、図 11.2 で示されものであることはすぐにわかりますね。

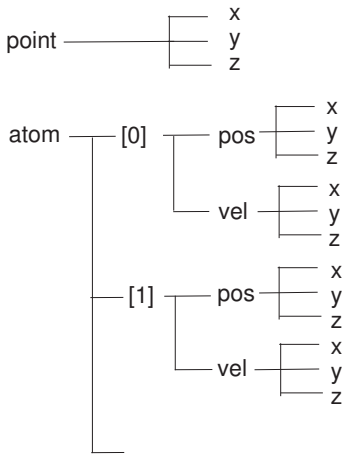

図 11.1: "file3.udf"のデータ構造

11.2.1 構造型データを C++クラスオブジェクトに読み込むには

以下では、C++でクラスへのこのような UDF クラスをマップする方法を議論します。最初に、以下のような C++のクラス、Vector3d を定義します。

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

2つのメソッド "get"と"put"が、UDF ファイル内の構造型データ `Vector3d` を読み書きするために加えられています。これらのメソッドの実装は、以下のようになります。

```
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)
}
```

同様に、以下のように　Atom クラスを定義することができます。

```
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)
}
```

Vector3d クラスと Atom クラスがこのように定義されていれば、UDF ファイル”file3.udf”を読み書きする以下のプログラムを書くことができます。

```
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);
```

第12章　ポインタの表現方法

12.1　はじめに

ここでは、C/C++でポインターを UDF ファイルにマップする方法を、議論しましょう。例として、空間に多くの球体を描くことを考えて下さい。我々は、それぞれの球体が重心の位置と球体タイプによって特性を示されると仮定します。球体タイプは、球体のサイズと色、そして”big_blue”、”small_green”のような名前を表しているデータ・セットです。我々のプログラムのクラス定義は、以下のようになるでしょう。

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

ここに sphere[i] は、球体タイプを指すポインター pType を持っています。

このようなデータをテキスト・ファイルへ書くためには、ポインターを整数または記号列といった読みやすいものにしたいでしょう。一つの方法はインデックスを使うことです。もし sphere[i].pType が &sphereType[p] と等しければ、球体タイプを指定するのに、インデックス p を使うことができます。我々の UDF ファイルの定義部分は、次のようになるでしょう。

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

データ部の例は、次のようなものです。

```
\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}
```

このデータは、 sphere[0] が small_green 、 sphere[1] が big_blue 、 sphere[2] が medium_red であることを示します。

このマッピングはシンプルで分かりやすいですが、この定義では、Sphere クラスの type_code が、C++ ポインターの置換えである (すなわち、 type_code が他のデータ・オブジェクトを指し示す) ということが明白ではありません。他のユーザーに論理的な構造を説明するために、コメントを付け加えることができますが、コンピュータに対してそのようなことはできません。

ファイルの中のこのような論理的な構造を記述するために、UDF にはポインターを表現する 2 つのデータ型、KEY とＩＤがあります。キーは、ポインターを文字列に置き換え、そして、ID はポインターを整数へ置き換えます。これらは、クラス定義にのみ記述され、そしてそのクラスのデータ・オブジェクトに対する唯一の名前として使われます。それぞれについて、以下のセクションでもうすこし説明しましょう。

12.2　KEY 型

キーは、そのクラスのデータ・オブジェクトの名前として使われる文字列です。上記の UDF ファイルは、キーを使うと以下のように書きなおすことができます。

```
//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}
```

ここで SphereType の　name が KEY です。KEY の値はデータ・オブジェクトを特定するためにクラス内でユニークでなくてはいけません。

逆に、Sphere の type は、<SphereType,KEY>として定義されます。それは、UDF クラス SphereType のデータ・オブジェクトを特定するための KEY 値が type であることを意味します。

12.3 ID 型

ID は、整数であること以外は、キーと同じです。クラスのいかなる 2 つのオブジェクトも、同じ ID を持っているべきではありません ID を使って、上記のデータ構造を書きなおすと以下のようになります。

```
//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}
```

