JAEA-Testing 2016-001
J-PARC 16-02
DOI:10.11484/jaea-testing-2016-001

JAEA-Testing

MLF データ解析フレームワーク " 万葉ライブラリ " 利用マニュアル

The User's Manual of "Manyo Library"
Data Reduction Software Framework at MLF, J-PARC

稲村 泰弘　伊藤 崇芳　鈴木 次郎　中谷 健

Yasuhiro INAMURA, Takayoshi ITO, Jiro SUZUKI and Takeshi NAKATANI

原子力科学研究部門
J-PARC センター
物質・生命科学ディビジョン
Materials and Life Science Division
J-PARC Center
Sector of Nuclear Science Research

June 2016
Japan Atomic Energy Agency
日本原子力研究開発機構

本レポートは国立研究開発法人日本原子力研究開発機構が不定期に発行する成果報告書です。
本レポートの入手並びに著作権利用に関するお問い合わせは、下記あてにお問い合わせ下さい。
なお、本レポートの全文は日本原子力研究開発機構ホームページ（http://www.jaea.go.jp）
より発信されています。

国立研究開発法人日本原子力研究開発機構 研究連携成果展開部 研究成果管理課
〒319-1195 茨城県那珂郡東海村大字白方 2 番地4
電話 029-282-6387, Fax 029-282-5920, E-mail:ird-support@jaea.go.jp

This report is issued irregularly by Japan Atomic Energy Agency.
Inquiries about availability and/or copyright of this report should be addressed to
Institutional Repository Section,
Intellectual Resources Management and R&D Collaboration Department,
Japan Atomic Energy Agency.
2-4 Shirakata, Tokai-mura, Naka-gun, Ibaraki-ken 319-1195 Japan
Tel +81-29-282-6387, Fax +81-29-282-5920, E-mail:ird-support@jaea.go.jp

JAEA-Testing 2016-001
J-PARC 16-02

MLF データ解析フレームワーク"万葉ライブラリ"利用マニュアル

日本原子力研究開発機構 原子力科学研究部門 J-PARC センター

物質・生命科学ディビジョン

稲村　泰弘、伊藤　崇芳*1、鈴木　次郎*2、中谷　健

(2016 年 1 月 23 日　受理)

万葉ライブラリとは、J-PARC の物質・生命科学実験施設（MLF）において実施される中性子散乱実験にて得られたデータの解析に使用されるソフトウェアの開発基盤（フレームワーク）を提供するものである。このフレームワークは MLF で稼働する多くのビームラインで動作する必要があり、中性子実験データの処理に対して共通に使用される機能と各装置の仕様にあわせたソフトウェア開発に使用される。このフレームワークは、様々な次元のヒストグラムを入れる容器、データコンテナを中心として構成されている。データコンテナは、多次元ヒストグラムのエラー伝搬機能付きの四則演算、メタデータの保管、ファイルの読み書きなどの機能を有する。様々な解析用の関数は、このデータコンテナを入出力として扱うことで、万葉ライブラリのソフトウェア上でのデータフォーマットを規定できるだけでなく、演算子を連結することで解析ソフトを作成することができる設計である。一方装置担当者や実験者がこれらの機能を使用したり解析コードを書いたりするためのインターフェースは、実行した結果がすぐに返ってくるインタラクティブな環境が必要である。万葉ライブラリは C++言語で作成しているが、ユーザーインターフェースとして Python を選択し、Python の環境から万葉ライブラリを容易に呼び出す仕組みを導入している。これらの特徴を生かして、すでに様々なデータ処理・解析コードが多数開発されており、現在の MLF の多数の装置において解析作業の要として動作している。本著は万葉ライブラリを初めて使用するユーザーのための利用マニュアルである。

J-PARC センター：〒319-1195　茨城県那珂郡東海村大字白方 2-4

＊1　総合科学研究機構 中性子科学センター（CROSS 東海）

＊2　高エネルギー加速器研究機構 共通基盤研究施設 計算科学センター，J-PARC センター 情報システムセクション

JAEA-Testing 2016-001
J-PARC 16-02

The User's Manual of "Manyo Library" Data Reduction Software Framework at MLF, J-PARC

Yasuhiro INAMURA, Takayoshi ITO*1, Jiro SUZUKI*2 and Takeshi NAKATANI

Materials and Life Science Division, J-PARC Center,

Sector of Nuclear Science Research

Japan Atomic Energy Agency

Tokai-mura, Naka-gun, Ibaraki-ken

(Received January 23, 2016)

Manyo Library is a software framework for developing analysis software of neutron scattering data produced at MLF, J-PARC. This software framework is required to work on many instruments in MLF and to include base functions applied to various scientific purposes at beam lines. This framework mainly consists of data containers, which enable to store 1, 2 and 3 dimensional axes data for neutron scattering. Data containers have many functions to calculate four arithmetic operations with errors distribution between containers, to store the meta-data about measurements and to read or write text file. The analysis codes are constructed using various analysis operators defined in Manyo Library, which executes functions with given data containers and output the results. On the other hands, the main interface for instrument scientists and users must be easy and interactive to treat data containers and functions or to develop new analysis codes. Therefore we chose Python as user interface. Since Manyo Library is built in C++ language, we've introduced the technology to call C++ function from Python environment into the framework. As a result, we have already developed a lot of software for data reduction, analysis and visualization, which are utilized widely in beam lines at MLF. This document is the manual for the beginner to touch this framework.

Keywords: Neutron Scattering, Data Reduction, Data Analysis, Framework Software

*1 Neutron Science and Technology Center, Comprehensive Research Organization for Science and Society

*2 Computing Research Center, Applied Research Laboratory, High Energy Accelerator Research Organization, Information System Section, J-PARC Center

目 次

Contents

図表リスト

This is a blank page.

1. はじめに

1.1. 万葉ライブラリとは

万葉ライブラリとは、J-PARC の物質・生命科学実験施設（MLF）において実施される中性子散乱実験にて得られたデータの解析に使用されるソフトウェアの開発基盤（フレームワーク）を提供するものである [1)]。

かつての中性子や放射光などの量子ビームを用いた大規模な散乱実験装置では、研究分野、目的、実験手法や装置の特性に応じた装置制御と解析ソフトウェアが研究開発され共同利用実験に供されてきた。これらのソフトウェアの開発は装置ごとに行われてきたために、研究施設内での実験パラメータやデータの共有が難しいだけでなく、ソフトウェアの長期にわたる維持管理が組織的に行えない問題点が以前から指摘されてきた。

そのような観点から MLF における中性子の実験装置関連の計算機環境は、J-PARC センターの MLF 計算環境検討グループによって統合的に研究開発が行われ、装置間の計算機環境の標準化の努力がなされてきた。検討範囲は、データ収集、データ解析、セキュリティ、シミュレーション、データ可視化など MLF での共同利用実験で必要な環境すべてである。

そのような状況下で、本書で述べる万葉ライブラリはデータ解析環境の中核となるソフトウェアとして開発されてきた。

1.2. 主な特徴

先に述べたように万葉ライブラリは、MLF における中性子実験データの処理に対して共通に使用される機能と各装置の仕様にあわせたソフトウェアの開発基盤を提供することが求められる。そこで採用されたのが、多数の開発者（研究者）が並列作業で機能追加をし、統一的な開発方針を策定することが可能となるオブジェクト指向型のフレームワークである。

