GOURMET TUTORIAL

Originally written by 土井正男 (第 1 から第 3 章) 平成 17 年
Modified by Takashi Taniguchi (第 1, 2 章) 平成 18 年
Modified by 滝本　淳一 (第 3 章) 平成 18 年

2006 年 10 月 13 日

目次

図目次

第 1 章

Gourmet

1.1 Gourmet とは

こ Gourmet は経済産業省の Octa プロジェクト (正式名称　高機能材料設計プラットフォームの開発プロジェクト）の中で開発されたプログラムである。このプロジェクトでは、材料設計に役立つ様々なシミュレーションプログラムが開発され、それらの共通基盤として Gourmet が開発された。Gourmet の開発目的は二つあった。

- 各種のシミュレーションプログラムに共通のインタフェースを与える。
- シミュレーションプログラムの間のデータ変換を容易にし、プログラムを連携を可能にする。

Gourmet は、計算科学で使われているシミュレーションプログラムに汎用のインタフェースを提供する。Gourmet はプログラムが必要とする入力データの一覧を示し、プログラムが出力するデータの一覧を示す。さらに、入力データの編集、結果の表示、グラフ化、アニメーション作成などについて多くの便利な機能を提供する。

Gourmet はプログラムの達人のためだけのものではない。プログラムを一度も書いたことのない学生であっても、Gourmet の上で、簡単なプログラムを作ることができる。Gourmet が装備している Python という簡易言語を用いるなら、簡単にプログラムの実習ができる。学生の演習や実験のツールとしても役に立つ。

ここでは、Gourmet の機能をできるだけ短時間で説明したいと思う。

Gourmet の解説として、次の本を推薦する。

- 物理仮想実験室　土井正男、滝本淳一編著　名大出版会　（2004）

この本には Gourmet を使って、大学で学ぶ物理のシミュレーションの例が多く載っている。また、この本に付属した CD を用いると、Gourmet が簡単にインストールできる。

Gourmet は Python という言語を用いてプログラムすることができる。Python は Basic,Perl の流れを汲む簡易言語である。歴史は新しいが使いやすさ、高速性の点から急速に普及している。Python は下記の URL からダウンロードできる。

- http://www.python.org/
- http://www.python.jp/

純粋に Python だけを使いたい人はここからダウンロードしたものを使ってよい。物理仮想実験室の 1-3 章と付録には Python についての簡単な解説がある。Python について詳しく学びたい人は以下の解説書を参考にされたい。

- Mark Lutz, David Asher 著、紀太章訳、「初めての Python」、オライリー・ジャパン (2000)
- Mark Lutz 著、飯坂剛一監訳、「Python 入門」、および「Python プログラミング」、オライリー・ジャパン (1998)

1.2 Gourmet から Python を使ってみる

Gourmet を起動してすると、以下のような Editor ウインドウが現れる。

図に示しているプログラム領域[注1]に, つぎのようなプログラム[注2]

```
%-------------------------------------------
# ファイル名 [1.2.py] (←この部分は書かなくてもよい)
print  'Welcome to Gourmet! '
print 1.0/3
%-------------------------------------------
```

[注1] このプログラム領域を Python Window (パイソン窓) と呼ぶこともある。

[注2] %-----と%-----で囲まれた領域がプログラム領域に書いてほしい部分である。以後も同様。
Python プログラムでは、ある行において#が存在する場合、その位置以降はコメント行となる。

を入力し、Run ボタンを押す。すると下の窓 (図中の”Program の実行結果が書き出される部分”と書いてある位置) に、次のような実行結果があらわれる。

```
%--------------------------------------------
Welcome to Gourmet!
0.333333333333
%--------------------------------------------
```

もう一度 Run ボタンを押せば、同じ計算が再度行われる。このときログ領域にある前回の結果の後に新しい計算結果が追加される。過去の計算結果を消去したい場合は、”Clear Log” ボタンを押す。書いたプログラムを保存したい場合には、図中の”Program 領域”のすぐ上にある”Save”ボタンを押すと、(ディレクトリとファイ名[注3]を指定して）保存することが出来る。後でもう一度、そのプログラムを実行したい場合には、”Save”ボタンの左横の”Load”ボタンをクリックし、ファイルを指定して読み込むことが出来る。

1.3 Python のデータ型

1.3.1 単純な型

Python プログラムの変数は文字列、数値などいろいろな型を持っている。文字列は、シングルクォート (’) またはダブルクォート (”) で挟む。数値には、通常の四則演算 (+,-,*,/) や、累乗 (**)、余り (%) 等の演算子が使える。文字列には、+（連結）等の演算子が使える。
以下のテキストを Python Window の上の画面にコピーペーストして Run ボタンを押す。

```
%--------------------------------------------
# ファイル名 [1.3.1.py]（←この部分は書かなくてもよい）
msg="Hellow "+ "world"
print msg
a = 10
b = 2
c = a+b
print "a=",a,";  b=",b,";  a+b=", c
%--------------------------------------------
```

次のような結果が下段の PythonWindow に現れる。

```
------------出力-----------------------------
Hellow world
a= 10 ;  b= 2 ;  a+b= 12
--------------------------------------------
```

[注3] ファイルの拡張子としては .py が使われる

二つの Python 文を同じ行に書くときには ";" を用いる。

```
%-----------------------------------------
a=10.0; b=20.0; print a+b, a-b, a*b
%-----------------------------------------

------------出力-----------------------------
30.0 -10.0 200.0
-----------------------------------------
```

1.3.2 リスト

[ ] の間にコンマで区切って要素を並べたものをリストという。

```
%-----------------------------------------
# ファイル名 [1.3.2a.py]（←この部分は書かなくてもよい）
q=[1,2,3]
print q
%-----------------------------------------

------------出力-----------------------------
[1, 2, 3]
-----------------------------------------
```

リストの各要素を参照するには添え字を使う。添え字は 0 から始まる。つまり q[0] の値が 1 であり、q[1] の値が 2 である。リストの大きさは len(q) で知ることができる。**リストの最後の要素は添え字 [-1]** をつけることによって呼ぶことができる。[注4]

```
%-----------------------------------------
# ファイル名 [1.3.2b.py]（←この部分は書かなくてもよい）
q=[1,2,3]
print q[0],q[1],q[2],q[-1]
%-----------------------------------------

------------出力-----------------------------
1 2 3 3
-----------------------------------------
```

[注4] q[4] のように配列要素を定義していない部分にアクセスするとどうなるか試してみよう。例えば, `print q[4]`

リストは Fortran や C における配列に近いものであるが、配列に比べてはるかに柔軟な操作ができる。

(1) 要素の追加：リストの大きさは可変で, 次に示す例のように append メソッドや”+”演算子により新しい要素を追加することができる。

```
%-------------------------------------------
# ファイル名 [1.3.2c.py]（←この部分は書かなくてもよい）
q=[1,2,3]
q.append(4)
print len(q), q
r=q+[5]
print len(r), r
s=q+[5,6]
print len(s), s
%-------------------------------------------

------------出力----------------------------
4 [1, 2, 3, 4]
5 [1, 2, 3, 4, 5]
6 [1, 2, 3, 4, 5, 6]
-------------------------------------------
```

(2) 部分リスト：下の例のように、 [n:m] **という書き方を用いて、リストの q[n] から q[m-1] までの要素**からなる部分リストを取り出すことができる。

```
%-------------------------------------------
# ファイル名 [1.3.2d.py]（←この部分は書かなくてもよい）
q=[1,2,3,4,5]
print q[0:3]
print q[2:-1]
print q[-1], q[4]
%-------------------------------------------

------------出力----------------------------
[1, 2, 3]
[3, 4]
5, 5
-------------------------------------------
```

最後の文の実行結果から、q[-1] が q[4] と同じ要素を指すことがよくわかる。

(3) リスト要素は同じ型である必要はない。

```
%-------------------------------------------
# ファイル名 [1.3.2e.py] (←この部分は書かなくてもよい)
q=["apple", 100, "orange", 80]
print q
n= 0
while ( n < len(q)): print q[n]; n=n+1
%-------------------------------------------
```

(註：while 文の文法は、1.4.3 節で詳しく説明する。)

```
------------出力----------------------------
['apple', 100, 'orange', 80]
apple
100
orange
80
-------------------------------------------
```

リストの要素はリストであってもよいから、次のような書き方もできる。

```
%-------------------------------------------
# ファイル名 [1.3.2f.py] (←この部分は書かなくてもよい)
q=[["apple", 100], ["orange", 80]]
n= 0
while ( n < len(q)): print q[n][0],q[n][1]; n=n+1
%-------------------------------------------
------------出力----------------------------
apple 100
orange 80
-------------------------------------------
```

このことにより多次元の配列を扱うことができる。

```
%-------------------------------------------
# ファイル名 [1.3.2g.py] (←この部分は書かなくてもよい)
table = [[1, -1],[2,-2] ]
print table[0][0],table[0][1]
print table[0]
print table
%-------------------------------------------
------------出力----------------------------
1 -1
[1, -1]
[[1, -1], [2, -2]]
-------------------------------------------
```

大きな多次元配列を扱うには後で述べる array という型を用いるほうが便利である。

1.3.3 配列への代入

配列への代入は各要素について行われる。

```
%-------------------------------------------
# ファイル名 [1.3.3.py] (←この部分は書かなくてもよい)
[a,b,p]=[-1, -2, [1,2] ]
print a, b
print p
%-------------------------------------------
------------出力-----------------------------
-1 -2
[1, 2]
-------------------------------------------
```

1.4 制御構文

1.4.1 for

一連の命令を繰り返し実行するには、for 文や while 文を用いる。Python の for 文は C や Basic の for 文に似ているが、書き方が違う。次の例を見ていただきたい。

```
%-------------------------------------------
# ファイル名 [1.4.1a.py] (←この部分は書かなくてもよい)
list=["apple", 100, "orange",80]
for item in list:
     print item
%-------------------------------------------
------------出力-----------------------------
apple
100
orange
80
-------------------------------------------
```

この例のように for 文の文法は以下のようになっている

```
for item in list:
     実行されるべき命令
```

for 文は、list の最初の要素を item に代入し、命令を実行し、次に list の 2 番目の要素を item に代入し命令を実行する。これを繰り返し、リストのすべての要素について命令を実行して終わる。**for 文で実行される一連の命令は for 文に続いて字下げして書かなくてはいけない。**Python は命令文のブロックを示すのに字下げを用いている。字下げは、タブまたは半角スペースによって行う。半角スペースを用いる

ときには正確に同じ数のスペースを入れないといけない。これはわずらわしいのでタブを使って字下げをすることを薦める。次の例は for 文を使った少し複雑な例である。

```
%-------------------------------------------
# ファイル名 [1.4.1b.py] (←この部分は書かなくてもよい)
list=[["apple", 100], ["orange", 80]]
for item in list:
     print item[0],":",item[1] "yen"
%-------------------------------------------
------------出力-----------------------------
apple : 100 yen
orange : 80 yen
-------------------------------------------
```

1.4.2 range 関数

C や Fortran では、ある int 型変数の値を変えながら、一連の命令を実行するというプログラムが多い。このようなプログラムの書き方を行うために range 関数が用意されている。

range(n) は、[0,1,...n-1] という配列と等価である。

range(n,m) は、[n,n+1,..m-1] という配列と等価である。

range(n,m,d) は、[n, n-d, n-2d,.. m'] という配列と等価である。

```
%-------------------------------------------
# ファイル名 [1.4.2a.py] (←この部分は書かなくてもよい)
a=range(5)
b=range(2,5)
c=range(5,2,-1)
print a, "\n", b, "\n", c     # "\n"は改行コード
%-------------------------------------------
------------出力-----------------------------
[0, 1, 2, 3, 4]
[2, 3, 4]
[5, 4, 3]
-------------------------------------------
```

range 関数を使って書いたプログラムの例を示す (計算結果は, $s = 1$)。

```
%-------------------------------------------
# ファイル名 [1.4.2b.py] (←この部分は書かなくてもよい)
a=[1,-1,0]
b=[2, 1,4]
s=0
for i in range(len(a)):
    s=s+a[i]*b[i]
print s
%-------------------------------------------
```

1.4.3 while

Python の while 文は C や Basic の while 文と同様, 条件が満たされている限り実行する命令である。for 文と同様に実行する命令のブロックは字下げで指定する。上に書いた for 文の例はしたのように書くこともできる。

```
%-------------------------------------------
# ファイル名 [1.4.3.py] (←この部分は書かなくてもよい)
a=[1,-1,0]
b=[2, 1,4]
s=0;i=0
while i < len(a):
    s=s+a[i]*b[i]
    i=i+1
print s
%-------------------------------------------
```

1.4.4 if,elif,else

if,else, elif (else if のこと) の使い方は C と同じである。

```
%-------------------------------------------
# ファイル名 [1.4.4.py] (←この部分は書かなくてもよい)
for i in range(10):
    if (i <3): print i, "small"
    elif ( i < 7): print i, "medium"
    else:      print i, "large"
%-------------------------------------------
```

Exercise 1:

(1) 0 から N までの整数の和を計算する Python プログラムを書け。(ファイル名 Ex1.py)

1.5 Python の関数

sin, cos, exp などの数学関数を使うときには import math を実行する。sin(x) ではなく、math.sin(x) と書く。

```
%-------------------------------------------
# ファイル名 [1.5a.py] (←この部分は書かなくてもよい)
import math
for i in range(5):
    x=i*0.1
    print x, math.sin(x), math.cos(x), math.exp(x)
%-------------------------------------------
```

乱数は import random を実行したうえで次のように実行する。

```
%--------------------------------------------
# ファイル名 [1.5b.py] (←この部分は書かなくてもよい)
import random
a=random
for i in range(5):
    print a.random()
%--------------------------------------------
```

ここで a は Python の乱数発生クラスのインスタンスである。(乱数発生機と考えると良い。)

システムコマンドを実行するには次の様にする。[注5]

```
%--------------------------------------------
# ファイル名 [1.5c.py] (←この部分は書かなくてもよい)
import os
os.system("dir")
%--------------------------------------------
```

自分で関数を定義したいときは def を用いて次のように書いて利用すること出来る。

```
%--------------------------------------------
# ファイル名 [1.5d.py] (←この部分は書かなくてもよい)
def square(x):
    return x*x
for i in range(5): print i, square(i)
%--------------------------------------------
%--------------------------------------------
# ファイル名 [1.5e.py] (←この部分は書かなくてもよい)
def max(x):
    max=-1.e20
    i=len(x)-1
    while (i>=0):
        if (x[i]>max): max = x[i]; imax=i
        i=i-1
    return [imax, max]
x=[-1.2, 3.5, -4.7, 2.9, -1.8]
print max(x)
%--------------------------------------------
```