次の Python スクリプトを実行してみると、ID がどのように働くかがわかるでしょう。

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

12.4 C++で読み込むには

下記の C++プログラムは”file 2.udf”を読むものです。

```
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;
     }
}
```

この例では、キーをポインターと関係づけるために、STL （スタンダード・テンプレート・ライブラリ）のデータ型 map が、使われています。プログラムは、すべての SphereType データが、 Sphere のデータの前に読まれることを前提としています。

もしキーのかわりに ID を使うなら、sphereTypeTable は、 map<int, SphereType*> として定義することになります。

12.5　ID 型と KEY 型に関連したメソッド

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

   class_name で指定されたクラスのオブジェクトに対して使用されているすべての ID 値が、vector として返されます。

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

class_name で指定されたクラスのオブジェクトで、id と同じ ID 値を持つデータのロケーションが返されます。

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

   class_name で指定されたクラスのオブジェクトに対して使用されているすべての KEY 値が、vector として返されます。

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

   class_name で指定されたクラスのオブジェクトで、key と同じ KEY 値を持つデータのロケーションが返されます。

第13章　PythonでUDFファイルを扱うには

13.1　$プレフィックス

Python スクリプトを使って、UDF ファイルを処理することができます。GOURMET で UDF ファイルを開けば、$オプションを利用してどんな UDF ファイルのデータでも処理することができます。例として、次の内容の UDF ファイル（Python_test.udf）を GOURMET で開いて見ましょう。（UDF ファイルを、テキストエディタで作成できることはご存知ですね）

```
// "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}
```

さて、Python ウィンドウで以下の Python スクリプトをタイプして Run ボタンを押して下さい。

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

出力は以下のようになるはずです。

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

このようにして、GOURMET のファイル・メニューによって開いた UDF のどんなデータも、接頭辞$でデータ名を指定すればアクセスすることができます。（これには接頭辞$が Python 構文解析プログラムにとって有効なシンボルでないことを使って、Python を拡張しています。）

上記の Python スクリプトは、Python スクリプト拡張機能を使っています。そして、これは GOURMET でのみ利用できます。

もし Python のコマンドラインで同じ事をしたければ、下記のようにする必要があります:

```
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")
```

ここで、”get” は、引数によって指定された値を返すメソッドです。そして”put”は、最初の引数の値を、第二引数によって指定されたデータに、設定するメソッドです。

もちろん、GOURMET の Python スクリプト・ウィンドウでも上記のプログラムを実行することができます。様々なファイルに存在するデータを操作するために、これらのメソッドを使うことができます。例えば、”fileA.udf”と”fileB.udf”のデータを結合して、”fileC.udf”に出力することを考えましょう。ここで、それぞれの UDF は以下の定義部を持つものとします。

```
// 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}
```

これを、以下の Python スクリプトによって実現することができます。

```
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])
```

ここで、get(”A[]”,[i]) は、i が 0 のとき get(”A[0]”) と同じことを表します。

UDFManager の他のメソッドは、付録 B　に要約されています。

付 録A　GOURMETに関するドキュメント

より詳細に関しては、以下のドキュメントを参照してください。

- **GOURMET Operations Manual:** GOURMET の操作に関するユーザーマニュアルです。
- **UDF Syntax Reference:** UDF 構文のリファレンスです。
- **Python Script Manual:** GOURMET で使う Python スクリプトに関するマニュアルです。
- **libplatform:** C++プログラマーのためのプラットホーム・インターフェース・ライブラリのリファレンスです。

付 録B　GOURMETなPythonの最短リファレンス

このセクションは、GOURMET における Python の使い方と Python スクリプトの書き方を簡単にまとめたものです。Python についてもっと詳しいことを知りたければ、最後の章の図書目録に挙げてある書籍を読んで下さい。