オブジェクト指向の利点の一例をあげてみる。例えばデータファイルを開く場合、フレームワークでは ReadData(string FileName)の関数を用意し標準的な読み方を実装しておく。一方で分光器によって機能不足の場合にはこの関数を上書きで変更することで、利用者は同じ関数名でその機能を利用できる。このように、オブジェクト指向では、施設標準の動作に加えて、分光器や研究者に応じた機能を組み込むことが容易である。この例のような利点があるために、オブジェクト指向に基づいた基盤ソフトウェアを整備することで、解析ソフトウェアの開発が少ない人的資源で迅速に行え、維持管理が長期にわたって容易になることが期待でき、施設全体のソフトウェア環境の信頼性向上が望めるのである。

実験データの読み込み、ヒストグラム化、データ解析の各種演算子などのパッケージは、オブジェクト指向プログラミング言語である C++言語のフレームワークとして開発されている。そして、個々の実験装置で別途必要な解析機能は、C++のモジュールとして追加することで拡張ができる。更に、C++の対話的利用(キャラクターベース、グラフィカル)は、スクリプト言語である Python を用いて開発が進められている。このようにして C++の高い機能と速度性能を、ユーザーが扱いやすい環境で利用可能としている。この解析環境フレームワークは、C++だけのバッチモードと、Python を使ったインタラクティブモードを有効に使い分けできる特徴を有している。

万葉ライブラリのフレームワークは、様々な次元のヒストグラムを入れる容器、データコンテナを中心として構成されている。多次元ヒストグラムのエラー伝搬機能付きの四則演算、イベントデータのヒストグラム化、データコンテナのファイルの読み書きなどがその主な機能である。様々な解析演算子は、このデータコンテナを入出力として扱うことで、万葉ライブラリ

のソフトウェア上でのデータフォーマットを規定できるだけでなく、演算子を連結することで解析ソフトを作成することができる設計である。

これらの特徴を生かして、様々なデータ処理・解析コードがすでに多数開発されている。現在の MLF に設置された多数の装置において、この万葉ライブラリやこれを用いた「空蝉」[2),3)]「絵巻」[4),5)]といったデータ処理パッケージが解析の要として動作しており、MLF における必須の技術として広く使用され、日々の成果創出の基盤となっている。

1.3. 開発体制

1.3.1. 開発初期

万葉ライブラリは 2002 年夏頃に MLF における中性子データ解析環境の整備を目的に開始された。開発は大友季哉氏（高エネルギー加速器研究機構(KEK)）、古坂道弘氏(現北海道大学)をアドバイザーに、鈴木次郎(当時 KEK 計算科学センター博士研究員)で KEK つくばにおいて開始された。

1.3.2. 現状

2015 年現在、開発メンバーとして、鈴木（KEK 計算科学センター, J-PARC Center）、中谷健、稲村泰弘（日本原子力研究開発機構(JAEA), J-PARC センター）、伊藤崇芳（一般財団法人 総合科学研究機構 中性子科学センター(CROSS)）らをコアとして、KEK, JAEA, CROSS の J-PARC センター員により構成される MLF 計算環境グループが開発を進めている。

1.4. 使用言語の選択

万葉ライブラリ利用者が直接触れることになるユーザーインターフェースとしてのプログラミング言語には、スクリプト言語であることから柔軟に利用でき、オブジェクト指向言語であり、また、将来性のある言語として Python が選択された。

万葉ライブラリの基礎部分を C++のクラスライブラリとして、整備することにした。これは、Python だけでは処理速度が遅いこと、C 言語によるライブラリの開発経験者が多いこと、並列化や分散処理に対応する予定であることなどによる。

C++クラスライブラリを Python から利用するためのインターフェース(ラッパ)は手作業で作成することは可能だが、自動化をするために SWIG を使用している。

1.5. 名前の由来

「万葉ライブラリ」の名前は、

1. 日本発祥のフレームワークとして **和名** が適しており、
2. 「万葉」とは「万葉集」「万葉仮名」などで知られているように「万（よろず）の言の葉（ことのは)」であり、 **様々な研究分野の研究者が様々なソフトウェアを結びつけ協調する解析環境** を示すにふさわしく、
3. 「万葉集」が千年以上伝えられてきたように **末長く使用されたい** という願いをこめて、

名付けられた。

2. データコンテナ

2.1. データコンテナの概要

万葉ライブラリでは独自のデータコンテナを通してデータ処理・解析に便利な機能を提供している。主な機能として次のものがある。

・データ列への名称付け
・誤差伝搬、マスクを考慮したデータコンテナの四則演算
・様々なデータ補正に向けた測定条件などの情報の付与
・データのファイル読み込み、書き出し

このデータコンテナを ElementContainer と呼ぶ。さらにこの ElementContainer を階層的に扱うために、ElementContainerArray、ElementContainerMatrix と呼ぶデータコンテナがある。この章ではこれらのデータコンテナの構造、規則、使用方法を述べる。

なお記述されているスクリプトは、すでに万葉ライブラリがインストールされていることを前提としている。すなわち

```
>>> import Manyo
Manyo >>>
```

が実行できているとする。

2.2. ElementContainer

ElementContainer は一つの１次元データを収めるコンテナである。ここで１次元データとは、横軸、強度、エラーの最低３つの配列を用いて表現されるデータのことである。
ElementContainer は横軸などの配列を収めるデータ領域と、それに付随した情報を収めておくヘッダ領域から成っている（Fig.1）。

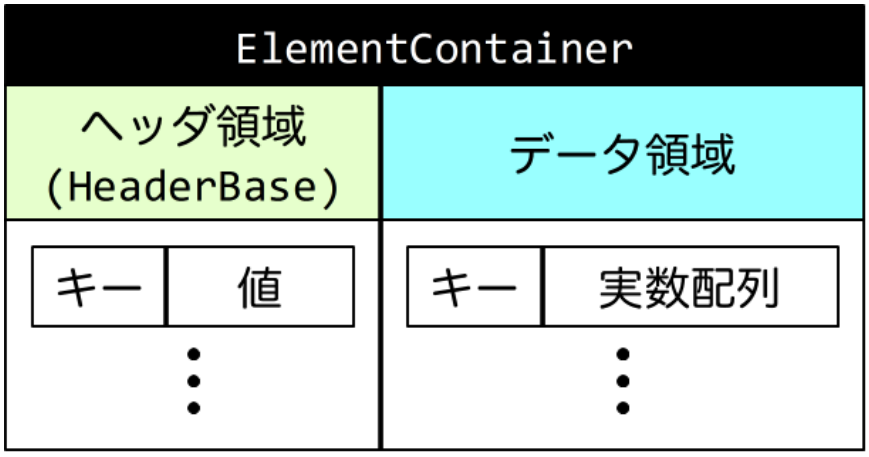

Fig.1 ElementContainer の模式図

データ領域は複数の実数配列を入れることができる。ヘッダ領域には整数、実数、文字列、および、それらの配列を複数入れることができる。

データ領域の実数配列には、それぞれに名前（キーと呼ばれる文字列、長さに制限はない）を付けて収める。例えば次のように用いる。

```
import Manyo
ec = Manyo.ElementContainer()

i=[10.0,22.0,33.0,44.0,55.0]
ec.Add("Intensity", i)
```

ここで"import Manyo"は Python から万葉ライブラリを呼び出すためのモジュールを読み込むコマンドである。一般に Python のモジュールのコマンドは、そのモジュール名にピリオドで接続することで実行される。例えば２行目の Manyo.ElementContainer()は、Manyo モジュールの ElementContainer というコマンド（メソッドともいう）を実行している。なおこのコマンドにより「新しく（空っぽの）ElementContainer を作る」ことができる。

また ec は ElementContainer、i は強度の実数配列である。収めた配列は、次のようにしてキーを用いて取り出すことができる。

```
ret = ec.Put("Intensity")
```