配列を返すことができることに注意しよう。この機能は便利である。

Exercise 2:

(1) N 個の実数値、$x_0, x_1, ...x_{N-1}$ に対し、その平均と分散を返す関数を書け (ファイル名 Ex2.1.py)

(2) 2 次元平面状におかれた N 個の点の座標のリスト $[[x_1, y_1], [x_2, y_2], ...[x_N.y_N]]$ を受け取り、それらの点を結んでつくる多角形の面積を返す関数を書け (ファイル名 Ex2.2.py)

[注5] 結果は、"Windows Command Prompt"のほうの画面に出力されるので注意

1.6 モジュール

モジュールとは複数の関数などをまとめたライブラリのことである。モジュールをプログラムから利用するには、そのモジュールを import する。

1.6.1 基本的な数学関数

```
%-------------------------------------------
# ファイル名 [1.6.1a.py]（←この部分は書かなくてもよい）
import math
y = math.sin(x)
%-------------------------------------------
```

import math を使ったときには関数の前にモジュール名を書いて math.sin(x) のように書かなくてはならない。これがわずらわしければ次のように from math import *と書く。

```
%-------------------------------------------
# ファイル名 [1.6.1b.py]（←この部分は書かなくてもよい）
from math import *
y = sin(x)
%-------------------------------------------
```

math モジュールに含まれる主な関数・定数三角関数 sin(x) cos(x) tan(x) 逆三角関数 asin(x) acos(x) atan(x) atan2(x,y) 指数関数 exp(x)、自然対数 log(x)、常用対数 log10(x) 双曲線関数 sinh(x) cosh(x) tanh(x) 平方根 sqrt(x)、円周率 pi、自然対数の底 e

1.6.2 システムコマンド

次の命令により、os のコマンドを起動することができる。

```
%-------------------------------------------
import os
os.system("dir")
%-------------------------------------------
```

1.7 Numerical Python

1.7.1 Numerical Python

モジュールの中でも特に便利なのが Numerical Python である。Numerical Python は Python で数値計算をおこなうためのライブラリである。内部的には C で実装されており大容量の配列についての演算が高速に実行できる。

Numerical Python で定義される配列は array と呼ばれる。array は整数 (Int)、実数 (Float)、複素数 (Complex) についての 1 次元または多次元の配列を表す。

配列の定義は次のように行う。(詳しい説明は Numerical Python の website からダウンロードできるマニュアルを参照。)

```
%-------------------------------------------
# ファイル名 [1.7.1a.py] (←この部分は書かなくてもよい)
from Numeric import *
v = array([0,1,2,3])                        # 1 次元配列
a = array([[0,1,2,3],[10,11,12,13]])        # 2 次元配列
print len(v), len(a), len(a[0])             # 4 2 4
v = arange(4)                               # v=array(range(4)) と同じ
o = zeros([2,3],Int)                        # o=array([[0,0,0],[0,0,0]])
p = zeros([2,3],Float)                      # p=array([[0.,0.,0], [0.,0.,0]]
q = zeros([2,3],Complex)                    # q=array([[0.+0.j, 0.+0.j, 0.+0.j],
                                                      [0.+0.j, 0.+0.j, 0.+0.j]]
print o
print p
print q
%-------------------------------------------
```

配列の一部を取り出すのはリストと同様であるが、リストより便利な機能がついている(以下の例では上のプログラムに続けてプログラムを実行した結果を示す)。

```
%-------------------------------------------
# ファイル名 [1.7.1b.py] (←この部分は書かなくてもよい)
print v[0],v[1:4],v[1:4:2]          # 0 [1 2 3] [1 3],  v[1:4:2] の 2 は increment
print a[0][0],a[0]                  # 0 [0 1 2 3]
print a[0,0], a[0,:]                # a[0][0],a[0,0] のどちらの表記も許される
print a[1,1:4]                      # [11 12 13] (m:n は第 m 成分から第 n 成分までの意)
%-------------------------------------------
```

配列の間の演算は各要素に対して行われ、結果の配列を返す。

```
%-------------------------------------------
# ファイル名 [1.7.1c.py] (←この部分は書かなくてもよい)
print v*v                    # [0 1 4 9]   (各成分の 2 乗)
print v*v+v                  # [0 2 6 12]
print sin(v)                 # [0.  0.84147098  0.90929743  0.14112001]
%-------------------------------------------
```

異なるサイズの配列の間の演算は一般にはできないが、一方の配列の次元が他方の配列の次元よりひとつだけ小さい場合の演算は可能である。この場合は次元の小さい方の配列の繰り返し配列がつくられて演算が行われる。

```
%-------------------------------------------
# ファイル名 [1.7.1d.py] (←この部分は書かなくてもよい)
print v+1                       # v+array([1,1,1,1]) と同じ
print 3*v                       # array([3,3,3,3])*v と同じ
print a*v                       # a* array([v,v]) と同じ
%-------------------------------------------
```

3*v、3*a などはベクトルとスカラー、行列とスカラーの間の演算と同じであるが a*v などは行列とベクトルの間の演算（行列積の演算）とは異なることに注意する必要がある。行列積やベクトルの内積を得るためには次のようにする。

```
%-------------------------------------------
# ファイル名 [1.7.1e.py] (←この部分は書かなくてもよい)
print dot(a,v)                      # [14,74], a と v の行列積
print dot(v,v)                      # 14, v と v の内積
%-------------------------------------------
```

Exercise 3 :

(1) $\boldsymbol{u}, \boldsymbol{v}$ を 3 次元空間のベクトルとする。 array を用いて以下のベクトル演算を実行してみよ。 $\boldsymbol{u} + 2\boldsymbol{v}, \boldsymbol{u} \cdot \boldsymbol{v}, (\boldsymbol{u} + \boldsymbol{v})/2$ (Ex3.1.py)

(2) 3 次元空間におかれた 4 つの点 $\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4$ のつくる 4 面体の体積を返す関数を書け。(Ex3.2.py)

(3) 3 次元空間におかれた N 個の質点 $\boldsymbol{r}_1, \boldsymbol{r}_2, ...\boldsymbol{r}_N$ がすべて等しい質量をもっていたとする。これらの質点の重心の位置を返す関数を書け。また、重心を通り, 単位ベクトル $\boldsymbol{n}$ に平行な軸の周りの慣性モーメントを返す関数を書け。(Ex3.3.py)

1.7.2 LinearAlgebra モジュール（線形代数）

LinearAlgebra モジュールは Numerical Python の array オブジェクト（配列）を引数にとり、線形方程式の解 (solve_linear_equations)、逆行列 (inverse)、行列式の値 (determinant) 、固有値、固有ベクトル (eigenvectors) などを計算してくれる。

```
%-------------------------------------------
# ファイル名 [1.7.2.py] (←この部分は書かなくてもよい)
from Numeric import *
from LinearAlgebra import *
a=array([[2,-1,0],[-1,2,-1],[0,-1,2]])
b=array([0,1,0])
print solve_linear_equations(a,b) # ax=b を解く
a_inverse = inverse(a) # a の逆行列
eigen_values, eigen_vectors = eigenvectors(a)
# a の固有値と固有ベクトルを計算
%-------------------------------------------
```

Exercise 4:

(1) 3 次元空間におかれた N 個の質点 $\boldsymbol{r}_1, \boldsymbol{r}_2, ...\boldsymbol{r}_N$ がすべて等しい質量をもっていたとする。これらの質点の重心の周りの慣性モーメントの固有値と慣性主軸の方向を返す関数を書け。(Ex4.1.py)

(2) 1 次元のシュレディンガー方程式

$$-\frac{\hbar^2}{2m}\frac{d^2\phi}{dx^2} + V(x)\phi(x) = E\phi(x) \tag{1.1}$$

は空間を幅 Δx で多数の区間に等分し、波動関数 $\phi(x)$ を等分点での値 $\phi_n = \phi(n\Delta x)$ の集合 $(\phi_0, \phi_1, ...\phi_N.)$ で置き換えると、次のような行列の形式で書くことができる。

$$\sum_m H_{nm}\phi_m = E\phi_n \tag{1.2}$$

ここで

$$H_{nm} = -\frac{\hbar^2}{2m\Delta x^2}(\delta_{n,m-1} - 2\delta_{n,m} + \delta n, m+1) + V(n\Delta x)\delta_{n,m} \tag{1.3}$$

この固有値問題を解いて、任意のポテンシャル $V(x)$ に対する固有値と固有関数を求めるプログラムを書け。(Ex4.2.py)（物理仮想実験, 7 章参照)

1.7.3 モジュール（高速フーリエ変換）

FFT モジュールは配列のフーリエ変換を行う。複素フーリエ変換 (`fft`), フーリエ逆変換 (`inverse_fft`), 実フーリエ変換 (`real_fft`), 実フーリエ逆変換

```
%-------------------------------------------
# ファイル名 [1.7.3.py]（←この部分は書かなくてもよい）
from FFT import *
y = array([1,0,1,0,1,0,1,0])
fy = fft(y) # y のフーリエ変換
print fy,fy.real,fy.imaginary # fy,fy の実部、fy の虚部
ffy = inverse_fft(fy) # 数値誤差を除けば y と同じもの
%-------------------------------------------
```

1.7.4 モジュール RandomArray（乱数）

RandomArray モジュールは一様乱数 (`random, uniform, randint`)、正規乱数 (`normal`) などの多くのタイプの乱数を生成する。また整数のランダムな並べ替え (`permutation`) も行う。

```
%-------------------------------------------
# ファイル名 [1.7.4.py]（←この部分は書かなくてもよい）
from RandomArray import *
print random([10]) #[0,1] の一様乱数を 10 個生成
print uniform(0, 1.0, [10,3]) #単位立方体内に一様に分布する
#10 個の点を生成
%-------------------------------------------
```

`random()` は [0, 1] の区間にある一様乱数を返す。

`random(shape)`（ここで `shape` は整数のリスト）は `shape` で指定される乱数列を返す。

例えば `random([5])` は乱数列 $[x_0, x_1, ..., x_4]$ を生成し `random([3,5])` は乱数の 2 次元配列 $[[x_{0,0}, x_{0,1}, ..., x_{0,4}], [x_{1,0}, x_{1,1}, ..., x_{1,4}], [x_{2,0}, x_{2,1}, ..., x_{2,4}]]$ を返す。

`uniform(min,max,shape)` は $[min, max]$ の区間にある一様乱数の配列を返す。

`randint(min,max,shape)` は $[min, max]$ の区間にある整数乱数の配列を返す。

`standard_normal(shape)` は正規乱数の配列を返す。

1.7.5 モジュール RNG.Statistics モジュール （統計計算）

RNG.Statisitics は平均値 (`average`)、分散 (`variance`) の計算、ヒストグラム生成 (`histogram`) などを行う。

```
%-------------------------------------------
# ファイル名 [1.7.5.py]（←この部分は書かなくてもよい）
from RNG.Statistics import *
u = random([10000]) #10000 個の乱数について
print average(u), variance(u) #平均値と分散を計算
print histogram(u, 20, [-1.0,1.0]) #ヒストグラムを生成
%-------------------------------------------
```

Exercise 5 :

(1) 正規分布する乱数 x_1, x_2 の和 $(x_1 + x_2)/\sqrt{2}$、差 $(x_1 - x_2)/\sqrt{2}$ のヒストグラムを求め、それらが正規分布になっていることを確かめよ。(Ex5.1.py)

(2) $[-1, 1]$ の区間に分布する n 個の一様乱数 $x_1, x_2, \cdots x_n$ の和 $y = \sum_i x_i$ の分布を $n = 2, 3, 4, 10$ について求めよ。n が大きくなるとガウス分布になることを、分散を適当にスケールして正規分布になることを調べることで確かめよ。(Ex5.2.py)

(3) 単位球内に一様に分布する N この点の座標 $\boldsymbol{r}_1, \boldsymbol{r}_2, ...\boldsymbol{r}_N$ を返す関数を書け。このとき $\langle \boldsymbol{r}^2 \rangle$,$\langle \boldsymbol{r}^4 \rangle$ を計算し理論値と合っているかどうかチェックせよ。(Ex5.3.py)

(4) $\boldsymbol{r}_1, \boldsymbol{r}_2, ...\boldsymbol{r}_N$ が単位球内に一様に分布する N この点の座標であるとする。$\boldsymbol{R} = \sum_n \boldsymbol{r}_n$ とするとき、$\langle \boldsymbol{R}^2 \rangle$,$\langle \boldsymbol{R}^2 \rangle$ を計算し、理論値とあっているかどうかチェックせよ。(Ex5.4.py)

(5) n 次元の単位球の体積をモンテカルロ法により求めよ。（$[-1, 1]$ の区間に一様分布する成分を持つ n 次元ベクトル $\boldsymbol{r}$ の絶対値が 1 より小さくなる確率から球の体積を求める。）(Ex5.5.py)

1.8 グラフを書く

Gourmet では gnuplot を用いてグラフを書くことができる。ここではそのうちの 1 つ gnuplot2 モジュールを使った書き方を紹介する

```
%-------------------------------------------
# ファイル名 [1.8.py]（←この部分は書かなくてもよい）
import gnuplot2
```

```
x_data=[1,2,3,4]
y1_data=[1,4,9,16]
y2_data=[4,3,2,1]
gnuplot2.plot( 'set xlabel "x "; set ylabel "arbitrary unit" ',
               [ [ x_data, y1_data,  'title "data 1" with lines ' ],
                 [ x_data, y2_data,  'title "data 2" with lines ' ] ] )
%-------------------------------------------
```

gnuplot が起動してグラフが描画される。 plot() 関数は、 plot(pre_comand, plot_list, post_comand) という形式で用いる。 plot_list は“ プロット指定 ” のリストで、1 つの“ プロット指定 ” は [x_data, y_data, attr] というリストである。 x_data、y_data はそれぞれ x 軸、y 軸のデータをリストで与えます。 atr は gnuplot の plot コマンドに与える属性で、上の例では 'title "data 1 " with lines ' です。 pre_comand, post_comand は共に文字列で、gnuplot の plot コマンドの前と後に実行したい gnuplot のコマンドを与えます。上の例では pre_comand は 'set xlabel "x "; set ylabel "arbitrary unit" 'という文字列で、2 つの gnuplot コマンドを含んでいる。