B.1　変数

Python が処理するデータは、数字 (整数、LONG 型整数、浮動小数点と複素数)、テキスト文字列、配列オブジェクトとその他のクラスのオブジェクトです。

例えば以下のように、使います。

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

変数のタイプ宣言が必要でないことに注目してください。文字列はシングルクオート (')、ダブルクオート (")、三重のシングルクオート (''') または三重のダブルクオート (""") によって囲みます。文字列データのための演算子+は、文字列の接続を意味します。数値に対して、通例の算術演算 (+、-、*、/)、べき乗演算 (**) とモジュロ演算子 (%) を使うことができます。

B.2　タプル、リスト、辞書

Python は、データの集合を表すために、タプル、リストと辞書のデータ型を提供します。これらは、以下のように使います。

```
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']
```

タプル、リスト、辞書はそれぞれ、"("、"

"、"{"で始まり、")"、"}"、"

"で終わります。

それぞれの要素は C 言語と同様に参照できます。(`tuple_example[0]`, `list_example[1]` など) また個々の要素は、上記の例　`list_example` のように,、違う型でもかまいません。 `list_example[0]` は整数の 1、

list_example[1] は浮動小数点数の 2.3、 list_example[2] は ['awk',' perl' , 'ruby'] というリストです。

辞書の要素はキーで参照でされます。例えば、 dict_example['awk'] は 1 であり、 dict_example['perl'] は [2,3] です。

これらは、最後のステートメントの出力で確かめることができます。

タプルは不変のオブジェクトです。タプルの要素の値や要素の数を変えることはできません。

一方で、リストの内容は変更できます。例えば、以下の通りです。

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

B.3 制御文

Python では、通例の制御構文を使えます。"if/else" は条件付き分岐ステートメント、"for" と "while" がループのためのステートメントです。

条件付き分岐の使用例は、下記の通りです。

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

ループの使用例は、下記の通りです。

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

"if/else" ブロックと "while" ブロックが字下げによって示されていることに注意してください。Python は、字下げにはとても厳密です。同し階層のブロックは、同じ数のスペースまたはタブ文字で字下げされなくてはいけません。

Python の "for" ステートメントは、C 言語の文と少し異なります。

以下の Python スクリプトで、これを説明します。

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

このスクリプトの出力は、リストの各要素のプリントアウトであるがお分かりですね。

"for" ステートメントは、"for" ブロック内の制御変数を、リストの要素に置き替えて繰り返し実行します。前の "while" ステートメントの例を "for" ステートメントで書きなおすと、次のようになります。

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

このプログラムは、以下のようにも書くことができます。

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

関数 range(N) は [0,1, ...N-1] に相当します。より一般的には、range(N,M) が [N,N+1,...M-1] に相当し、range(N, M, k) は、N から (M- 1) まで増分 k で増える数値のリストに相当します。

B.4 関数とクラス

関数は、def ステートメントによって定義することができます。

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

Python は、C++に類似した「クラス」の実装を持っています。下記のものは、簡単なクラスの定義と使用方法を示しています。

```
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()
```

ここでは、４つのメソッドが、このクラスのために定義されています。 __init__ は Vector3d オブジェクトのコンストラクター、 norm は各要素の二乗和を返し、 multiply は現在のベクトルを c 倍した新しいベクトルを返し、 list はベクトルの要素をリストにして返します。

すべてのメソッドの最初の引数 self は、オブジェクト自身を表します。(C++言語での *this と同じ)。

B.5 モジュール

プログラム・モジュール (関数とクラス) は、import ステートメントによってロードされます。例えば、数学関数、sin、cos、log、exp などは、「math」と呼ばれるモジュールで定義されていますが、これら使うためには、下記のようにします。

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

この場合、math 関数に前の接頭辞 `math` は必要です。もし math 接頭辞を使うのが嫌なら、下記のようにも書くことができます。

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

もちろん、あなた自身のモジュールを作ることができます。上記の関数 `fact(n)` をファイル "factpy" に保存すれば、あなたは、以下のようにしてこれを使うことができます。

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

B.6 GOURMET の中で使う UDFManager

あなたは、UDFManager クラスを使って、UDF ファイルのデータを処理することができます。
以下のような UDF ファイルを処理したいとしましょう。

```
// "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}
```