多くの場合、ElementContainer のデータ領域には、物理量、強度、強度エラーの三種類の実数配列からなるヒストグラムデータを収める。 ヒストグラムデータとは、Fig.2 (a)に示すように、ある物理量の軸に沿って区切り（bin:ビン）を設け、その区切り内における強度とエラーを扱うものである。よって、その強度はいわゆる積分値であり、区切り幅（ビン幅）が大きくなるとその強度も大きくなる。

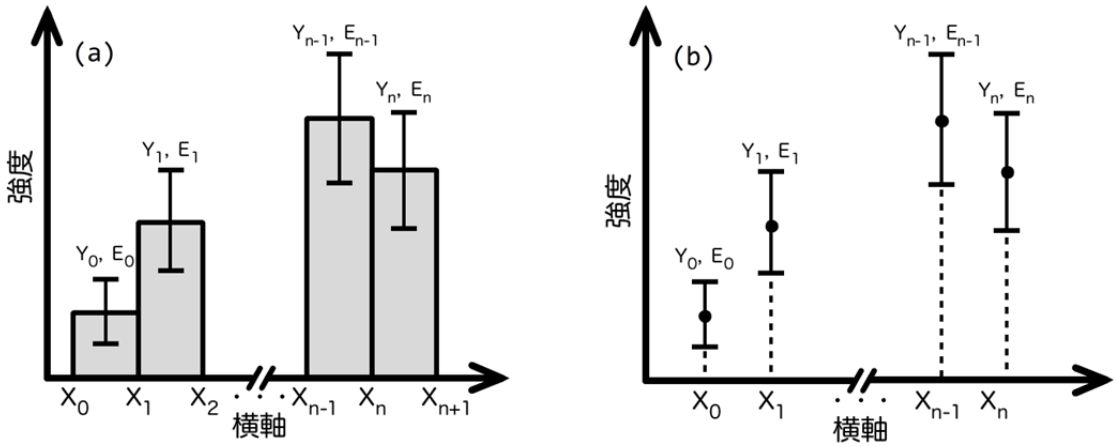

Fig.2 ヒストグラムデータ(a)と散布データ(b)

一方、一般に扱われるデータは Fig.2 (b)に示したように横軸と縦軸が一対一で対応するものであり、その縦軸の値はその横軸の単位あたりの強度となっているものが大部分である。我々はこれを散布データと呼び、ヒストグラムデータとは区別して扱う。万葉ライブラリのデータコンテナは、ヒストグラムデータを扱うことを基本とする。

データをヒストグラムとして扱うことにより、次のような利点がある。

- 計測機器との直接的なデータのやりとりが可能

- 情報の劣化がない

- 単位変換が容易（ヤコビアンを考慮せずに済む）

このようにヒストグラムデータの場合、当然ながら横軸の区切りの個数は強度、強度エラーよりも一つ多い。次のようにして ElementContainer にヒストグラムデータを収める。

```
import Manyo
ec = Manyo.ElementContainer()
t=[0,100,200,300,400,500]    # tof （強度やエラーより個数が一つ多い）
i=[10.0,22.0,33.0,44.0,55.0] # Intensity （強度）
e=[5.0,11.0,16.5,22.0,27.5]  # Error (エラー）

ec.Add("TOF", t)
ec.Add("Intensity", i)
ec.Add("Error", e)
```

ここで ec は ElementContainer、t、i、e は飛行時間、強度、強度エラーの実数配列とする。ここでは単純に t を TOF というキーで、i を Intensity、e を Error というキーで収納しただけであり、後述する ElementContainer の四則演算機能やその他の関数を利用するには、どの配列が区切りで、どの配列が強度やエラーなのかを指定する必要がある。

```
ec.SetKeys("TOF", "Intensity", "Error")
```

このように区切り、強度、エラーとしてどのキーの実数配列を用いるかを明示する。のちに述べるように、ElementContainer を用いた関数では ElementContainer は SetKeys されていることが期待される。また収納できる実数配列の数に制限は無いため、区切り、強度、エラーの実数配列を収納した時は必ず SetKeys を実行しておくことが重要である。

2.2.1. 配列の収納と取り出しについて

ElementContainer にデータ列を収納する時、サンプルコードでは Python のリスト形式のデータを収納しているように見える。しかし万葉ライブラリ自体は C++で実装されているため、実際には C++に対応した別の形式で ElementContainer 内に収納されている。それは vector<double>と呼ばれる形式（C++の標準ライブラリ Standard Template Library のコンテナ）であるが、ElementContainer はリスト形式で実数配列を受け取ると内部でこの形式に変換している。収納されている実数配列を取り出すためのコマンドは準備されているが、ほとんどの場合リスト形式ではなく、vector<double>となることに注意すること。 前述した

```
ret = ec.Put("Intensity")
```

として取り出された実数配列もリストではなく、vector<double>である。

唯一リストで取り出せるのは、SetKyes で設定された区切り、強度、エラーである。これらは特別な扱いを受けており、リストで取り出す方法が用意されている。

```
ec.SetKeys("TOF", "Intensity", "Error")
# リストで取り出す場合
x_list = ec.PutXList()  # x 軸をリストで取り出す
i_list = ec.PutYList()  # 強度をリストで取り出す
e_list = ec.PutEList()  # エラーをリストで取り出す
```

```
# vector<double>で取り出す場合
x_vec = ec.PutX()
i_vec = ec.PutY()
e_vec = ec.PutE()
```

なお、現在どの実数配列が SetKeys で設定されているかを確認するには PutXKey、PutYKey、PutEKey や、PutHistKeyList を用いる。

```
print "X key=",ec.PutXKey()  # X のキー
print "Y key=",ec.PutYKey()  # Y のキー
print "E key=",ec.PutEKey()  # E のキー
print "[X key, Y key, E key]=",ec.PutHistKeyList()  # リストで取り出す
```

2.2.2. ElementContainer の内容の表示 (Dump)

先に述べたように実数配列などを ElementContainer に収めた時に、何が収納されているのかを以下のように表示することができる。

```
ec.Dump()
```

【結果】

```
 1  *** header object start
 2  ***Int4Map***
 3  Index    Key    Value
 4
 5  ***DoubleMap***
 6  Index    Key    Value
 7
 8  ***StringMap***
 9  Index    Key    Value
10
11  ***Int4VectorMap***
12  Index    Key    Size    Value
13
14  ***DoubleVectorMap***
15  Index    Key    Size    Value
16
17  ***StringVectorMap***
18  Index    Key    Size    Value
19
20  *** header object end
21
22  The number of vectors is 3
23
24  x key = TOF
25  y key = Intensity
26  e key = Error
```

```
27
28   Index    Key    Size    Unit    Values
29   0    Error    5    None    [5,11,16.5,22,27.5]
30   1    Intensity    5    None    [10,22,33,44,55]
31   2    TOF    6    None    [0,100,200,300,400,500]
```

この Dump()コマンドによりヘッダ領域の内容とデータ領域の情報が示される。この例では最初の２０行がヘッダ情報部分であるが、何も収納されていないのでタイトルなどが表示されているだけである。一方最後の１０行がデータ領域の情報である。こちらには先の例で行ったように、

1. TOF、Intensity、Error のキーを持つ３つの実数配列が収納されている
2. それらの三つの配列がこの ElementContainer の区切り、強度、エラー（ x key, y key, e key ）として登録されている

ことがわかる。

2.2.3. ElementContainer の複製

Python などで ElementContianer を扱うときの注意点の一つが、ElementContainer の複製を作ろうとした時である。例えば以下のように ec1 という ElementContainer があったとして、それを ec2 という名前の複製を作成しようとする。

```
ec1 = Manyo.ElementContainer()
ec1.Add("x",[1,2,3,4])
ec1.Add("y",[1,1,1])
ec1.Add("e",[1,1,1])
ec1.SetKeys("x","y","e")

ec2 = ec1
```