Exercise 6 :

(1) $y = \sin(x)$,$y = \cos(x)$ の二つの曲線をプロットするプログラムを書け。(Ex6.py)

1.9 3 次元表示

以下の図 (左) のように Edit Window の上の列に並んだボタンの中から Window をクリックし、その中のメニューの Viewer を選択する。 すると、図 (右) のような Viewer ウィンドウが画面に現れる。

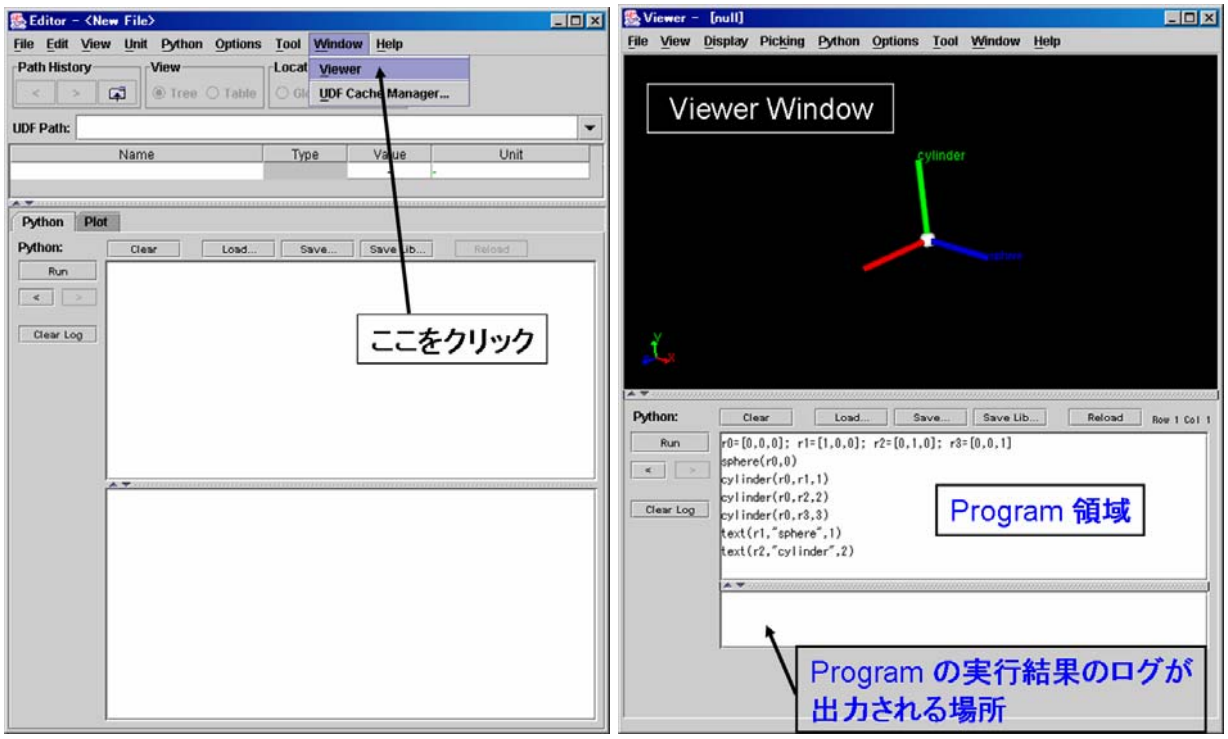

図 1.1: Viewer ウィンドウの起動と Viewer ウィンドウを使った簡単な例

ここに、Python を使って様々な 3 次元オブジェクトを 3D 表示できる。次のような簡単な例 (Python Program）を実行して理解していこう。以下のプログラムを Viewer のプログラム領域に書き込み、

```
%-------------------------------------------
# ファイル名 [1.9.py]（←この部分は書かなくてもよい）
r0=[0,0,0] ;r1=[1,0,0]; r2=[0,1,0]; r3=[0,0,1]
sphere(r0,0)          # 各オブジェクトの引数の最後引数はの色とサイズを指定するところ。
cylinder(r0,r1,1)     # ゼロは既成の色とサイズのパッケージ 0 番を使うという意味
cylinder(r0,r2,2)     # 色は、0: 白， 1:青， 2:緑　3:赤　4:灰色
cylinder(r0,r3,3)
text(r1,"sphere",1)
text(r2,"cylinder",2)
%-------------------------------------------
```

Run ボタンをクリック。すると図 1.1(右）のように 3 つの円筒と 1 つの球、2 つのテキストオブジェクトが画面に表示される。**表示したオブジェクトの拡大、縮小、移動などはマウスでできる。**

- 回転：　左ボタンでドラッグ
- 拡大・縮小；　右ボタンでドラッグ
- 移動： Shift キーを押しながら左ボタンでドラッグ

また、Alt キーを押しながら、キーボード上の上下左右の矢印キーあるいは [と] のキー（左右のカギ括弧文字）を押すことでも回転ができる（15　°刻み)。以下のようなオブジェクトを表示することができる。

- 点

  ```
  point(pos,[R,G,B,T,size])
  ```

  size はピクセル単位で、省略可能（デフォルトは 3 ピクセル)。
- 線

  ```
  line(pos1,pos2,[R,G,B,T,thickness])
  ```

  thickness は太さで、ピクセル単位で指定する。thickness を省略すると 1 ピクセルの太さになる
- ポリライン（複数の線分をつないだもの）

  ```
  polyline(pos list,[R,G,B,T,thickness])
  ```

  pos list に指定した点を順に線で結ぶ。thickness を省略すると 1 ピクセルの太さになる
- 多角形

  ```
  polygon(pos list,[R,G,B,T])
  ```

  pos list に指定した点を頂点とする多角形
- 円

  ```
  disk(center,[R,G,B,T,radius,nx,ny,nz])
  ```

center を中心に半径 radius の円盤。nx, ny, nz は円盤に垂直なベクトル

- 楕円：中点指定

```
ellipse1(center,[R,G,B,T,a,b,nx,ny,nz])
```

center を中心に a, b を長径・短径とする楕円。nx, ny, nz は楕円に垂直なベクトル

- 楕円：焦点指定

```
ellipse2(pos1,pos2,[R,G,B,T,a,nx,ny,nz])
```

pos1 と pos2 を焦点とし、長径が a の楕円。nx, ny, nz は楕円に垂直なベクトル

- 球

```
sphere(pos,[R,G,B,T,radius])
```

pos を中心とし、半径 radius の球

- 円柱

```
cylinder(pos1,pos2,[R,G,B,T,radius])
```

pos1 と pos2 を結ぶ線分を軸とし、半径 radius の円柱

- 円錐

```
cone(pos1,pos2,[R,G,B,T,radius])
```

pos1 を底面の中心、pos2 を頂点とする円錐。radius は底面の半径

- 楕円体

```
ellipsoid1(center,[R,G,B,T,a,b,c,ax,ay,az,cx,cy,cz])
```

center を中心とし、2a, 2b, 2c を主軸の長さとする楕円体。主軸 a の方向は (ax, ay, az)、主軸 b の方向は (ax, ay, az) と (cx, cy, cz) の両方に垂直な方向

- 回転楕円体

```
ellipsoid2(pos1,pos2,[R,G,B,T,a])
```

pos1 と pos2 を焦点とし、長径が a の回転楕円体

- 四面体

```
tetra(pos1,pos2,pos3,pos4,[R,G,B,T])
```

pos1 から pos4 を頂点とする四面体

- 立方体

```
cube(pos,length,[R,G,B,T])
```

pos を中心とする、1 辺 length の立方体

- 矢印

```
arrow(pos1,pos2,[R,G,B,T,radius,height])
arrow(pos1,pos2,[R,G,B,T,radius,height,magnification])
```

pos1 から pos2 へ向かう矢印。矢じ りを円錐と見たときの半径を radius、高さを height に指定する。magnification を指定すると、矢じ りの大きさは変えずに、矢の長さを magnification 倍する

- 文字列

```
text(pos,string,[R,G,B,T,size])
```

位置 pos を開始点にして文字列 string を表示。size は文字のサイズをポイント単位で指定。詳細は物理仮想実験を参照

Exercise 7 :

(1) simple cubic (sc), body centered cubic (bcc), face centered cubic (fcc) の結晶構造を表示するプログラムを書け. (Ex7.1.py)

(2) ランダムウォークの軌跡を書け. (Ex7.2.py)

1.10 場の表示

Gourmet において、場の表示を行うには meshfield クラスを用いる。 meshfield は、格子点あるいは、4 面体要素の節点の上で定義された場の値を補間して、場の等値面の表示、断面における場の強さの色表示、ベクトル場の矢印による表示などを行うクラスである。ここでは、簡単な例として立方格子上で定義されたポテンシャルを描くプログラムを紹介する。

```
%-------------------------------------------
# ファイル名 [1.10.py] (←この部分は書かなくてもよい)
[Lx,Ly,Lz,Nx,Ny,Nz]=[1.0,1.0,1.0,10,10,10]
field = meshfield("regular",[[0,Lx],[0,Ly],[0,Lz]],[Nx,Ny,Nz])     # 文1
for i in range(0,Nx+1):
     for j in range(0,Ny+1):
          for k in range(0,Nz+1):
               [x,y,z]=[i*Lx/Nx,j*Ly/Ny,k*Lz/Nz]
               field.set([i,j,k],x*x+y*y+z*z)                       #文2
field.clevel(0.5,0.5,2)                                           #文3
field.ccolor([0,0,1,1,1,0,1,1])                                   #文4
field.cplane([1,1,1],  [1,0,0])                                   #文5
field.cplane([1,1,1],  [0,1,0])
field.cplane([1,1,1],  [0,0,1])
field.draw()                                                      #文6
field.delete_clevel(); field.delete_cplane()                      #文7
%-------------------------------------------
```

文 1 は `meshfield` のひとつのオブジェクトである `field` を生成している。 `meshfield` のオブジェクトを生成するには、メッシュのタイプ、領域、分点の数を指定する。ここでは 3 辺の長さ L_x, L_y, L_z の直方体領域のそれぞれに分点 N_x, N_y, N_z を設定している。 `field` に値を設定するには文 2 のように格子のインデックス [i,j,k] とその点での `field` の値を設定する。文 3 は `field` に等値面を描くよう指示を与えている。メソッド `clevel` は `clevel(value1,value2, color_attr)` のように 3 つの引数をとり、 `value1` と `value2` の間にある値をつないだ面を `color_attr` を用いて表示する（通常 `value1=value2` で用いる）。 `color attr` は 2-1-6 で述べたように `[r,g,b,t]` の 4 つの数字で指定することもできる。文 4 は `field` の強さを色で表示するための色指定を行っている。 `ccolor` の引数は `[r1,g1,b1,t1,r2,g2,b2,t2]` の形式であり、 `field` の最小値を `[r1,g1,b1,t1]` の色で、最大値を `[r2,g2,b2,t2]` の色で表示するよう指示を与えている。文 5 は `field` の色表示を行う断面を指定している。 `cplane` は `[x,y,z],[nx,ny,nz]` の引数をとり、点 `[x,y,z]` をとおり法線ベクトルが `[nx,ny,nz]` であるような断面について `field` の色表示を行うよう指示している。文 6 の `draw()` 命令で以上の指示が実行され、Viewer に表示されます。文 `delete_clevel()`、 `delete_cplane()` は等値面や断面の指定を取り消す。

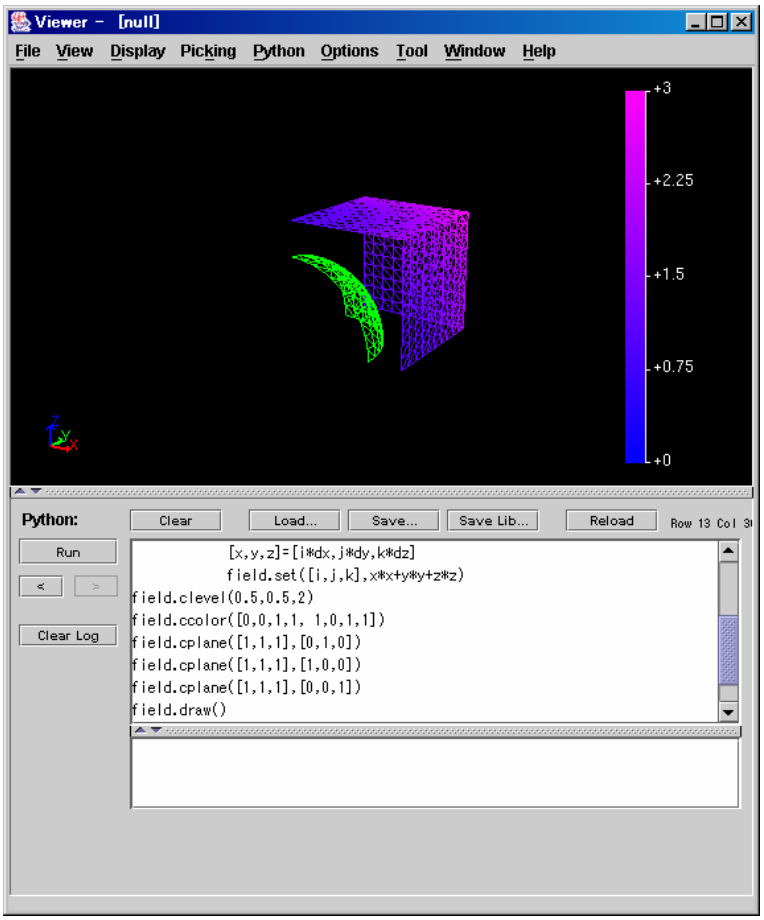

図 1.2: Viewer 場の表示例

第 2 章

UDF によるデータの表現

Gourmet のうえで様々なプログラムをつくることができるが、プログラムが複雑になると、入力の項目が増えて、どれが変えるべきパラメータであり、どれが変えてはいけない変数であるかわからなくなる。またプログラムの出力結果を別のプログラムで利用しようとしたとき、出力項目のなかでなにが必要な項目であるかわからなくなる。

このような問題を解消するために考えられたものが UDF (User Definable Format) である。UDF とは User Definable Format の略であり、定められたデータ形式でなく、それを使うユーザが自由に決めることのできるデータ形式である。