あなたは、次のような Python スクリプトを使って、このファイルを読んだり、書いたりすることができます。

```
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()
```

ここで、

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

では、”file.udf”を処理する UDFManager を作っています。

creates a UDFManager which handles 'file.udf'. `get(location)` メソッドが指定したロケーションにあるデータを返します。

このロケーションは文字列形式で明確に指定することができます。例えば、以下のように指定できます。

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

同じ内容を、次のように数値のリストを使って、記述することもできます。

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

B.6.1 UDFManager メソッドのサマリー

UDFManager クラスのメソッドの一覧を以下に示します。

パラメーター **location** は Location クラスのインスタンスです。Location クラスは、簡単な文字列操作で UDF 変数名を扱えるようにするための補助のクラスです。

- UDFManager(udffilename): udffilename で指定したファイルが開かれ、UDFManager オブジェクトが作られます。
- get(location,unit=””): 指定された location の値を返します。
- getArray(location,unit=””): 指定された location の値の配列を返します。
- put(value,location,unit=””): value が、location にセットされます。
- write([filename,record,mode]): データが、実際にファイルに書かれます。
- totalRecord(): UDF ファイルの総レコード数が返されます。
- currentRecord(): 現在のレコード位置が返されます。
- jump(record_no_or_label): 現在のレコードが、整数番号またはラベル文字列で指定されたレコード位置へ移動されます。
- nextRecord(): 現在のレコードが次のレコードへ移動されます。
- newRecord(label=””): 指定されたラベル文字列を持った新しいレコードが作られ、最後のレコードへアペンドされます。
- seek(location): 現在のデータの位置が指定されたロケーションへ移動します。
- tell(): 現在のデータのロケーションが返されます。
- next(): 現在の配列のロケーションに含まれる最も右のインデックスが 1 増やされます。
- prev(): 現在の配列のロケーションに含まれる最も右のインデックスが 1 減らされます。
- type(location): 指定された location の UDF データ型が、文字列として返されます。
- size(location): 指定された location が配列ならば、要素の数が返されます。
- getIDList(class_name): class_name で指定されたクラスのオブジェクトに対して使用されているすべての ID 値が、リストとして返されます。

- getKeyList(class_name): class_name で指定されたクラスのオブジェクトに対して使用されているすべての KEY 値が、リストとして返されます。
- getLocation(class_name,key_or_index): class_name で指定されたクラスのオブジェクトで、key_or_index と同じ KEY/ID 値を持つデータのロケーションが返されます。

B.6.2 描画メソッドのサマリー

UDFManager 拡張モジュールが GOURMET で使われているので、GOURMET のビュー・スクリーンで 3 次元のオブジェクトを描くことができます。

ここで、パラメータは以下の意味を表しています。
coordinateN(N is integer) は、[x、y、z] といった 3 次元の座標の値をもつリストであるか、、あるいは同様な座標データを表す $ の接頭辞を持つロケーション文字列。
coordinate_list は [[x1, y1, z1], [x2,y2,z2]...] といった 3 次元の座標の値をもつリスト。
attribute_id は描画属性を指定する整数値またはリスト。

下記に、図画関数の一覧を示します。

- 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,ax,ay,az,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,ax,ay,az,cx,cy,cz])
- ellipsoid2(coordinate1,coordinate2,[r,g,b,t,a])
- tetra(coordinate1,coordinate2,coordinate3,coordinate4,[r,g,b,t])
- cube(coordinate1,length,[r,g,b,t])
- cone(coordinate1,coordinate2,[r,g,b,a,radius])
- arrow(coordinate1,coordinate2,[r,g,b,t,radius,height])
- text(coordinate,contents,[r,g,b,t,size])
- clearDraw() すべての描画オブジェクトが消去されます。
- appendDraw() 1 つの Python 実行タスク内で、この呼び出し以後の描画オブジェクトがオーバーライトされます。

B.7 参考文献

(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/