一見違う名前のデータコンテナに代入したように見えても、それは単に一つのデータコンテナの実体に異なる名前が付けられただけのものである。例えば、ec2 に以下のようにヒストグラムの変更を行った場合、その変更が ec1 にも加えられている、つまり ec1 と ec2 は見た目は違うが中身は同じであることを示している。

```
ec2.Add("y2",[2,2,2])
ec2.SetKeys( "x", "y2", "e" )

ec1.Dump()
```

【結果】

```
（省略）
The number of vectors is 4

x key = x
y key = y2
e key = e
```

```
Index    Key    Size    Unit    Values
0    e    3    None    [1,1,1]
1    x    4    None    [1,2,3,4]
2    y    3    None    [1,1,1]
3    y2    3    None    [2,2,2]
```

そこで実際に複製を作る場合、新しく ElementContainer を作ることを明示する必要がある。正しくは下記のように行う。

```
ec1 = Manyo.ElementContainer()
ec1.Add("x",[1,2,3,4])
ec1.Add("y",[1,1,1])
ec1.Add("e",[1,1,1])
ec1.SetKeys("x","y","e")

ec2 = Manyo.ElementContainer( ec1 )
```

つまり、今まで使用してきた「新しい（空っぽの）ElementContainer を作る」コマンドに、複製を撮りたい ElementContainer を引数として与えれば良い。この結果作成された ec2 は一見 ec1 と同じであるがその実体は異なっているので、先に行ったように ec2 を書き換えても ec1 は不変である。

```
ec2.Add("y2",[2,2,2])
ec2.SetKeys( "x", "y2", "e" )

ec1.Dump()
```

【結果】

```
（省略）
The number of vectors is 3

x key = x
y key = y
e key = e

Index    Key    Size    Unit    Values
0    e    3    None    [1,1,1]
1    x    4    None    [1,2,3,4]
2    y    3    None    [1,1,1]
```

2.2.4. ElementContainer 関連のコマンドまとめ

ElementContainer を作成するためのコマンドを Table 1 に、ElementContainer が持つコマンドを Table 2 に示す。

Table 1 ElementContainer 作成コマンド

コマンド	機能
ElementContianer の作成	Manyo.ElementContainer()
ElementContianer の複製	Manyo.ElementContainer(ec)

Table 2 ElementContainer のコマンド

コマンド	機能
Add(key, XXX)	ElementContainer への実数配列の追加。 XXX は Python List か vector<double>
Put(key)	キーの実数配列の取り出し （ vector<double> ）
SetKeys(xkey, ykey, ekey)	ヒストグラムの X,Y,E となる実数配列の指定
PutXList()	X の実数配列の取り出し(Python List)
PutYList()	Y の実数配列の取り出し(Python List)
PutEList()	E の実数配列の取り出し(Python List)
PutX()	X の実数配列の取り出し(vector<double>)
PutY()	Y の実数配列の取り出し(vector<double>)
PutE()	E の実数配列の取り出し(vector<double>)
PutXKey()	X の実数配列のキー
PutYKey()	Y の実数配列のキー
PutEKey()	E の実数配列のキー
PutKeysList()	実数配列のキーのリスト
PutHistKeyList()	ヒストグラムのキーのリスト
PutHeader()	ヘッダ情報の複製の取り出し
InputHeader()	ヘッダ情報の置き換え
SaveTextFile(FileName)	ヒストグラムのテキストファイル保存
LoadTextFile(FileName)	テキストファイルからのヒストグラム読み込み

いくつかのコマンドは次章以降で触れられる。また Appendix B「万葉ライブラリ関数リファレンス」も参照のこと。

2.3. ヘッダ

ヘッダ領域にはデータ領域に収めたデータに関する様々な情報（整数、実数、文字列、および、それらの配列）を名前（キー）を付けて収められる。例えば、データ領域に収めたヒストグラムを観測した検出ピクセルの ID や位置情報を記録することに用いる。ヘッダにこれらの情報を記録することで、データに付随するパラメータの管理が容易になり、また、これらをデータの検索に用いることができるという利点がある。次の表(Table 3)にそれぞれのデータコンテナでの利用例を挙げた。

Table 3 データコンテナのヘッダ利用例

種類	例
ElementContainer	ピクセル ID、検出器、散乱角、ピクセル位置
ElementContainerArray	検出器 ID、マスク情報
ElementContainerMatrix	ラン番号、モニターカウント

2.3.1. ヘッダへの収納の意味（HeaderBase）

ヘッダ情報は HeaderBase と呼ばれる形式のコンテナに、実数、整数、文字列といった情報の形式ごとに分けられて保存される。 先に ElementContainer はデータ領域とヘッダ領域の二つの領域を持つと述べたが、実際にはヘッダ領域とはこの HeaderBase 形式コンテナのことである。

よって ElementContaier のヘッダへの収納とは、「ElementContainer 内の HeaderBase コンテナに情報を収める」ということを意味している。 同様に、ElementContainerArray や ElementContainerMatrix のヘッダへの収納も、この HeaderBase を利用することになる。

2.3.2. ElementContainer のヘッダへの収納

ElementContainer ヘッダへの収納は、実数配列と同様にキーを用いて以下のように、AddToHeader()というコマンドで行うことができる。

```
import Manyo

ec=Manyo.ElementContainer()
ec.AddToHeader( "L1", 20.193 )       # L1 というキーで実数を収納
ec.AddToHeader( "NumOfPixels", 100 ) # NumOfPixels というキーで整数を収納
ec.AddToHeader( "Sample", "CuGeO3" ) # Sample というキーで文字列を収納
```

収納されたヘッダ情報は、先に示した Dump()コマンドでデータ領域の実数配列と同時に表示される。

```
ec.Dump()
```

【結果】

```
*** header object start
```

```
***Int4Map***
Index       Key     Value
0   NumOfPixels     100

***DoubleMap***
Index       Key     Value
0   L1      20.193

***StringMap***
Index       Key     Value
0   Sample  CuGeO3

***Int4VectorMap***
Index       Key     Size    Value

***DoubleVectorMap***
Index       Key     Size    Value

***StringVectorMap***
Index       Key     Size    Value

*** header object end

The number of vectors is 0

x key = None
y key = None
e key = None

Index       Key     Size    Unit    Values
```

このように収めた情報が情報の形式（整数、実数、文字列など）に分けて収納されていることがわかる。

2.3.3. ヘッダからの情報の取り出し

ElementContainer に収納されたヘッダ情報を取り出すこともできる。ただし ElementContainer から直接取り出すコマンドはない。 実際には、

1. ElementContainer から HeaderBase を取り出し、
2. その HeaderBase から収められた情報を取り出す、

という２段階の手順を踏むことになる。

また収められた情報の形式ごとに取り出しコマンドが異なることに注意すること。 この取り出し方法は、対象が ElementContainerArray や ElementContainerMatrix でも同じである。

```
hh = ec.PutHeader()  # ElementContainer から hh という名前で HeaderBase を取り出す

L1=hh.PutDouble( "L1" )  # HeaderBase から L1 というキーの実数を取り出す
NumOfPixels=hh.PutInt4( "NumOfPixels" ) # NumOfPixels キーの整数を取り出す
```

```
SampleName=hh.PutString( "Sample" )     # SampleName キーの文字列を取り出す

print "L1=",L1
print "NumOfPixels=",NumOfPixels
print "Sample Name=",SampleName

hh.Dump()
```