UDF で書かれたデータに対して Gourmet は様々なサービスを提供している。UDF データについては、入力データの編集、結果データの表示、グラフ化、アニメーション作成などが、Gourmet の機能、や Python プログラミングを使って簡単に実現することができる。

まず UDF とはどんなものであり、それに対して Gourmet がどんなサービスを提供するのかをざっと見てみる。

2.1 UDF ファイルに対する Gourmet のサービス概要

テキストエディタを使って次のようなファイルを作る。

```
%-----------------------------------------
# ファイル名 [2.1.udf]（←この部分は書かなくてもよい）
\begin{def}
     i:int
     a:float
     s:string
     x:float [m]
     v:float [m/s]
     array[]:float [kg]
     pos:{x: float [m], y:float [m], z:float [m]} "position of the particle"
     trajectory[]:{time:float [s], x:float [m], v:float [m/s]}
\end{def}
```

```
\begin{data}
     i:2
     a:3.4
     s:"distance and velocity"
     x:2
     v:4
     array[]:[0.2, 0.3, 0.4]
     pos:{1,-1,2}
     trajectory[]:[{0,0,1}, {1,1,2},{2,3,3}]
\end{data}
%-----------------------------------------------
```

ここに示したデータは UDF(User Definable Format) 形式のデータと呼ばれる。UDF は`\begin{def}`と`\end{def}` の間に書かれたデータ形式の定義部分と`\begin{data}` と `\end{data}`の間に書かれたデータそのものからなっている。

これを 2.1.udf という名前をつけてセーブする。次に Gourmet の File メニューのオープンから 2.1.udf を開くと今書いたファイルの内容を見ることができる。

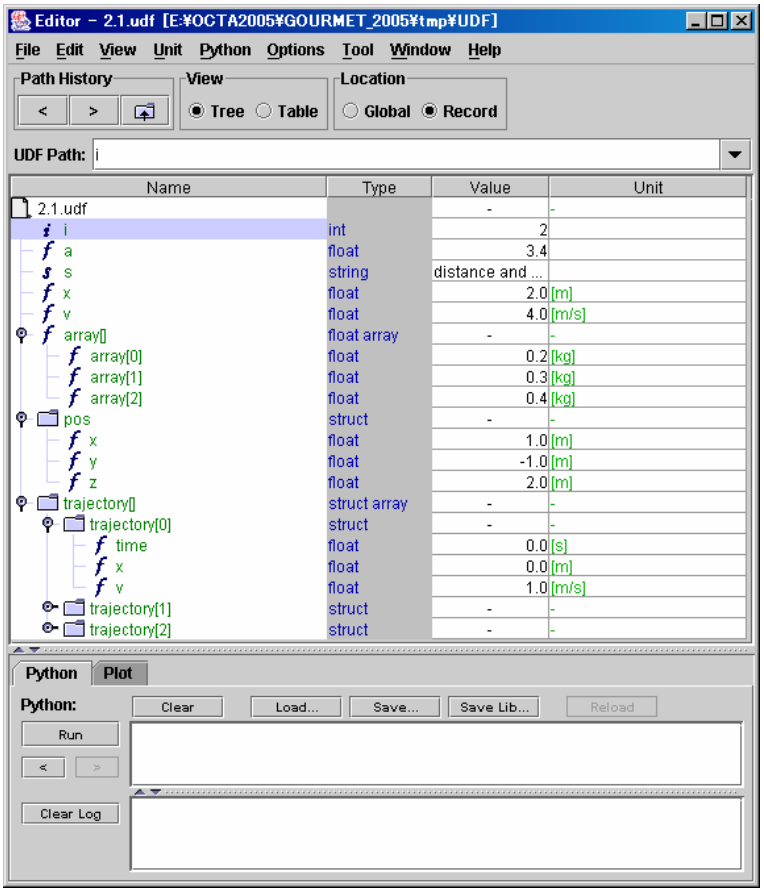

図 2.1: UDF 形式のデータ表示の例

まず、薄く影になっている変数 i を見ていただきたい。i は整数型（int 型）のデータでありその値は 2 であることが分かる。その下の変数 a は実数型（float 型）のデータでありその内容は 3.4 であることが

分かる。変数 s は文字型 (string) でありその内容は ”distance and velocity ”である。x は [m] という単位を持った float 型のデータでありその内容は 2.0[m] である。v は [m/s] という単位を持った float 型のデータでありその内容は 4.0[m/s] である。

単位付のデータに対しては、別の単位系で表示することが可能である。a の [m] と書かれたところをマウスで右クリックし、[cm] の単位を選択すると、a の表示が 2.0[m] から 200.0[cm] に変わる。同様に v を 400.0[cm/s] と表示することができる。後で述べるが、ユーザは自分の単位系を定義することが可能であり、天文単位、原子単位などを定義して用いることができる。

`array[]` と書かれたデータは [kg] という単位を持った float 型のデータの配列である。C と同じく配列の各要素は `array[0],array[1]` という名前で参照することができる。配列の中身を見るには左窓の `array[]` をマウスでクリックすると `array[0],array[1],array[3]` などの項目が現われて、その値が右窓に表示される。配列の添え字は 0 から始まる。

`pos` と書かれたデータは構造を持っている。右窓の下の `pos` をマウスでクリックすると `x,y,z` という下部の構造が現われそれぞれの値は 1.0[m],-1.0[m],2[m] であることが示される。C と同じく構造型のデータはピリオド ”.”をつけて参照することができる。この例ではそれぞれのデータは `pos.x`, `pos.y`, `pos.z` という名前で参照される。

`trajectory[]` と書かれたデータは構造型データの配列である。配列の各要素は `time, x,v` という項目からなっており、各要素は `trajectory[0].time`, `trajectory[2].x` などの名前で参照することができる。

UDF データにつけられた名前は UDF Path と呼ばれる。自分に必要なデータを取り出したいときには UDF Path:と書かれた窓の中にデータの名前を入力してやればよい。Gourmet は、入力された UDFPath にジャンプして、データを表示してくれる。

しかし、データ処理を行うには手で入力するのではなく Python のプログラムを用いるほうが便利である。Gourmet の Python には特別な機能拡張があり、上に示したデータを Python の変数と同様に扱うことができる。次のような Python プログラムを走らせて見る。

```
%-------------------------------------------
# ファイル名 [2.1a.py] (←この部分は書かなくてもよい)
print $a, $array[0], $pos.x
%-------------------------------------------
```

下段に結果が現われる。この結果から分かるように、**UDF データは UDF Path にの先頭に$記号を付けて参照することが出来る。**

Python プログラムを用いてデータの変更をすることも可能である。次のプログラムを Python 窓に書き、走らせてみよ。(Python 窓上段のプログラムを消すには Clear ボタンを押す。また Python 窓下段の結果を消すには Clear Log ボタンを押す。)

```
%-------------------------------------------
# ファイル名 [2.1b.py] (←この部分は書かなくてもよい)
print "before a=", $a
$a = - $a
print "after a=", $a
%-------------------------------------------
```

構造体のデータや配列データに対しては、個々の要素だけでなく全体を一度に扱うことも可能である。次のプログラムを走らせてみよう。

```
%-----------------------------------------
# ファイル名 [2.1c.py]（←この部分は書かなくてもよい）
print $array[]
print $pos
%-----------------------------------------
%-----出力-------------------------------------
[0.2, 0.3, 0.4]
[1.0, -1.0, 2.0]
%-----------------------------------------
```

これまではデータの値を見るときに Tree モードを用いてきた。このモードでは、すべてのデータは data view window の右側に縦に表示される。一方、データの値をみるとき、表の形式で見たほうが見やすい場合もある。この場合には Table モードを用いると良い。Tree モード、Table モードの選択は View のボタンを切り替えることで実行できる。

使っているシステムが Window であれば、Gourmet 画面上のデータを選択して、クリップボードにコピーすることができる。またクリップボード上のデータをデータの上にペーストすることができる。このようにして、テキストエディタや Excell のプログラムとの間でデータのやり取りをすることができる。Gourmet の操作法の詳細は ”物理仮想実験 ”か Gourmet のマニュアルを参照されたい。

2.2 UDF ファイル

UDF は \begin{def}と \end{def}の間に書かれた定義部と \begin{data}, \end{data} の間に書かれたデータ部からなる。定義部ではデータの型 (実数、整数、配列、構造型) やそれぞれの単位を定義し、データ部でその実際の値を与える。UDF で書かれたファイルを Gourmet で開くと、データの一覧、編集解析を容易に行うことができる。プログラムの扱うデータ形式を UDF で書くことにより、データの意味が明らかとなり、プログラムが使いやすいものとなる。

2.2.1 単純な形のデータ

UDF ファイルの例を示す。

```
%-----------------------------------------
# ファイル名 [2.2.1a.udf]（←この部分は書かなくてもよい）
\begin{def} // 定義部
    title : string "title of the project"
    Nmax: int "maximum value of the array"
    xmax: double "maximum value of the random number"
    result[]: double "list of the random number"
    result_2D[][]:double "double array "
\end{def}
```

```
\begin{data} //データ部
    title: "test of random number"
    Nmax: 10
    xmax: 10.0
    result[]: [ 2.4 3.2 2.5]
    result_2D[][]:[[1.1, 2.2]. [2.3, 2.4]]
\end{data}
%-----------------------------------------
```

定義部

(1) データ名定義部において//の後に書かれたものはコメントである。コメントは Gourmet では無視される。一方 ””で囲まれたものは変数に対する説明文である。説明文はユーザに表示される。

(2) 単純型データには short, int、double、string がある。これ以外にユーザがデータの型を定義することができる。(クラス定義参照)

(3) データ名の後ろに［］をつけてデータが配列であることを表する。配列の大きさをデータ名定義部に書く必要はない。[][] は 2 次元配列、[][][] は 3 次元配列を表す。

データ部

(1) データには UDF 変数名と ”: ”に続けてデータの値を書く。

(2) 定義部で定義された UDF 変数名はデータ実体部にどんな順序で現れても構わない。また定義部で定義された UDF 変数名がデータ実体部になくても構わない。

(3) 配列データの始まりと終わりには ”[”と ”] ”をつける。配列の各要素はコンマ、タブ、改行など任意のもので区切ることができる。

(4) データ部において、+-および数字で始まるものは int または double 型のデータとみなされる。string 型のデータは ””で区切って表する。これ以外はすべてコメントとみなされる。

(5) （3）-（4）に述べた規則を用いると、UDF のデータ実体部の書き方はかなり自由になる。

上述したルールを用いることで、上に示したデータは次のように書くこともできる。

```
%-----------------------------------------
\begin{data}
# ファイル名 [2.2.1b.udf]（←この部分は書かなくてもよい）
Name of the project
     title: "test of random number"
Maximum number of array
     Nmax: 10
Maxumum value of the random number
     xmax: 10.0
Generated random number
     result[]: [
          first 2.4
          second 3.2
```

```
          third 2.5
          ]
\end{data}
%-----------------------------------------
```

ここで Name of the project、 Maximum number of array、 Generated random number などは ””で囲まれていないのでコメントである。

2.2.2 単位

UDF 変数に単位をつけることができる。単位をつけるには定義部で型の後ろに［m], [kg] のように単位をつける。

```
%-----------------------------------------
# ファイル名 [2.2.2a.udf]（←この部分は書かなくてもよい）
\begin{def}
  x:float [m]   "position of the particle"
  v:float [m/s] "velocity of the particle"
  e:float [J]   "energy of the particle"
\end{def}
\begin{data}
  x: 1
  v: 2
  e: 5
\end{data}
%-----------------------------------------
```

ユーザが単位を定義することもできる。

```
%-----------------------------------------
# ファイル名 [2.2.2b.udf]（←この部分は書かなくてもよい）
\begin{unit}
        [L]=0.1[nm] , [T]=1[fs], [M]=2.E-26[kg]
        [E] = [M*L^2/T^2]
\end{unit}
\begin{def}
        atom[]:{x: double [L], v: double [L/T], energy:double [E] }
\end{def}
\begin{data}
        atom[]:[{1, 2, 0.3}, {2, -1,0.4}]
\end{data}
%-----------------------------------------
```

このような UDF ファイルを Gourmet で読み込みとデフォルトではユーザの定義した単位をつけて表示される。

[**単位系の変換**] デフォルトの単位系から SI 単位系に変換して表示することができる。Unit メニューの下の Select Unit Set をクリックすると単位系を選択できる。このとき選択できるのはデフォルト単位系 (読み込んだ UDF ファイルの中で定義された単位系）と SI 単位系である。もしユーザが別に単位系を定義して登録しておけば、ユーザの好みの単位系で表示することもできる。

単位の変換は個々のデータごとに行うことができる。変更したい単位にマウスをあわせて左ボタンを押すと、設定したい単位の入力が要求される。いくつかの単位の中から選ぶこともできるし、ユーザが定義した単位を入力することも可能である。OK ボタンを押すと単位が変更されるのがわかる。

2.2.3 構造型データ

C や Fortran90 のサポートする構造型のデータに対応するものである。構造型データの例を下に示す。

構造型データの定義部

```
%-------------------------------------------
# ファイル名 [2.2.3a.udf] (←この部分は書かなくてもよい)
\begin{def}
    vector:{x:double [m], y:double [m], z:double[m]}
    simulation_condition:{
        temperature: double [K]
        box:{Lx:double[m], Ly:double[m], Lz:double[m]}   }
\end{def}
%-------------------------------------------
```

構造型変数を定義するには { と } の間に要素となる変数の定義を書く。要素となる変数は上の例の box のように構造型変数であっても構わない。構造型変数の要素の値は simulation_condition.temperature のように参照できる。 simulation_condition.temperature、 simulation_condition.box.Lxなどは単純型変数であるが、 simulation_condition.box、 simulaiton_condition などは構造型変数である。

構造型データのデータ部

上に定義したデータの実体部は以下のように書く。

```
%-------------------------------------------
# ファイル名 [2.2.3b.udf] (←この部分は書かなくてもよい)
\begin{def}
    vector:{ 1.0 3.0 -1.0}
    simulation_condition:{ 4.0 { 1.0 1.0 3.0} }
\end{def}
%-------------------------------------------
```

構造型変数のデータは、単純型変数の時と同じように UDF 変数名: に続けて書く。simulation_condition のように最上部の UDF 変数名を書く必要はあるが、その要素名を書く必要はない。simulation_condition がどのような要素を持っているかは、定義部で分かっているからである。