【結果】

```
L1= 20.193
NumOfPixels= 100
Sample Name= CuGeO3

***Int4Map***
Index       Key     Value
0   NumOfPixels     100

***DoubleMap***
Index       Key     Value
0   L1      20.193

***StringMap***
Index       Key     Value
0   Sample  CuGeO3

***Int4VectorMap***
Index       Key     Size    Value

***DoubleVectorMap***
Index       Key     Size    Value

***StringVectorMap***
Index       Key     Size    Value
```

ここで、Dump()は ElementContainer の時と同様に、HeaderBase の内容を書き出すコマンドである。

2.3.4. HeaderBase の扱い方

HeaderBase は、下に示す Table 4 にあるような形式の情報が保持されている。いずれも名前（キー）と情報の組み合わせで収められている。 取り出しコマンドも Table 4 に、収納コマンドは Table 5 に示しておく。なお、取り出す時の形式が、情報を Add した時の形式（整数、実数、文字列）と異なっていても、自動的に変換される。ただし、文字列を数値（整数、実数）に変換する場合、変換が明らかに不可能な場合はワーニングメッセージとともに、0 または 0.0 が戻ってくる。

Table 4 HeaderBase からの情報取り出しコマンド

形式	表現	取り出しコマンド
実数	Double	PutDouble(key)
整数	Int4	PutInt4(key)
文字列	String	PutString(key)
実数配列	vector<Double>	PutDoubleVector(key)
整数配列	vector<Int4>	PutInt4Vector(key)
文字列の配列	vector<String>	PutStringVector(key)

Table 5 HeaderBase への情報収納コマンド

コマンド	形式	機能
Add(key, XXX)	XXX を自動判別	新しいキーで値を追加する （既にキーがあるとエラー）
OverWrite(key, XXX)	XXX を自動判別	既にあるキーを値で置き換える （キーがない場合は Add と同じ）

基本的に HeaderBase は単独で用いられることはなく、ElementConainer や ElementContainerArray, Matrix から取り出される形で使用されることが多いであろう。

先に述べた PutHeader()では、データコンテナ内の HeaderBase のコピーが作成される。その意味は、この PutHeader()による HeaderBase に対して情報を加えたり変更しても取り出し元のデータコンテナの HeaderBase には影響を与えない、ということである。

この変更をデータコンテナ内の HeaderBase に反映させるには、InputHeader()コマンドを利用して変更後の HeaderBase で ElementContainer のヘッダを置き換える必要がある。それらのコマンドを Table 6 に示す。

Table 6 ElementContainer の HeaderBase 取り出し・置き換え

コマンド	機能
PutHeader()	データコンテナ内の HeaderBase のコピーを取り出す。
InputHeader(XXX)	データコンテナ内の HeaderBase を XXX で置き換える。

よって、ElementContainer からヘッダを取り出してその内容を変更する場合のスクリプトは以下のようになる。

```
ec=Manyo.ElementContainer()

ec.AddToHeader( "SampleName", "Cupper" )
ec.AddToHeader( "Temperature", 20.0 )
ec.AddToHeader( "Pressure", 1.0 )

ec.Dump()

head = ec.PutHeader()
head.OverWrite( "SampleName", "Gold" )
head.OverWrite( "Temperature", 300.0 )
head.OverWrite( "Pressure", 0.5 )

ec.InputHeader( head )

ec.Dump()
```

ここでは、まず新しく作成した ElementContainer のヘッダに、"SampleName"として"Cupper"を、"Temperature"として 20.0 を、"Pressure"として 1.0 を、与えている。その内容を確認するために *Dump* を行っている。その後、取り出した HeaderBase に対して、先ほどのキーの値をそれぞれ"Gold"、300.0、0.5 に置き換えている。最後にその編集した HeaderBase で ElementContainer のヘッダを上書きして、その *Dump* を行っている。

【結果】

```
*** header object start
***Int4Map***
Index       Key     Value

***DoubleMap***
Index       Key     Value
0   Temperature     20
1   Pressure        1

***StringMap***
Index       Key     Value
0   SampleName      Cupper

***Int4VectorMap***
Index       Key     Size    Value

（中略）

*** header object start
***Int4Map***
Index       Key     Value

***DoubleMap***
Index       Key     Value
```

```
0   Temperature     300
1   Pressure        0.5

***StringMap***
Index       Key     Value
0   SampleName      Gold

***Int4VectorMap***
Index       Key     Size    Value

（以下略）
```

確かに、"SampleName"が"Cupper"から"Gold"に、"Temperature"が 20.0 から 300.0 に、"Pressure"が 1.0 から 0.5 へ変更されていることがわかる。

2.3.4.1. HeaderBase の内容の確認

HeaderBase 自体も Dump()コマンドを持っており、自身の内容の表示ができる。

```
head1 = ec.PutHeader()
head1.Dump()
```

【結果】

```
*** header object start
***Int4Map***
Index       Key     Value

***DoubleMap***
Index       Key     Value
0   Info    2

（以下略）
```

このように実は ElementContainer の Dump()は、まさにこの HeaderBase の Dump()も表示していることがわかる。

2.4. 万葉ライブラリの変数・配列について

ここまで万葉ライブラリの基本機能である ElementContainer やヘッダを見てきたが、いずれも測定データなどの数値や文字列、及びそれらの配列といった情報を Python 上で与えたり受け取ったりすることが必要である。

しかしながら万葉ライブラリ自体は C++で作成されているため、データコンテナや関数の引数及び戻り値にはしばしば C++の変数のタイプ（型）が使用されている。

そのため万葉ライブラリの関数が Python 上で簡単に扱えるように、一部の関数は Python の変数を半自動的に C++の変数に変換する、もしくはその逆を行うように実装されている。しかし、特に数値配列を扱う場合、直接の変換ができない場合も多い。

そこで本節では万葉ライブラリ（C++）と Python の間で型の違いがどのように扱われているのか、Python 上でそれらを扱う場合の特徴と注意点を述べておく。

2.4.1. 変数のタイプ

万葉ライブラリで使用される数値や文字列など単純な変数（配列や特殊なコンテナを省く）の型は、少し独特な表現が用いられる。よく使用されるものを Table 7 に列挙する。

Table 7 万葉ライブラリで使用される変数

万葉ライブラリ	Python	型	C 言語
Int4	int	整数（符号あり）	int
UInt4	int に変換	正の整数（符号なし）	unsigned int
Int8	int	長整数（符号あり）	long long
UInt8	int に変換	正の長整数（符号なし）	unsigned long long
Double	float	実数	double
string	str	文字列	string

本書の Appendix B「万葉ライブラリ関数リファレンス」にあるように、万葉ライブラリの関数における引数や戻り値は、Int4, UInt4, Double, string など万葉ライブラリや C++の型で定義・表現されている。しかしながら、Python から見ればすべて int や float, str で置き換えることができる。

また「Python と万葉ライブラリ間のこれらの変換は自動的に行われる」ので、使用時には特に気にしなくていよい。

ただし次節で述べるように、これら変数の型を用いた配列は少し異なるので注意が必要である。

2.4.2. 配列

一般に Python では、数値の配列を扱うのに Python のリストが使用される。Python のリストは汎用的・高機能であり、非常に簡便に使えるが、万葉ライブラリのベースである C++ではそれをそのまま利用することはできないため、別の入れ物（コンテナ）で置き換えている。

万葉ライブラリでの数値や文字列などの配列を扱う枠組みとして、標準テンプレートライブラリ（Standard Template Library : STL）と呼ばれるものが用いられている。その STL の中で定義されている vector と呼ばれるコンテナを積極的に利用している。

万葉ライブラリは Python 上でこの vector をある程度 Python リストとかなり近い形で扱えるように実装されている。使用できる vector と Python 上での作成方法を Table 8 に示す。