2.2.4 クラス

同じ名前の要素を持つものを一つの型として定義すると便利である。例えば 3 次元空間のベクトルを表す量は常に x, y, z の 3 つの成分を持っているので、これらをまとめてベクトル型としておくと便利である。UDF においてデータ型を定義する時には class というキーワードを用いる。class という名前はこれが C++のクラスを反映したものだからである。UDF クラスの定義とその利用法の例を以下に示する。

```
%-------------------------------------------
# ファイル名 [2.2.4.udf] (←この部分は書かなくてもよい)
\begin{def}
   class Vector3d:{ x: double[unit], y:double[unit], z:double[unit]}[unit]
   class Atom:{ pos: Vector3d [nm], vel: Vector3d [m/s]}
   atom[]:Atom
\end{def}
%-------------------------------------------
```

`class Vector3d:{ x: double[unit], y:double[unit], z:double[unit]}[unit]` は、`Vector3d` というデータ型が `x、y、z` という名前のついた 3 つの double 型要素をもつもので, かつ各要素は同じ単位 [unit] を持っていると宣言している。

`class Atom:{ pos: Vector3d [nm], vel: Vector3d [m/s]}` は `Atom` というデータ型が、`pos、vel` という `Vector3d` 型の 2 つの要素をもっていることを宣言している。

`pos` の単位は [nm] であり、`vel` の単位は [m/s] である。`atom[]:Atom` はそのような `Atom` 型のデータの配列を表す。

2.3 Action

自分で作った Python プログラムを UDF 変数に関連して記憶させることができる。例えば intro.udf の `pos\verb` の絶対値を計算するプログラムを考えよう。まず以前作成した 2.1.udf のコピーを 2.3.udf という名称で作成する。そして、次のようなテキストファイルを作る。[注1]

```
%-------------------------------------------
# ファイル名 [2.3a.act] (←この部分は書かなくてもよい)
action pos : norm() : \begin
print $pos.x**2 + $pos.y**2 + $pos.z**2
\end
%-------------------------------------------
```

これを 2.3a.act という名前をつけて 2.3.udf と同じディレクトリにセーブする。次に 2.3.udf を開き File メニューの Header を開きアクションファイルを 2.3a.act とし、このファイルをセーブする。[注2] 再び 2.3.udf を読み込み、Tree Window のなかの `pos` の項目を右ボタンでクリックすると `norm` というメッセージが出てくる。左ボタンを押すと、Python Program Window に上のプログラ

[注1] `\begin` の次の行の print の前に、タブを入れたくなる (個人差あり) が、入れると実行されないので注意

[注2] 単に、今回だけ実行したいのなら、セーブする必要はない.

ム `print $pos.x**2 + $pos.y**2 + $pos.z**2` が表示され、Python Log Window に結果が表示される。このように **Action とはデータに関係付けられた Python プログラムを実行することである。**Python プログラムはどのデータに関連づけても良いが、Action の機能に近いデータに関連付けるべきである。

Action の内容は以下のように制御文を含んだ複雑なプログラムにすることも可能である。ここでは、例として sup-norm() というアクションを示す。ファイル 2.3a.act に一部変更を行い保存する。その後、”File”→”Reload Action”を行えば変更した Action が利用可能である。

```
%-------------------------------------------
# ファイル名 [2.3b.act] (←この部分は書かなくてもよい)
action pos : norm () : \begin
print $pos.x**2 + $pos.y**2 + $pos.z**2
\end
action pos : sup-norm () :\begin
max=abs($pos.x)
if max < abs($pos.y) : max = abs($pos.y)
if max < abs($pos.z) : max = abs($pos.z)
print max
\end
%-------------------------------------------
```

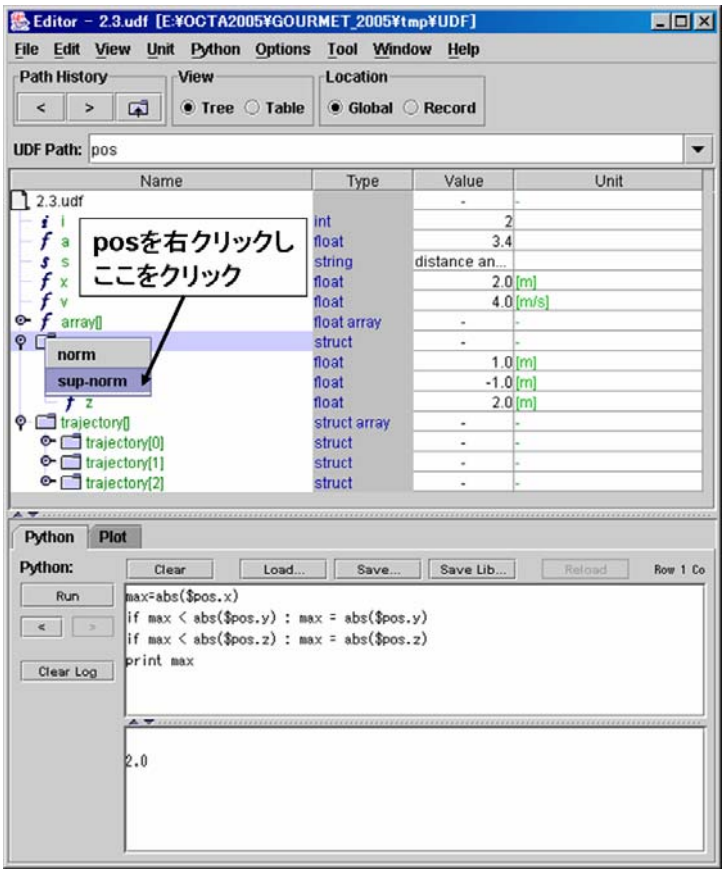

図 2.2: Action の実行例

もう少し便利な例として trajectory をグラフに描画することを考える。2.3a.act を次のように変える。

```
%-------------------------------------------
# ファイル名 [2.3c.act]（←この部分は書かなくてもよい）
action pos : norm() : \begin
print $pos.x**2 + $pos.y**2 + $pos.z**2
\end
action trajectory[]: plot() : \begin
import gnuplot2
t = $trajectory[].time
x = $trajectory[].x
gnuplot2.plot( 'set xlabel "t"; set ylabel "x" ',[[t,x, 'with lines ']])
\end
%-------------------------------------------
```

これをセーブし、Gourmet のファイルメニューで ReloadAction を押す intro.udf に登録されたアクションファイルが更新される。trajectory[] の項目を右ボタンでクリックし左ボタンを押すと、trajectory の内容がグラフにプロットされる。

一つのデータ項目にいくつのアクションを定義しても良い。

```
%-------------------------------------------
# ファイル名 [2.3d.act]（←この部分は書かなくてもよい）
action pos : norm() : \begin
print $pos.x**2 + $pos.y**2 + $pos.z**2
\end
action trajectory[]: plot() : \begin
import gnuplot2
t = $trajectory[].time
x = $trajectory[].x
gnuplot2.plot( 'set xlabel "t"; set ylabel "x" ',[[t,x, 'with lines ']])
\end
action trajectory[]: plot_v() : \begin
import gnuplot2
t = $trajectory[].time
v = $trajectory[].v
gnuplot2.plot( 'set xlabel "t"; set ylabel "v" ',[[t,v, 'with lines ']])
\end
%-------------------------------------------
```

第 3 章

プログラムからの UDF 入出力

3.1 はじめに

既に作成済みのシミュレーションエンジン、あるいはこれから開発するエンジンで Gourmet/UDF の機能を使おうとする場合、何らかの方法でエンジンが UDF ファイルの入出力を行う必要があります。これを実現するのに、3 つの方法があるでしょう：

1. エンジンの独自フォーマットファイルと UDF ファイルの変換フィルタを作る
2. エンジンが UDF ファイルを直接入出力する
3. エンジンを Gourmert に統合し、Gourmet からのエンジンの制御まで可能にする

本稿では 1. だけを扱いますが、ここで説明する知識だけで 2. は容易に実現出来ます。3. の統合については、『物理仮想実験室』の第 10 章や、libplatform のマニュアル等を参照して下さい。

また、既に UDF 化されているエンジン（たとえば Cognac）の出力 UDF から、必要なデータだけ取り出して別の解析プログラム（たとえば AVS）の入力ファイルを作りたい場合にも、変換フィルタを作成することで対応できます。

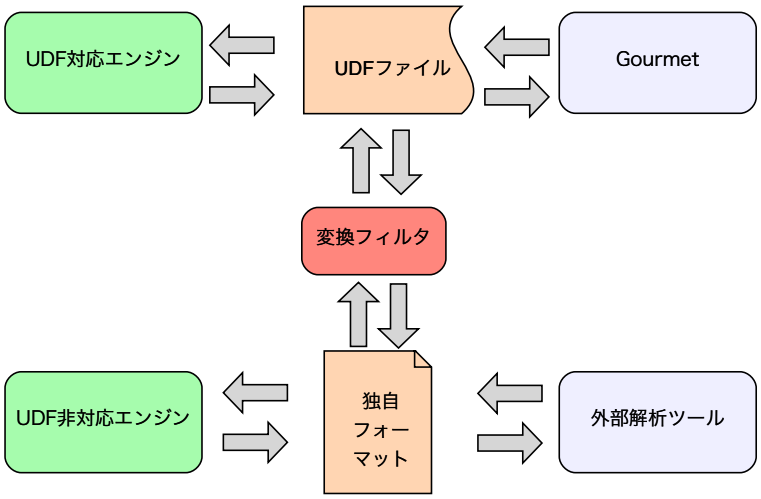

図 3.1: 変換フィルタによる UDF ファイルと他フォーマットの変換

変換フィルタを作成するのに用いることの出来るプログラミング言語は、Python, C, Fortran 及び C++ があります。本稿ではまず最も手軽な Python を利用した変換フィルタの作成について説明し、その後 C 言語からの UDF 入出力について説明します。Fortran については省略しますが、流れは C 言語の場合と殆ど同じです。

なお、本章で用いる UDF ファイルや C プログラムのソースファイルは、sample というフォルダの中に準備してあります。

3.2　Python からの UDF 入出力の補足

Gourmet を用いて Python から UDF を読み書きする方法は既に 2 章で説明されていますが、復習もかねて、すこし補足しておきます。

3.2.1　構造体の入出力

例として以下のような UDF ファイルを考えましょう：

```
//===== python/udf/simple.udf =====
\begin{global_def}
Temperature: double
Atom: {
    Name: string
    AtomicNumber: int
    AtomicWeight: double
    Pos: {
        x: double
        y: double
    }
}
\end{global_def}

\begin{data}
Temperature: 300.0
Atom: { "Carbon" 6 12.0 {1.0 2.0} }
\end{data}
```

この UDF をまず Gourmet で開いてください（sample/python/udf/simple.udf にあります）。その後、Gourmet の Python プログラム領域 に、たとえば

```
print $Temperature
print $Atom
print $Atom.Pos.y
```

と入力して実行すれば、

```
300.0
['Carbon', 6, 12.0, [1.0, 2.0]]
2.0
```

のように出力されます。このように、UDF パス名の先頭に $ を付けることで UDF 変数の値が取り出せます。Atom のような構造体全体を指定した場合、Python のリスト [...] として値が得られます。もちろん、print 文で出力するのではなく、Python 変数に値を格納することも出来ます：

```
T = $Temperature
atom = $Atom
y = $Atom.Pos.y
```

さらに、Python の多重代入の機能を使えば、構造体の要素を別々の変数に分けて格納することも出来ます：

```
[name, an, aw, [x, y] ] = $Atom
print name, an, aw
print x, y
```

あるいは

```
[name, an, aw, pos ] = $Atom
```

なども可能です。

UDF 変数に値を設定するのも簡単です：

```
$Temperature = 350.0
pos = [10.0, 20.0]
$Atom = ['Chlorine', 17, 35.5, pos]
```

これを実行して、実際に UDF 変数の値が変更されることを確認して下さい。(変更されるのはメモリ上の値だけで、保存しない限り、UDF ファイル中の値は変更されません。)

3.2.2 配列の入出力

つぎに配列の入出力を見てみましょう。

```
//===== python/udf/array.udf =====
\begin{global_def}
a[]: double
aa[][]: double
b[]: {
    x: double
```

```
    y: double
}
\end{global_def}

\begin{data}
a[]: [ 0.0 1.1 2.2 3.3 ]
aa[][]: [ [0.0 0.1 0.2 ] [1.0 1.1 1.2 ]  ]
b[]: [ { 0.0 0.1 } { 1.0 1.1 } { 2.0 2.1 } ]
\end{data}
```

この UDF からデータを読むには

```
p = $a[1]
q = $aa[2][1]
r = $b[0]
a = $a[]
aa = $aa[][]
b = $b
```

などが使えます。上記を実行し、さらに p,q,r,a,aa,b の値を print してみて下さい。配列全体を Python 変数に代入した場合、やはりリストとして格納されていることがわかると思います。多次元配列 aa[][] や構造体の配列 b[] は「リストのリスト」になります。なお、a = $a などはエラーになりますので注意してください。

Python のリストの要素数は関数 len() で得られます。

```
print len(a), len(aa), len(aa[0])
---> 4 2 3
```

あるいは、UDF ファイルから直接 UDF 配列のサイズを取得することも出来ます：

```
print size('a[]'), size('aa[][]'), size('aa[]'), size('aa[0][]')
---> 4 6 2 3
```

len() は Python の標準機能で、 size() は Gourmet による拡張機能です。

配列の一部、あるいは全部のデータの変更は以下のようにします：

```
$aa[0][1] = 99.9
$a[] = [0.0, 10.0, 20.0, 30.0]
$b[1].y = 101.1
```

さらに、配列要素の追加や削除も可能です。たとえば、今の例では a[0], a[1], a[2], a[3] が既存ですが、

```
$a[5] = 50.0
```

により a[4], a[5] が追加され、後者が 50.0 に設定されます。値が未設定の a[4] には、ファイルを保存する際に、デフォルトの値（int なら 0, double なら 0.0, string なら ”) が代入されます。UDF 配列要素の削除には関数

```
erase(num, location)
```

を用います。これにより、location で指定した要素から始めて num 個の要素が削除されます。たとえば

```
erase(3,'a[3]')
```

により、a[3] から a[5] までが削除されます。

3.2.3 レコードの取り扱い

レコードについては既に説明しました。ここでは、Python プログラムによりレコード間を移動したり、レコードの追加・削除を行う方法を説明します。

まず、record.udf を Gourmet で開いてください。

```
//===== python/udf/record.udf =====
\begin{global_def}   // グローバルデータ用の定義
T: double
\end{global_def}

\begin{def}          // レコードデータ用の定義
P: double
\end{def}

\begin{data}         // グローバルデータ
T: 273.0
\end{data}

\begin{record}{"#0"} // 最初のレコードデータ
\begin{data}
P: 0.0
\end{data}
\end{record}
// 以下略
```

このファイル中にレコードがいくつあるかは、関数 totalRecord() により知ることが出来ます：

```
print totalRecord()
---> 4
```

この場合、レコード 0 からレコード 3 までの 4 個のレコードが存在します。

現在どのレコードに居るかは currentRecord() で知ることが出来ます。UDF ファイルを開いた直後は、レコード 0 に居ると思います。

別のレコードに移動するには、jump(record_number) を用います。たとえば

```
jump(2)
print currentRecord()
print $P
```

によりレコード 2 に移動します。

レコードの追加は関数 newRecord() で行います。追加後は新しいレコードに移動します。

```
newRecord()
print currentRecord()
---> 4
$P = 16.0       # 追加したレコードの P を設定
```

3.3 例題エンジン： Udon

実際にファイル変換プログラムを作成してみるための例として、Udon というエンジンを準備してありますので、これについて説明しておきます（Wiondows 用の実行ファイルが udon-simulator/udon.exe に、ソースファイルが udon-simulator/src/ にあります）。

Udon は、一本の高分子鎖の両端を固定して熱運動をさせ、分子鎖の発生する張力を測定するプログラムです。分子鎖のモデルは、Kremer-Grest のモデルをさらに簡単化したものです。一本の分子鎖は $N+1$ 個のビーズを繋いだもので、0 番と N 番のビーズの位置は固定されています。

ビーズ j $(j = 1, 2, \ldots, N-1)$ の運動方程式は

$$m\ddot{\mathbf{r}}_j = \mathbf{F}_j - \zeta\dot{\mathbf{r}}_j + \mathbf{f}_j(t) \tag{3.1}$$

ここで $\mathbf{f}_j(t)$ は熱浴から受けるランダム力、ζ は熱浴との摩擦係数で、揺動散逸定理

$$\langle f_{j\alpha}(t_1) f_{k\beta}(t_2)\rangle = 2\zeta k_B T \delta_{jk}\delta_{\alpha\beta}\delta(t_1 - t_2) \tag{3.2}$$

を満たします。$\mathbf{F}_j$ は隣のビーズとの結合によりビーズ j に働く力

$$\mathbf{F}_j = -\frac{\partial U_{total}}{\partial \mathbf{r}_j} \tag{3.3}$$

$$U_{total} = \sum_{j=0}^{N-1} U(r_{j,j+1}) \tag{3.4}$$

です。繋がれたビーズ間に働くポテンシャル $U(r)$ は

$$U(r) = U_{FENE}(r) + U_{LJ}(r) \tag{3.5}$$

$$U_{FENE}(r) = -\frac{1}{2}kr_{max}^2 \log\left[1-(r/r_{max})^2\right] \tag{3.6}$$

$$U_{LJ}(r) = \begin{cases} 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} + \frac{1}{4}\right] & (r < r_c \equiv 2^{1/6}\sigma) \\ 0 & (r > r_c) \end{cases} \tag{3.7}$$

とします。$U_{LJ}(r)$ も繋がれたビーズ間にしか働きませんので、分子鎖は自分自身と重なってしまうこともあり得るモデルになっています。

計算には、$\sigma = 1, m = 1, \epsilon = 1$ の単位系を用います。

入力すべきパラメタは

- N: ボンドの数 N（ビーズの数 $= N + 1$）
- R: 末端間距離 R
- T: 温度 T
- gamma: 摩擦係数 ζ/m
- Kfene: FENE ポテンシャルのバネ定数 k
- rmax: FENE ポテンシャルの最大のび r_{max}
- dt: タイムステップ
- maxStep: 総タイムステップ数
- intervalStep: データを出力する間隔

準備したエンジンは、これらの入力パラメタを、以下のような書式のファイルから読み込みます

```
N R
T gamma
Kfene rmax
dt maxStep intervalStep
```

エンジンの実行は

```
% ./udon test.in > test.out
```

で行います。`test.in` が上記の書式の入力ファイルで、計算結果は標準出力に出力されますので、適当なファイルにリダイレクトして下さい（標準エラー出力にも、計算の進行状況を示す簡単な出力があります)。標準出力には、intervalStep 毎に

```
it t F Faverage
x[0] y[0] z[0]
x[1] y[1] z[1]
...
x[N] y[N] z[N]
```

というデータが出力されます。

3.4　Udon シミュレータ用ファイル変換フィルタの作成

では、この Udon シミュレータ用のファイル変換フィルタを Python で作成してみましょう。

3.4.1　UDF ファイルを Udon 入力ファイルに変換

まず、Udon シミュレータの入力パラメタを表す UDF ファイルを定義し、それを Udon の入力ファイルに変換するフィルタを、Python で作成してみます。

入力 UDF の定義は、いろいろな考え方があると思いますが、ここでは以下の定義を用います：

```
//===== python/udon/Udon-1.udf =====
\begin{global_def}
Input: {
    Molecule: {
        N:  int
        FENE: {
            Kfene:  double  "spring constant"
            rmax:   double
        }
    }

    SimulationCondition: {
        R:  double      "end-to-end distance"
        Bath: { // Heat bath
            T:  double      "Temperature"
            Gamma:  double  "friction constant"
        }
        dt: double          "time step"
        MaxStep: int        "total number of steps"
        IntervalStep: int   "output interval"
    }
}
\end{global_def}
// データ部は省略
```

この UDF ファイルを Udon の入力ファイルに変換する Python プログラムは、たとえば次のようになります：

```
#===== python/udon/udf2udon.py =====
fout = open("/path/to/test.in","w")
```

```
print >> fout, $Input.Molecule.N, $SimulationCondition.R
print >> fout, $Input.SimulationCondition.Bath.T, \
               $Input.SimulationCondition.Bath.Gamma
print >> fout, $Input.Molecule.FENE.Kfene, $Input.Molecule.FENE.rmax
print >> fout, $Input.SimulationCondition.dt, \
               $Input.SimulationCondition.MaxStep, \
               $Input.SimulationCondition.IntervalStep


fout.close()
```

出力用のファイルの作成には、Python が標準で持つ関数 `open()` を使います。`open()` は作成したファイルオブジェクトを返しますので、それを `fout` に保存しています。ここでは、変換後のファイルの出力先ディレクトリと出力ファイル名を直接プログラム中に記述していますので、適当に変更してから実行して下さい。Windows の場合も、パス名の区切りは `\` ではなく `/` を使って下さい（たとえば `open("c:/tmp/test.in","w")` など）。

`print` 文で `>> fout` を指定することで、ファイル `fout` に出力することが出来ます。

3.4.2 単位の指定

これで UDF から入力ファイルへの変換は可能になったのですが、せっかく UDF を使うのですから、単位の指定を追加してみます。

```
//===== python/udon/Udon-2.udf =====
\begin{unit}
[kB]    = 1.38e-23 [J/K]
[amu]   = 1.66e-27 [kg]
[sigma] = {$Input.Unit.sigma} [nm]
[mass]  = {$Input.Unit.mass}    [amu]
[epsilon] = {$Input.Unit.epsilon} [kB*K]
[t0]    = [sigma*mass^(1/2)*epsilon^(-1/2)]
\end{unit}

\begin{global_def}
Input: {
    Molecule: {
        N:  int
        FENE: {
            Kfene:  double [epsilon/sigma^2]    "spring constant"
            rmax:   double [sigma]
```

```
        }
    }

    SimulationCondition: {
        R:  double [sigma]      "end-to-end distance"
        Bath: { // Heat bath
            T:  double [epsilon/kB]     "Temperature"
            Gamma:  double [1/t0]       "friction constant"
        }
        dt: double [t0]     "time step"
        MaxStep: int        "total number of steps"
        IntervalStep: int   "output interval"
    }

    Unit: {
        sigma:  double [nm]
        mass:   double [amu]
        epsilon: double [kB*K]
    }
}

\end{global_def}
//===== data =====
\begin{data}
Input: {
    Molecule {
        N  20
        FENE { 15.0  1.5 }
    }
    SimulationCondition {
        R 5.0
        Bath { 1.0 0.5 }
        dt 0.005, MaxStep 20000, IntervalStep 200
    }
    Unit { sigma 2.0, mass 50.0, epsilon 300 }
}
```

Udon シミュレータでは、長さ、質量、エネルギーの単位として、σ, m, ϵ を用いますが、この 3 つのパラメタの実際の値を指定するために、Unit という構造体を定義しています。そして、先頭の unit 定

義部で、UDF 内で用いる単位を定義しています。

この Udon-2.udf を Gourmet で開き、単位変換の機能を試してみてください。

3.4.3 Udon の出力ファイルを UDF に変換する

次に、Udon の出力ファイル（test.out）を UDF に変換してみます。もしまだ Udon を実行していなければ、変換されて出来た test.in を用いて実際に Udon を実行し、出力を test.out に保存して下さい：

```
% ./udon test.in > test.out
```

まず出力用の UDF 定義は

```
//===== python/udon/Udon-3.udf =====
// unit と global_def は Udon-2.udf と同じ
//
\begin{def}
class Vector: {
    x: double [unit]
    y: double [unit]
    z: double [unit]
} [unit]

Output: {
    Iteration: int
    Tension:    double [epsilon/sigma]  "tension"
    Taverage:   double [epsilon/sigma]  "batch average of tension"
    Position[]: Vector [sigma]  "positions of beads"
}
\end{def}
```

を用います。計算結果には、一定のタイムステップ毎に同じデータが出力されますので、当然それぞれをレコードにまとめることになります。したがって、global_def ではなく def を用いて定義しました。また、3 次元のベクトルを表す Vector というクラスを定義し、Position[] は Vector の配列としています。Vector の定義での単位の指定法にご注意下さい。

では、Udon-3.udf を Gourmet で開いて下さい。Udon-3.udf には、Udon-2.udf と同じ入力パラメタが設定済みです。変換後の UDF ファイルには、計算条件も書かれていた方がわかりやすいので、入力 UDF ファイル（あるいはそのコピー）を Gourmet で開き、そこに計算結果を追加することにします。

```
# udon2udf.py:  Udon output ==> UDF file
fin = open('/path/to/test.out','r')

N = $Input.Molecule.N
```

```
eraseRecord(0)    # 既存のレコードを全て消去
while 1:
    s = fin.readline().split()
    if not s:
        break
    newRecord()
    $Output.Iteration = int(s[0])
    $Output.Tension = float(s[2])
    $Output.Taverage = float(s[3])
    pos = []
    for i in range(0,N+1):
        s = fin.readline().split()
        pos.append( [float(s[0]), float(s[1]), float(s[2])] )

    $Output.Position[] = pos

fin.close()
jump(0)          # 先頭のレコードに戻っておく
print "end"
```

まず Udon の出力ファイル test.out を `open()` します。次に、ボンドの数 `N` を `Input.Molecule.N` から読んで置きます。その後の `eraseRecord(0)` で既存のレコード（もしあれば）を全て消去します。

`while` ループの先頭の

```
s = fin.readline().split()
```

は

```
tmp = fin.readline()
s = tmp.split()
```

を中間変数 `tmp` を使わずにコンパクトに書いたものです。`readline()` は `fin` から一行読み込み、結果を文字列で返します。この文字列に対して `split()` メソッドを用いると、その文字列を空白（スペース、タブ、改行）を区切り文字として分割した単語のリストが返されます。Udon の出力ファイルは、一定のタイムステップ毎に

```
it t F Faverage
x[0] y[0] z[0]
x[1] y[1] z[1]
...
x[N] y[N] z[N]
```

でしたから、最初に読む行は

```
it t F Faverage
```

です。したがって、例えば readline() の結果は "0 0.0 1.2345 0.9876" という文字列になり、split() の結果 s は

```
[ "0", "0.0", "1.2345", "0.9876" ]
```

というリストになります。

ファイルの最後に到達すると、readline() の結果は空文字列になり、それを split() すると空のリストになります。空のリストは論理値としては偽になりますので、

```
s = fin.readline().split()
if not s:
    break
```

の部分の break が実行されることにより while ループから抜けます。

ファイルの最後に到達していない場合は、newRecord() により新しいレコードを作成します。次に

```
$Output.Tension = float(s[2])
```

等で、s の要素を実数（float）あるいは整数（int）に変換し、UDF 変数に代入しています。その後、変数 pos を空リストで初期化し、for ループで fin から粒子の座標を一行づつ読みながら、実数に変換し、x, y, z をまとめてリストにし、pos に追加しています。つまり、pos はリストのリストになります。N+1 個の粒子の座標を全て pos に追加したら、全体を UDF ファイルの Output.Position にコピーします。Position は「Vector の配列」ですが、Python から値を設定するときは、「リストのリスト」を使えばよいことを理解して下さい。

Output.Position の値を設定する for ループは、Python 変数 pos を用いず、直接ループ内で UDF 配列に値を設定するのでもかまいません：

```
for i in range(0,N+1):
    s = fin.readline().split()
    $Output.Position[i] = [float(s[0]), float(s[1]), float(s[2]) ]
```

3.4.4 アニメーション表示

さて、計算結果を UDF に変換したら、当然 Gourmet の機能を使って分子鎖の運動をアニメーション表示したくなるでしょう。

Gourmet でアニメーションを行うには、単に 1 つのレコードのデータからアニメーションの 1 フレームを描画する Python プログラムを作成するだけで済みます。そのプログラムは、たとえば以下のようになるでしょう：

```
#===== python/udon/show.py =====
atomAttr = 3
bondAttr = 2
```

```
p1 = $Output.Position[0]
sphere(p1,3)
for i in range(1,$Input.Molecule.N+1):
    p2 = $Output.Position[i]
    sphere(p2,atomAttr)
    cylinder(p1,p2,bondAttr)
    p1 = p2
```

まずは show.py をロードして実行し、アニメーション表示を行ってみて下さい。

では描画プログラムについて簡単に説明しておきましょう。

atomAttr と bondAttr は粒子とボンドの描画に用いる属性です。ここでは整数値 3 と 2 を用いていますので、

```
$PF_FILES/lib/ShapeDrawingAttr.txt
```