Table 8 万葉ライブラリで使用する vector

型	表記	空の配列作成
整数配列	vector<Int4>	MakeInt4Vector()
正の整数配列	vector<UInt4>	MakeUInt4Vector()
実数配列	vector<Double>	MakeDoubleVector()
文字列の配列	vector<string>	MakeStringVector()

次に、Python リストと、vector の操作の違いを Table 9 に示す。

Table 9 Python リストと vector の違い

機能	Python リストでの操作	万葉ライブラリ vector<XXX> での操作
空の配列作成	AA=[]	AA=Manyo.MakeXXXVector()
配列追加	AA.append(m)	AA.append(m) （ただし XXX と同じ型）
値取り出し	AA[n]	AA[n]
値代入	AA[n]=m	AA[n]=m （ただし XXX と同じ型）
配列の大きさ	len(AA)	AA.size()
配列最後の値	AA[-1]	AA[AA.size()-1]
配列ソート	AA.sort()	単独では不可
配列逆順	AA.reverse()	単独では不可

このように、配列追加や値の出し入れなどの単純な操作に関しては、両者ほぼ同じ仕様で扱うことができる。

例えば for ループで配列の文字を取り出す場合、Python リストを用いると

```
for word in word_list:
    print "word=",word

もしくは

for i in range( len(word_list) ):
    print "word=",word_list[i]
```

となり、vector を用いると

```
for i in range( word_vect.size() ):
    print "word=",word_vect[i]
```

となる。特に range を用いて利用する場合は非常に似通っている。ただし変数だけだと実際にどちらなのかわかりづらいのでコード作成時には注意すること。 なお、変数が目的の型かどうかを確認するのに isinstance という Python コマンドがあるので利用するのも良い。

```
>>> import Manyo
>>> vv = Manyo.MakeDoubleVector()
>>> ll = []
>>>
>>> isinstance( ll, list )
True
>>> isinstance( vv, Manyo.VecDouble )
True
```

なお、型の確認に使用されている VecDouble は万葉ライブラリ内で定義されている型の表現であり、他のものは Table 10 で示したようになる。（vector<string>だけ少し異なるので注意）

Table 10 万葉ライブラリ内で定義される vector 型の表現

型	表現
vector<Int4>	VecInt4
vector<UInt4>	VecUInt4
vector<Double>	VecDouble
vector<string>	StringVector

2.4.3. Python リストと vector 間の変換

万葉ライブラリの関数において引数において vector を必要とするものもある。そのために万葉ライブラリには、Python リストと vector のデータの変換を行うための関数が用意されている。

用意されている関数は、CppToPython である。

以下に、実数の Python リストを C++の実数配列 vector<Double> に変換する様子を示す。

```
1   >>> import Manyo
2   >>> ll = [ 1.0, 2.0, 3.0, 4.0 ]
3   >>> ctp = Manyo.CppToPython()
4   >>> vec = ctp.ListToDoubleVector( ll )
5   >>> isinstans( vec, Manyo.VecDouble )
6   True
7   >>> ll_from_vec = ctp.VectorToList( vec )
8   >>> print ll_from_vec
9   [1.0, 2.0, 3.0, 4.0]
```

1 行目： 万葉ライブラリモジュールを読み込む
2 行目： Python リスト *ll* を用意する
3 行目： *CppToPython* 関数を呼び出し、 *ctp* とする
4 行目： *ll* を vector<Double> に変換し結果を *vec* に入れる
5 行目： *vec* が vector<Double> かどうかを確認する
6 行目： True と戻ってきたので *vec* は vector<Double>である
7 行目： 次に *vec* を引数として、Python リストに逆変換し、結果を *ll_from_vec* に入れる
8 行目： 結果を表示して Python リストであることを確認する

他の変換についても Table 11 に簡単に示す。

Table 11 Python リストと vector の変換コマンド

変換元	変換先	コマンド
Python リスト	vector<Int4>	ListToInt4Vector
Python リスト	vector<UInt4>	ListToUInt4Vector
Python リスト	vector<Double>	ListToDoubleVector
Python リスト	vector<string>	ListToStringVector
vector<Int4>	Python リスト	VectorToList
vector<UInt4>	Python リスト	（上に同じ）
vector<Double>	Python リスト	（上に同じ）
vector<string>	Python リスト	VectorStringToList

この Table 11 にあるように、

- Python リストから、それぞれの型の vector へは独立したコマンド
- vector から、Python リストへは文字列以外は一つのコマンド

となっている。

2.5. ElementContainerArray と ElementContainerMatrix

ElementContainerArray、ElementContainerMatrix を用いると Fig. 3 に示すように ElementContainer を階層的にまとめることができる。

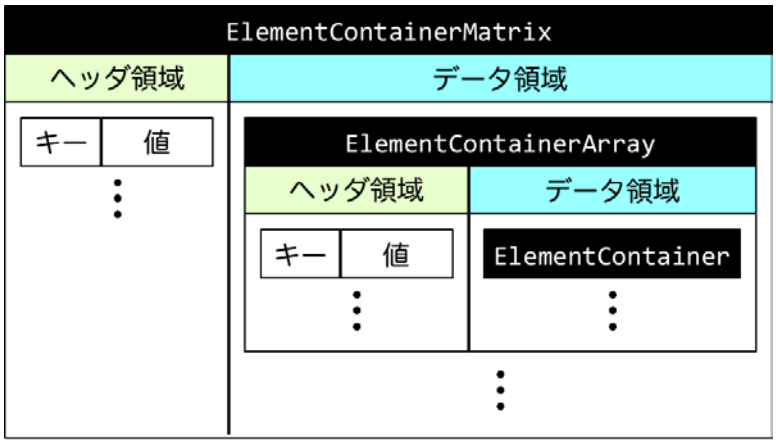

Fig.3 ElementContainerArray、ElementContainerMatrix の模式図

どちらも ElementContainer と同様にデータ領域とヘッダ領域から成っている。ElementContainerArray の場合のデータ領域は複数の ElementContainer であり、ElementContainerMatrix の場合のデータ領域は複数の ElementContainerArray である。それらのヘッダ領域には ElementContainer の場合と同様に様々な情報を収めることができる。

主な用途としては、ElementContainerArray は一つの検出器（内部に多数の検出部位を含んでいる）分のデータを保持するのに使用され、ElementContainerMatrix は一つの検出器グループ（バンクなど）や、装置全体の検出器のデータを全てまとめて扱いたい時に使用される。

2.5.1. ElementContainerArray の機能

ElementContainerArray は、その内部のデータ領域には複数の ElementContainer を同時に収めることができ、ヘッダ領域には一つの HeaderBase がある。

2.5.1.1. データ領域へのアクセス

ElementContainerArray に ElementContainer を追加するには、Add()を用いる。

```
# ElementContainer の作成
ec1 = Manyo.ElementContainer()
ec1.Add("x",[1,2,3,4])
ec1.Add("y",[1,1,1])
ec1.Add("e",[1,1,1])
ec1.SetKeys("x","y","e")

# ElementContainer の複製
ec2 = Manyo.ElementContainer( ec1 )
```

```
ec3 = Manyo.ElementContainer( ec1 )

# ElementContainer 区別用にヘッダへ登録
ec1.AddToHeader( "Name", "ec1" )
ec2.AddToHeader( "Name", "ec2" )
ec3.AddToHeader( "Name", "ec3" )

# ElementContainerArray の作成
eca = Manyo.ElementContainerArray()

# ElementContainerArray へ ec1,ec2.ec3 を追加
eca.Add( ec1 )
eca.Add( ec2 )
eca.Add( ec3 )
```

ここで、ElementContainerArray に Add()によって収められた ElementContainer は、ec1 や ec2, ec3 の複製であるので、ec1 や ec2 を変更したり、消去(del)したりしても ElementContainerArray 内の ElementContainer は変更されない。

追加した ElementContainer の総数を知りたい場合、PutSize()を用いる。

```
print "The number of ElementContainers = ",eca.PutSize()
```

【結果】

```
The number of ElementContainers = 3
```

一方、収められている ElementContainer を取り出す場合は、Put()を用いる。

```
ec = eca.Put(0)
print "ElementContainer name = ",ec.PutHeader().Put("Name")

ec = eca.Put(1)
print "ElementContainer name = ",ec.PutHeader().Put("Name")

ec = eca.Put(3)
print "ElementContainer name = ",ec.PutHeader().Put("Name")
```

これは Python の for 文と、PutSize()を用いてより簡単に書ける。

```
for i in range( eca.PutSize() ):
    ec = eca.Put(i)
    print "ElementContainer name = ",ec.PutHeader().Put("Name")
```

【結果】

```
ElementContainer name = ec1
ElementContainer name = ec2
ElementContainer name = ec3
```

ここで注意するのは、 Put()によって取り出された ElementContainer は ElementContainerArray 内のデータの複製であることであり、取り出された ElementContainer に変更を加えたり消去しても ElementContainerArray 内のデータには影響がないということである。 もし ElementContainerArray 内のデータそのものを取り出したい場合、PutPointer()もしくは eca( )などのコマンドで取り出すことが可能であるが、これは万が一取り出したデータを消去すると ElementContainerArray の中身も消去され、整合性が取れなくなり、エラーが起きたり最悪の場合、実行が停止（落ちる）ことになるため、扱いには注意が必要である。よって特に必要でなければ Put()を用いるべきである。

2.5.1.2. ヘッダ領域へのアクセス

ElementContainer におけるヘッダ領域の情報の取り出し、置き換えと全く同じである。 すなわち、Table 12 で示すコマンドが使用できる。

Table 12 ElementContainer の HeaderBase 取り出し・置き換え

コマンド	機能
PutHeader()	データコンテナ内の HeaderBase のコピーを取り出す。
InputHeader(XXX)	データコンテナ内の HeaderBase を XXX で置き換える。

2.5.1.3. ElementContainerArray の複製

ElementContianerArray の複製も、ElementContainer の複製と同様に、作成時に引数として複製を行いたい ElementContainerArray を与える。

```
    eca_copy = Manyo.ElementContainerArray( eca )
```

2.5.1.4. ElementContainerArray 関連のコマンドまとめ

ElementContainerArray を作成するコマンドを Table 13 に、ElementContainerArray のコマンドを Table 14 に示す。

Table 13 ElementContainerArray 作成コマンド

コマンド	機能
ElementContianerArray の作成	Manyo.ElementContainerArray()
ElementContianerArray の複製	Manyo.ElementContainerArray(eca)

Table 14 ElementContainerArray のコマンド

コマンド	機能
Add(ec)	ElementContainer の追加
Put(i)	i 番目の ElementContainer の取り出し
PutSize()	含まれている ElementContainer の個数
PutHeader()	ヘッダ情報の取り出し
InputHeader()	ヘッダ情報の置き換え

また Appendix B「万葉ライブラリ関数リファレンス」も参照のこと。

2.5.2. ElementContainerMatrix の機能

ElementContainerMatrix は、その内部のデータ領域には複数の ElementContainerArray を同時に収めることができ、ヘッダ領域には一つの HeaderBase がある。

それ以外のコマンドなどは、全く ElementContainerMatrix とほぼ同じである。

2.5.2.1. ElementContainerMatrix のコマンドまとめ

ElementContainerMarix を作成するコマンドを Table 15 に、ElementContainerMatrix のコマンドを Table 16 に示す。

Table 15 ElementContainerMatrix 作成コマンド

コマンド	機能
ElementContianerMatrix の作成	Manyo.ElementContainerMatrix()
ElementContianerMatrix の複製	Manyo.ElementContainerMatrix(ecm)

Table 16 ElementContainerMatrix のコマンド

コマンド	機能
Add(eca)	ElementContainerArray の追加
Put(i)	i 番目の ElementContainerArray の取り出し
PutSize()	含まれている ElementContainerArray の個数
PutHeader()	ヘッダ情報の取り出し
InputHeader()	ヘッダ情報の置き換え

また Appendix B「万葉ライブラリ関数リファレンス」も参照のこと。

2.5.3. ヘッダ情報による検索

先に述べたように ElementContainerArray の主な用途としては、複数の検出単位を持つ検出器のデータをまるごと保持するのに使用されるが、その検出単位の情報で検索をかけ必要な検出単位のヒストグラムデータだけ取り出したい時がある。また ElmentContainerMatrix に対してもそこに含まれる ElementContainer や ElementContainerArray に対して検索をかけたいこともある。そのためにヘッダ情報で検索を行うコマンドが用意されている。万葉ライブラリのコマンドの SearchInHeader である。

簡単に使い方を示す。以下に示すスクリプトは、すでにヘッダ情報として"PIXELID"（整数）が入った ElementContainer が複数含まれている ElementContainerArray や ElementContainerMatrix に対して、特定の範囲の"PIXELID"を持つ ElementContainer だけ取り出すというものである。この例では、検索結果を ElementContainerArray に入れて返す。

```
# SearchInHeader を作成する
# この時に検索対象の ElementContainerArray
# もしくは ElementContainerMatrix を与える
import Manyo
SIH = Manyo.SearchInHeader( dat )

# "PIXELID"が 10〜20 のものを検索する
# 正しく検索できれば True が戻る
isFound = SIH.Search( "PIXELID", 10, 20 )

# もし True だったら検索結果を取り出す
ret_eca=None
if isFound:
    ret_eca = SIH.PutResultAsArray()

del SIH  # 使い終わったら delete する
```

また、ElementContainerMatrix に含まれている ElementContainerArray に対してヘッダ検索をかける場合は以下のように行う。

```
# SearchInHeader を作成する
# この時に検索対象の ElementContainerMatrix を与える
import Manyo
SIH = Manyo.SearchInHeader( dat )

# "DETID"が 10〜20 のものを検索する
# 正しく検索できれば True が戻る
isFound = SIH.SearchArray( "DETID", 10, 20 )

# もし True だったら検索結果を取り出す
ret_ecm=None
if isFound:
    ret_ecm = SIH.PutResultAsArray()

del SIH  # 使い終わったら delete する
```

その他の主なコマンドを Table 17, Table 18 に示す。

Table 17 SearchInHeader の主な検索コマンド

コマンド	対象
Search(Key, 値)	ElementContainer ヘッダ内の整数や実数情報
Search(Key, 上限値, 下限値)	ElementContainer ヘッダ内の整数や実数情報
Search(Key, 文字列)	ElementContainer ヘッダ内の文字列情報
SearchArray(Key, 値)	ElementContainerArray ヘッダ内の整数や実数情報
SearchArray(Key, 上限値, 下限値)	ElementContainerArray ヘッダ内の整数や実数情報
SearchArray(Key, 文字列)	ElementContainerArray ヘッダ内の文字列情報

Table 18 SearchInHeader の主な検索結果取り出しコマンド

コマンド	対象
PutResultAsArray()	Search()の結果を ElementContainerArray で返す。
PutResultSearchArray()	SearchArray()の結果を ElementContainerMatrix で返す。