に指定されている属性を用います。

$Output.Position[0] は、粒子 0 の x, y, z 座標からなるリストになります。このリストは、そのまま描画関数 sphere() や cylinder() に与えることが出来ます。まず粒子 0 を描画した後、for ループで N 個の粒子とボンドを描画します。ボンドは直線ではなく cylinder() を使っています。

演習：atomAttr と bondAttr を好みに応じて設定してみて下さい。

3.4.5　アクションにまとめる

以上で Python によるファイル変換と描画については説明を終ります。このほか、色々な解析が Python で出来ると思いますので、試してみて下さい。

この節では、これまでに作った Python プログラムを、アクションメニューから呼び出して実行出来るようにします。

アクションファイル udon.act には、これまで作った 3 つの Python プログラムを記述します：

```
#===== action/udon.act =====

#----- create Udon input file -----
action Input: Create_Udon_Input(outputFilePath="[...]"): \begin
fout = open(r'outputFilePath'.strip('"'),"w")  # 後で説明します

print >> fout, $Input.Molecule.N, $Input.SimulationCondition.R
print >> fout, $Input.SimulationCondition.Bath.T, \
    $Input.SimulationCondition.Bath.Gamma
print >> fout, $Input.Molecule.FENE.Kfene, $Input.Molecule.FENE.rmax
print >> fout, $Input.SimulationCondition.dt, \
    $Input.SimulationCondition.MaxStep, \
```

```
    $Input.SimulationCondition.IntervalStep

fout.close()
\end

#----- read Udon output file -----
action Output: Read_Udon_Output(inputFilePath="[...]"): \begin
fin = open(r'inputFilePath'.strip('"'),"r")

N = $Input.Molecule.N
eraseRecord(0)
while 1:
    s = fin.readline().split()
    if not s:
        break
    newRecord()
    $Output.Iteration = int(s[0])
    $Output.Tension = float(s[2])
    $Output.Taverage = float(s[3])
    pos = []
    for i in range(0,N+1):
        s = fin.readline().split()
        pos.append( [float(s[0]), float(s[1]), float(s[2])] )

    $Output.Position[] = pos

fin.close()
jump(0)
print "Read_Udon_Output: end"
\end

#----- show animation -----
action Output: Animation(): \begin
atomAttr = 3
bondAttr = 2
p1 = $Output.Position[0]
sphere(p1,3)
for i in range(1,$Input.Molecule.N+1):
    p2 = $Output.Position[i]
```

```
        sphere(p2,atomAttr)
        cylinder(p1,p2,bondAttr)
        p1 = p2
    \end
    #===== udon.act: end =====
```

ファイルを open() するところが複雑ですが、説明は後ですることにして、とりあえずこのアクションを UDF から利用出来るようにするため、UDF ファイルの先頭に以下の様なヘッダを追加します：

```
//===== action/Udon.udf =====
\begin{header}
\begin{def}
    Action: string;
\end{def}
\begin{data}
    Action: "udon.act"
\end{data}
\end{header}
// 以下 Udon-3.udf と同じ
```

この UDF ファイルを用いて Udon シミュレータを使う手順は、以下のようになります：

1. Udon.udf を Gourmet で開き、Input の内容を適当に設定する。
2. Gourmet の File... メニューの Save As... で別の名前のファイルに保存。
3. Input を右クリックしてアクションメニューを呼び出し、Create_Udon_Input を実行。
4. ダイアログボックスの outputFilePath の入力欄（Values 欄）を右クリック。
5. ファイル選択ダイアログが現れるので、Udon 入力ファイルを保存したいディレクトリに移動し、FileName: にファイル名を入力。Open ボタンを押す。
6. アクションのダイアログに戻るので、OK を押す。Udon 入力ファイルが作成される。
7. 作成された入力ファイルを用いて Udon を実行し、Udon 出力ファイルを作成。
8. Gourmet 上で Output を右クリックし、 Read_Udon_Output アクションを実行。
9. 先ほどと同様の手順で inputFilePath に Udon 出力ファイルを指定して、OK を押す。Udon 出力ファイルの内容が UDF に読み込まれる。
10. Output を右クリックし、 Animation アクションを実行。

このように、Python によるファイル変換とアクションを組み合わせると、既存のエンジンの入力ファイルの作成や計算結果の解析を、とても簡単に行うことが出来ます。ぜひ活用して下さい。

さて、udon.act の内容ですが、ほとんど udf2udon.py, udf2udon.py, show.py と同じです。ここでは Udon の入出力ファイルのパス名を取得し、そのファイルを open する部分を説明しますが、少し込み入った話になりますので、細部に立ち入りたくない方は、読み飛ばされても結構です。

まず、

```
action Input: Create_Udon_Input(outputFilePath="[...]"): \begin
```

の部分に注目して下さい。outputFilePath というアクション引数のデフォルト値として "[...]" という文字列が指定されています。この場合、アクション引数の値として、ファイルのパス名をダイアログ経由で取得することが可能になります。具体的には、アクション起動時の引数入力ダイアログで、該当する引数の入力欄（Values の欄）を右クリックするとファイル選択ダイアログが表示されるので、そこでファイルを選択（あるいはディレクトリを選択してファイル名を入力）することが出来ます（もちろん、引数の入力欄に直接パス名をキーボードから入力することも出来ます）。

このようにしてファイル選択ダイアログからファイルのパス名を取得できるのですが、Windows の場合、パス名の区切りが \ であるため、すこし厄介です。たとえば outputFilePath が"C:\tmp\test.in" の場合、2 回現れる \t が Python により TAB 文字のエスケープシークエンスと解釈されてしまいます（"\tmp" は TAB, m, p の 3 文字からなる文字列）。

この問題は r"C:\tmp\test.in" の様に文字列の先頭に r を付ければ回避出来ます。r"\tmp" は\, t, m, p の 4 文字からなる文字列です。このように先頭に r が付いた文字列を raw string と呼びます。

アクションの引数 outputFilePath をこの raw string にする仕組みが

```
fout = open(r'outputFilePath'.strip('"'),"w")
```

の部分です。これを理解するには、アクション引数がアクション本体の Python スクリプトに渡される仕組みを理解する必要があります。

たとえばファイル選択ダイアログで C:\tmp\test.in を選択した場合、

```
outputFilePath = "C:\tmp\test.in"
```

という代入文が実行されるわけではありません（これだと outputFilePath が TAB 文字を含んでしまいます）。実際には、スクリプト本体中に現れる outputFilePath という「単語」が、全て "C:\tmp\test.in" で置き換えられます。この置き換えは、outputFilePath がスクリプト中で引用符で囲まれていても行われます。従って、

```
fout = open(r'outputFilePath'.strip('"'),"w")
```

は Gourmet によりまず

```
fout = open(r'"C:\tmp\test.in"'.strip('"'),"w")
```

に置き換えられ、その後 Python インタープリタにより実行されます。

r'"C:\tmp\test.in"' という raw string は、両端の 2 重引用符 " を含め、16 文字からなる文字列です。両端の 2 重引用符 " はパス名から除く必要がありますので、strip('"') を用いています。strip(s) は、文字列の両端から、文字列 s に含まれる文字を取り去る、というメソッドです。文字列リテラル（'foo' や r'\tmp'）も立派な string オブジェクトなので、string のメソッドを呼び出すことが出来ます。

演習：Tension（あるいは Taverage）の時間変化のグラフを gnuplot で表示するアクションを追加してみて下さい。

演習：張力の平均値と標準偏差を計算する Python プログラム（あるいは action）を作成してみて下さい。

Python を用いた UDF 操作については以上で終了です。次章以降では、C 言語からの UDF の入出力について説明します。

3.5　C 言語での簡単な例題

3.5.1　UDF ファイル

まずおおまかな流れを理解するため、以前も用いた以下の様な単純な UDF ファイルを考えます。

```
//===== C/simple/simple.udf ===== （再掲）
\begin{global_def}
Temperature: double
Atom: {
    Name: string
    AtomicNumber: int
    AtomicWeight: double
    Pos: {
        x: double
        y: double
    }
}
\end{global_def}

\begin{data}
Temperature: 300.0
Atom: { "Carbon" 6 12.0 {1.0 2.0} }
\end{data}
```

3.5.2　C プログラム

この UDF ファイルからデータを読み込む C プログラムは、以下のようになります：

```
1   /*===== C/src/simple-in.c =====*/
2   #include <stdio.h>
3   #include "fc_interfaceC.h"
4
5   #define MAXNAME 256
6
```

```
 7  int main()
 8  {
 9      double T;
10      char name[MAXNAME];
11      int atomicNumber;
12      double atomicWeight;
13      int inUDF;
14      int status;
15
16      status = openUDFManager("simple.udf", &inUDF);
17
18      status = getDoubleValue(inUDF, "Temperature", &T);
19      status = getIntValue(inUDF, "Atom.AtomicNumber", &atomicNumber);
20      status = getDoubleValue(inUDF, "Atom.AtomicWeight", &atomicWeight);
21      status = getStringValue(inUDF, "Atom.Name", MAXNAME, name);
22
23      printf("Temperature = %f\n", T);
24      printf("Atom: Name = %s,  AtomicNumver = %d,  AtomicWeight = %f\n",
25              name, atomicNumber, atomicWeight);
26
27      status = closeUDFManager(inUDF);
28      return 0;
29  }
```

全体の流れは、図 3.2 にまとめてあります。

UDF を読み書きするプログラムを C 言語で作成する場合、必要な関数は `fc_interfaceC.h` というヘッダファイルに宣言されています。3 行目でこのヘッダファイルをインクルードしています。

16 行目で入力元の UDF ファイルを開きます。そこで用いている

```
int openUDFManager(char *file_name, int *udf_handle)
```

は、第 1 引数 `file_name` で指定した UDF ファイルを開き、開かれた UDF ファイルへの「UDF ハンドル」を `udf_handle` に返します。open が成功すると戻り値は `0`、失敗すれば`-1` になります。上の例では見やすくするためエラーチェックを省略していますが、実際はもちろん `status` が 0 かどうかをチェックする必要があります。

「UDF ハンドル」とは、現在開かれている UDF ファイル毎にライブラリにより割り当てられる整数の値です。UDF ファイルを読み書きする関数には、常にこの UDF ハンドルを指定することで、読み書きの対象となる UDF ファイルを指定します。

具体的にデータを入力しているのが 18〜21 行目です。データの読み込みには、`get???Value()` という関数を使います。ここで `???` は読み込みたい値の型に応じて変化します。たとえば、整数値、実数値

```
/* C から UDF を読む */

#include "fc_interfaceC.h"

.....

    int inUDF;
    int intVar;

    /* UDF ファイルを開く */
    status = openUDFManager("fname.udf",&inUDF);

    /* get???Value() を用いてデータを読む */
    status = getIntValue(inUDF,"udf.path",&intVar);
    /* これで、udf.path の値が intVar に読み込まれる */

    /* UDF ファイルを閉じる */
    status = closeUDFManager(inUDF);
```

図 3.2: C から UDF ファイルを読む手順

の読み込みは、getIntValue(), getDoubleValue() を用います。

```
int getDoubleValue(int udf_handle, char *udfpath, double *value)
```

udf_handle は openUDFManager() で得られた UDF ハンドルです。udfpath は読み込みたい UDF 変数の「UDF パス名」を指定し、value には読み込んだ値を格納する変数へのポインタを指定します。これらの関数は全て成功すれば 0、失敗すれば -1 を返します。

文字列の読み込みの場合は、21 行目のように、文字列の格納先の配列 name だけでなく、そこに確保されているメモリのサイズ（MAXNAME）も指定する必要があります。

入力が終れば、27 行のように、UDF ファイルを閉じます。

3.5.3 コンパイル・リンク

開発環境

さて、この C プログラムをコンパイルして実行するのに必要な環境について、簡単にまとめておきます。

まず、C 言語でプログラムを作るには、当然コンパイラが必要です。Windows の場合、Cygwin を gcc/g++ を含めてインストールすることをお勧めします。C 言語での開発であっても、UDF 入出力のためには、gcc だけでなく g++ まで必要になる点にご注意下さい。また、make コマンドも同時にインストールされることを強くお勧めします。

Linux では、通常 gcc/g++/make は既にインストールされていると思いますが、無ければ追加でインストールして下さい。

MacOS X の場合、Apple のサイトから最新の Xcode をダウンロードしてインストールすると、gcc/g++/make もそれに含まれます。

OCTA と環境変数

次に、当然ですが OCTA がインストールされている必要があります。インストーラを使ってインストールした場合、自動的に環境変数 $PF_FILES が

インストールした OCTA2005 ディレクトリ内の GOURMET_2005 というディレクトリ

を指すように、設定されているはずです。たとえば Windows で C: ドライブの最上位階層にインストールしたなら、$PF_FILES は

```
C:\OCTA2005\GOURMET_2005
```

になっているはずです。設定されているかどうかは、シェル（bash, zsh, tcsh, etc.）を開き、

```
% env
```

と入力し（% はシェルのプロンプト）、出力の中に PF_FILES があるかどうかで確認できます。設定されていない場合は、シェルの設定ファイルなどに PF_FILES の設定を追加するなどして下さい。

$PF_FILES 以外にも、コンパイル・リンクをスムーズに進めるために設定されていた方がよい環境変数がありますが、これらを一括して設定してくれるスクリプトが $PF_FILES/bin/ に pfsetenv.sh などの名前で準備されています。お使いのシェルから

```
% source $PF_FILES/bin/pfsetenv.sh
```

とすると、必要な環境変数が全て設定されます。(bash/zsh なら上記でよい。tcsh の場合は pfsetenv.csh を使う)。Windows のコマンドプロンプト（cmd.exe）を使う場合は、

```
C:\> $PF_FILES/bin/pfsetenv.bat
```

となります。

ヘッダファイルとライブラリ

インクルードするヘッダファイル fc_interfaceC.h は $PF_FILES/include にあります。これはコンパイル時に必要です。

一方、リンク時に必要なライブラリですが、C++/C/Fortran から UDF を読み書きするための関数群は、libplatform.a というライブラリに含まれます。Windows の場合、MinGW 用のライブラリが

```
$PF_FILES/lib/win32/libplatform.a
```

にインストールされているはずですが、自分でライブラリを作成し直した方が確実かもしれません。通常の（MinGW でない）Cygwin 環境や Linux/MacOS X の場合も、ご自分でライブラリを作成して頂くことになります。しかし、その手順は単純ですので、ご安心下さい。具体的には、以下の手順でライブラリが作成されます。

まず、環境変数 PF_FILES が正しく設定されていることを確認します。次に、Gourmet のソースファイル一式を解凍します。Windows ならば

```
C:\> cd $PF_FILES
C:\OCTA2005\GOURMET_2005> GOURMET2005_SRC.EXE
```

Linux/MacOS X ならば

```
% cd $PF_FILES
% tar zxvf gourmet2005_src.tar.gz
```

その後 src ディレクトリに移動して、make します：

```
% cd src
% make
% make install
```

これで、libplatform.a が作成され、インストールされます。インストールされる先は、

```
$PF_FILES/lib/cygwin/libplatform.a
```

（あるいは、cygwin の替りに linux, macosx としたもの）です。

コンパイル・リンクの実行

さて、以上の準備が整えば、実際に先ほどの C プロフラムをコンパイルして実行してみることが出来ます。

```
% gcc -c simple-in.c -I$PF_FILES/include
% g++ -o simple-in simple-in.o -L$PF_FILES/lib/cygwin -lplatform
% ./test-1
```

C のソースファイルは 1 つだけですので、コンパイルとリンクを同時に行うことも出来ます

```
% g++ -o simple-in simple-in.c -I$PF_FILES/include
                      -L$PF_FILES/lib/cygwin -lplatform
```

（実際は 1 行で入力して下さい）。C 言語を使用していても

リンクの際は gcc ではなく g++ を使う必要がある

ことにご注意下さい。

実際は、上記のような長いコマンドを入力するのは煩雑で間違いも多くなるので、通常は make コマンドを用います。最小限の Makefile は

```
CFLAGS = -I$(PF_FILES)/include
LDFLAGS = -L$(PF_FILES)/lib/$(PF_ARCH) -lplatform
simple-in: simple-in.c
    $(CXX) -o $@ simple-in.c $(CLAGS) $(LDFLAGS)
```

となります。ここで PF_ARCH は pfsetenv.sh によって設定される環境変数で、環境に応じて win32, cygwin, linux, macosx などの値を取ります。

Windows (MinGW) 用に作成済みの実行ファイルが C/simple/simple-in.exe にありますので、実行してみて下さい。

3.6 C 言語からの UDF 入出力

これでおおまかな流れはつかんで頂けたと思いますので、C 言語からの UDF の入出力についてもう少し詳しく説明していきます。

3.6.1 配列の入力

配列データの UDF からの入力を説明します。例えば、Python からの入力の例でも用いた

```
//===== C/array/array.udf =====
\begin{global_def}
a[]: double
aa[][]: double
b[]: {
    x: double
    y: double
}
\end{global_def}

\begin{data}
a[]: [ 0.0 1.1 2.2 3.3 ]
aa[][]: [ [0.0 0.1 0.2 ] [1.0 1.1 1.2 ]  ]
b[]: [ { 0.0 0.1 } { 1.0 1.1 } { 2.0 2.1 } ]
\end{data}
```

からデータを読むには

```
1   /*===== C/src/array-in.c =====*/
2   #include <stdio.h>
3   #include "fc_interfaceC.h"
4
5   #define BUF_SIZE 1000
6
7   int main()
8   {
9       int inUDF;
10      int status;
11      double result;
12      double buf[BUF_SIZE];
13      int dimension;
14      int size_list[10];  /* max 10 dimension */
15      int i, j, k;
16
17      status = openUDFManager("array.udf", &inUDF);
18
19      getDoubleValue(inUDF,"a[3]", &result);
20      printf("a[3] = %f\n", result);
21
22      getDoubleValue(inUDF, "aa[1][2]", &result);
```

```
23      printf("aa[1][2] = %f\n", result);
24
25      getDoubleValue(inUDF, "b[2].y", &result);
26      printf("b[2].y = %f\n", result);
27
28      getDoubleArray(inUDF, "a[]", BUF_SIZE, &dimension,
29                                      size_list, buf);
30      printf("dimension of a = %d,  size of a = %d\n",
31                          dimension, size_list[0]);
32      for(i=0; i<size_list[0]; ++i) {
33          printf("a[%d] = %f\n", i, buf[i]);
34      }
35
36      getDoubleArray(inUDF, "aa[][]", BUF_SIZE, &dimension,
37                                      size_list, buf);
38      printf("dimension of aa = %d,  size of aa = %d %d\n",
39                          dimension, size_list[0], size_list[1]);
40      k = 0;
41      for(i=0; i<size_list[0]; ++i) {
42          for(j=0; j<size_list[1]; ++j) {
43              printf("aa[%d][%d] = %f\n", i, j, buf[k++]);
44          }
45      }
46
47      closeUDFManager(inUDF);
48      return 0;
49  }
```

とします（見やすくするため、エラーチェックを省略しています）。

特定の配列要素の値を得るためには、19, 22, 25 行などのように、getDoubleValue() などが使えます。

配列全体を読み込むためには、28, 36 行のように getDoubleArray() などを用います：

```
int getDoubleArray(int udf_handle, char *udfpath, int capacity,
            int *num_dim, int size_list[], double *values)
```

ここで udfpath は読み込みたい配列の UDF Path で、配列指定 [] を含みます。読み込んだ値は C の配列 values に返されます。capacity には values に格納出来る最大の要素数を与えます。num_dim には、読み込んだ UDF 配列の次元が返されます。（udfpath が 1 組の [] を含めば num_dim = 1、[][] を含めば num_dim = 2 などとなります。つまり getDoubleArray() を呼ぶ前に次元はわかっている

ことになるので、この情報は冗長です)。size_list[] は 1 次元の整数の配列で、たとえば指定した udfpath のデータが 4x5 の 2 次元配列だった場合、size_list[0] = 4, size_list[1] = 5 となります(size_list[] には次元以上の個数の要素が確保されていなければなりません)。また、C の配列 values[] には、2 次元の UDF 配列の値が、[0][0], [0][1], [0][2], ..., [1][0], [1][1], の順に記憶されます。41 から 45 行のループをよく理解して下さい。

3.6.2 UDF ファイルの作成と出力

次に、C プログラムから UDF ファイルへ出力する方法を説明します。

```
 1  /*===== C/src/simple-out.c =====*/
 2  #include "fc_interfaceC.h"
 3
 4  int main()
 5  {
 6      int outUDF;
 7      int status;
 8      double Temperature = 345.0;
 9      char Name[] = "Chlorine";
10      int AtomicNumber = 17;
11      double AtomicWeight = 35.5;
12      int dummy;
13
14      status = createUDFManager("simple-out.udf", "simple-def.udf",
15                                1, &outUDF);
16
17      status = setDoubleValue(outUDF, "Temperature", &Temperature);
18      status = setDoubleValue(outUDF, "Atom.AtomicWeight", &AtomicWeight);
19      status = setIntValue(outUDF, "Atom.AtomicNumber", &AtomicNumber);
20      status = setStringValue(outUDF, "Atom.Name", dummy, Name);
21
22      status = writeUDFData(outUDF);   // これを忘れないように！
23
24      closeUDFManager(outUDF);
25      return 0;
26  }
```

出力用の UDF ファイルを作成するには、CreateUDFManager() を用います：

```
int CreateUDFManager(char *out_fname, char *def_fname,
                     int copy_flag, int* udf_handle)
```

out_fname に作成したい UDF ファイル名を指定します。def_fname は、作成するのに用いる UDF 定義を含む UDF ファイルの名前を指定します。このファイルは、UDF 定義だけでなく、データを含んでいてもかまいません。copy_flag は 0 または 1 を指定します。1 を指定すると、def_fname で指定したファイルの内容が、作成されるファイル out_fname にコピーされます。一方 0 を指定すると、ファイル out_fname の先頭に、

```
\include{"def_fname"}
```

が記入されます。udf_handle に作成した UDF ファイルを指す整数値が戻されます。関数の戻り値は、成功すれば 0、失敗すれば -1 です。

UDF 変数に値を設定するには setDoubleValue() などの関数を用います：

```
int setDoubleValue(int udf_handle, char *udfpath, double *value)
```

設定する値（value）をポインタで渡していることにご注意下さい。文字列を設定する場合は

```
int setStringValue(int udf_handle, char *udfpath, int dummy, char *value)
```

です。これにより、value で指定した値を、udfpath で指定した UDF 変数に設定します。dummy の値は用いられません。

setDoubleValue() などの関数は、メモリ上の UDF 変数に値を設定するだけで、ファイルへの書込は行いません。実際に UDF ファイルへの書込を行うには、writeUDFData() を用います。出力 UDF ファイルを close する前に、必ずこの関数を呼ぶことを忘れないでください。

3.6.3 配列の出力

配列への出力は以下のようになります：

```
 1  /*===== C/src/array-out.c =====*/
 2  #include <stdio.h>
 3  #include "fc_interfaceC.h"
 4
 5  int main()
 6  {
 7      int outUDF;
 8      int status;
 9
10      double a[] = { 10.0, 11.0, 12.0 };
11      double aa[2][3] = { {10.0, 10.1, 10.2}, {11.0, 11.1, 11.2} };
12      struct _vec {
13          double x;
14          double y;
15      } b[] = { {0.0, -0.1}, {1.0, -1.1}, {2.0, -2.1} };
```

```
16
17      int size_list[2];
18      char udfpath[256];
19      int i;
20
21      status = createUDFManager("array-out.udf", "array-def.udf",
22                                    1, &outUDF);
23
24      size_list[0] = 3;
25      setDoubleArray(outUDF, "a[]", size_list, a);
26
27      size_list[0] = 2;
28      size_list[1] = 3;
29      setDoubleArray(outUDF, "aa[][]", size_list, (double *)aa);
30
31      for(i=0; i<3; ++i) {
32          sprintf(udfpath,"b[%d].x",i);
33          setDoubleValue(outUDF, udfpath, &b[i].x);
34          sprintf(udfpath,"b[%d].y",i);
35          setDoubleValue(outUDF, udfpath, &b[i].y);
36      }
37
38      status = writeUDFData(outUDF);
39
40      closeUDFManager(outUDF);
41      return 0;
42  }
```

UDF 配列に出力するには、setDoubleArray() などの関数を用います：

```
int setDoubleArray(int udf_handle, char *udfpath, int size_list[],
                double *values)
```

出力される配列の次元は udfpath で決まります。たとえば"aa[][]" なら 2 次元です。size_list[] に各次元のサイズを与えます。実際のデータは、C の配列 values に与えます。2 次元（以上）の UDF 配列に値を設定する場合、UDF 配列の要素の値を [0][0], [0][1], [0][2], ..., [1][0], [1][1], の順に並べたものを、C の 1 次元配列 values に準備します。

構造体の配列（b[]）への出力は面倒です。現状では、31 から 36 行のループのように、個々の配列要素の個々の構造体要素について出力先の UDF Path を文字列として作成し、そこに一つづつ出力することになります。

3.6.4 レコードの取り扱い

次に、C からレコードの作成や、レコード間の移動を行う方法を説明します。用いる UDF は 3.2.3 で用いたものと同じ（C/record/record.udf）です。

レコードを含む UDF ファイルからデータを読みには、以下のようにします：

```
 1  /*===== C/src/record-in.c =====*/
 2  #include <stdio.h>
 3  #include "fc_interfaceC.h"
 4
 5  int main()
 6  {
 7      int inUDF;
 8      int status;
 9      int numRec, rec;
10      double T, P;
11
12      openUDFManager("record.udf", &inUDF);
13
14      status = totalRecord(inUDF, &numRec);
15      printf("number of records = %d\n", numRec);
16
17      for(rec=0; rec<numRec; ++rec) {
18          status = jumpRecord(inUDF, rec);
19
20          status = getDoubleValue(inUDF,"P", &P);
21          status = getDoubleValue(inUDF,"T", &T);
22          printf("record %d: P = %f,   T = %f\n", rec, P, T);
23      }
24
25      closeUDFManager(inUDF);
26      return 0;
27  }
```

14 行にあるように、レコードの総数は

```
int totalRecord(int udf_handle, int *numRec)
```

で得られます。レコードの総数は numRec に返されます（関数の戻り値は、いつもの通り成功なら 0、失敗なら -1 です）。

特定のレコードに移動するには、18 行のように

```
int jumpRecord(int udf_handle, int record_number)
```

を用います。20 行で読む "P" はレコード変数で、各レコード毎に異なる値を持ちます。一方 21 行で読む "T" はグローバル変数で、どのレコードに居ても同じ値を持ちます。

レコードの作成等は、以下のように行います：

```
 1  /*===== C/src/record-out.c =====*/
 2  #include <stdio.h>
 3  #include "fc_interfaceC.h"
 4
 5  #define LABELSIZE 100
 6
 7  int main()
 8  {
 9      int outUDF;
10      int status;
11      int numRec = 3;
12      int rec;
13      double T = 300.0;
14      double P;
15      char label[LABELSIZE];
16
17      status = createUDFManager("record-out.udf", "record-def.udf",
18                                1, &outUDF);
19
20      for(rec=0; rec<numRec; ++rec) {
21          status = newRecord(outUDF, LABELSIZE, label);
22          P = rec*rec + 100.0;
23          status = setDoubleValue(outUDF,"P", &P);
24
25          printf("record %d: label = %s\n", rec, label);
26      }
27      status = setDoubleValue(outUDF, "T", &T);
28
29      status = jumpRecord(outUDF, 1);
30      P = 99.0;
31      status = setDoubleValue(outUDF, "P", &P);
32
33      status = writeUDFData(outUDF);
34      closeUDFManager(outUDF);
```

```
35        return 0;
36    }
```

新しいレコードの作成は

```
int newRecord(int udf_handle, int max_label_length, char *label)
```

を用います（21 行)。作成されたレコードの名前が `label` に返されます。`label` が指すメモリ領域は呼び出し側で確保しておく必要があります。`max_label_length` は `label` に確保されている文字数を指定します。

最後はすこし駆け足になりましたが、これで C 言語から UDF ファイルの入出力を行うのに必要な事項は説明したはずです。もしさらに詳しい情報が必要な場合は、「libplatform リファレンスマニュアル」（`libplatform_jpn.pdf`）を参照して下さい。

演習：Udon シミュレータ用のファイル変換フィルタ（UDF から Udon 入力ファイルへ、Udon 出力ファイルから UDF へ）を C 言語で作成してみて下さい。