また Appendix B「万葉ライブラリ関数リファレンス」も参照のこと。

3. データコンテナの演算

3.1. マスク

マスクとは何らかの処理を行う際にその処理に使用しない値に与えるものである。

ElementContainer のヒストグラムを用いて何らかの処理を行う場合、マスクされた値はその処理には使用しないことを意味する。ただし、これはあくまでルールであるため、実際に処理を行う関数を作成する際に、開発者が留意すべきものである。より具体的な例をあげれば、いくつかの ElementContainer の平均値を出す場合、マスクされた値は平均値の計算には使用しないように関数を書く、というのがルールとなる。

また、MLF における万葉ライブラリの拡張パッケージである「空蝉」などでは、決まり事として ElementContainer, ElementContainerArray のヘッダにキーとして"MASKED"という情報を与えており、この値が 1 ならマスクがかかっているデータコンテナである、と定義し、データ処理関数の中で使用している。

3.1.1. マスクのルール

1. マスクされた値には負のエラーを与える。
2. マスクのエラーの絶対値は何でもよいが、通常のエラー計算を行ったと期待される。
3. マスクは ElementContainer のヒストグラムの横軸の bin ごとにつけることができる。
4. 関数を作成する際には、極力マスクを考慮して開発すること。

3.1.2. 四則演算におけるマスク

マスクは、万葉ライブラリにおいて ElementContainer 同士の演算でも考慮される。

演算時には通常どおりエラーを計算したのち、演算された値のどちらかにマスクがかかっていれば、その計算されたエラー値の符号をマイナスにする仕様となっている。

よってマスク処理が考えられていない関数においても、エラー伝搬・計算において二乗和を計算することで通常のエラーとして扱えるため、マスクのありなしでエラーが変化することはない。

3.2. ElementContainer 同士の四則演算

ElementContainer、および ElementContainerArray, ElementContainerMatrix 同士で四則演算を行う事ができる。 また、ElementContainer を定数倍、もしくは定数を加算、減算することも可能である。

以下の例では、ec1, ec2 にそれぞれ同じ横軸を持つヒストグラム（ElementContainer）が入っているものとする。 横軸が異なる場合の振る舞いについては、3.4 節にて述べる。

3.2.1. 四則演算

四則演算は Table 19 で示すように行う。

Table 19 ElementContainer 同士の四則演算

演算	フォーマット
足し算	ret_ec = ec1+ec2
引き算	ret_ec = ec1-ec2
かけ算	ret_ec = ec1*ec2
割り算	ret_ec = ec1/ec2

これら四則演算は、結果を新しい ElementContainer を作成して戻す（各演算の ret_ec の部分）。 またエラー伝搬も行う。また前節で述べたマスク処理も行われる。

3.2.2. 定数演算

定数演算とは定数倍、もしくは定数加算を行う機能のことである。それらを Table 20 に示す。

Table 20 ElementContainer の定数演算

演算	フォーマット
定数倍（新規コンテナ）	ret_ec = ec1.Mul(val)
定数倍（自分自身の変更）	ec1.MulMySelf(val)
定数加算（新規コンテナ）	ret_ec = ec1.Plus(val)
定数加算（自分自身の変更）	ec1.PlusMySelf(val)

定数倍

ec1 の強度を一律 val 倍する。エラーも val 倍される。

定数加算

ec1 の強度に一律に val を加える。エラーは変更なし。

演算結果

これらの関数には、演算結果を新しいコンテナとして作成するもの（新規コンテナ）と、自分自身を書き換えるもの（自分自身の変更）がある。必要に応じて使い分けることができる。

3.2.3. 使用例

以下に、四則演算と定数演算の具体的なサンプルスクリプトを示す。

3.2.3.1. サンプルデータ作成

四則演算を行うためのサンプルデータを作成する。

```
import Manyo

# ElementContainer 作成 1
ec1 = Manyo.ElementContainer()
ec1.Add("x",[0,1,2,3,4,5])
ec1.Add("y",[100,200,300,400,500])
ec1.Add("e",[10.0,14.14,17.32,20.0,22.36])
ec1.SetKeys( "x", "y", "e" )

# ElementContainer 作成 2
ec2 = Manyo.ElementContainer()
ec2.Add("x",[0,1,2,3,4,5])
ec2.Add("y",[200,300,400,500,600])
ec2.Add("e",[14.14,17.32,20.0,22.36,24.49])
ec2.SetKeys( "x", "y", "e" )
```

3.2.3.2. 四則演算

ElementContainer の加減乗除を行う。

```
# 四則演算
ret_add = ec1 + ec2
ret_sub = ec1 - ec2
ret_mul = ec1 * ec2
ret_div = ec1 / ec2

# 結果出力
print "ret_add Intensity=",ret_add.PutYList()
print "ret_add Error=",ret_add.PutEList()

print "ret_sub Intensity=",ret_sub.PutYList()
print "ret_sub Error=",ret_sub.PutEList()

print "ret_mul Intensity=",ret_mul.PutYList()
print "ret_mul Error=",ret_mul.PutEList()

print "ret_div Intensity=",ret_div.PutYList()
print "ret_div Error=",ret_div.PutEList()
```

【結果】

```
ret_add Intensity= [300.0, 500.0, 700.0, 900.0, 1100.0]
ret_add Error= [17.318764390105894, 22.358935573948955, 26.457180499818946,
                29.99949332905474, 33.162172727371164]
ret_sub Intensity= [-100.0, -100.0, -100.0, -100.0, -100.0]
ret_sub Error= [17.318764390105894, 22.358935573948955, 26.457180499818946,
                29.99949332905474, 33.162172727371164]
ret_mul Intensity= [20000.0, 60000.0, 120000.0, 200000.0, 300000.0]
ret_mul Error= [2449.3664486964785, 5476.665043619155, 9164.997763229405,
                13416.226593196761, 18163.950038469055]
ret_div Intensity= [0.5, 0.6666666666666666, 0.75, 0.8, 0.8333333333333334]
```

```
ret_div Error= [0.061234161217411966, 0.060851833817990614,
           0.057281236020183784, 0.05366490637278704, 0.05045541677352516]
```

3.2.3.3. 定数演算

ElementContainer の定数演算を行う。

```
# 定数演算
ret_mul2 = ec1.Mul( 1.5 )
ec1.MulMySelf( 2.6 )
ret_plus = ec2.Plus( 100 )

# 出力
print "ret_mul2 Intensity=",ret_mul2.PutYList()
print "ec1.MulMySerf(2.6) Intensity=",ec1.PutYList()
print "ret_plus Intensity=", ret_plus.PutYList()
```

【結果】

```
ret_mul2 Intensity= [150.0, 300.0, 450.0, 600.0, 750.0]
ec1.MulMySerf(2.6) Intensity= [260.0, 520.0, 780.0, 1040.0, 1300.0]
ret_plus Intensity= [300.0, 400.0, 500.0, 600.0, 700.0]
```

3.3. ElementContainer における横軸の bin 変換

横軸の bin 変換とは、ヒストグラムの bin の配列を変更することである。例えば一つの bin の強度が弱ければ統計精度を上げるために隣り合う複数の bin をまとめて一つの bin にしたいときがある。または不揃いな bin 幅の配列を等しい bin 幅の配列にしたい場合、異なる領域のデータとして扱いたい（bin の最大値、最小値を変更する）場合もあるだろう。

この bin 変換で気をつけないといけないことは、ヒストグラムの性質上「bin 幅が変わると見かけの強度（カウント数）も変わる」のが基本である、ということである。もう少し別の言い方をすれば「bin 幅を変更しても、bin あたりの見かけの強度（カウント数）は変わるが、その総和は変わらない」のがヒストグラムである(Fig.4 を参照)。

ElementContainer には、bin 変換を行う関数が存在するので、それを示す。

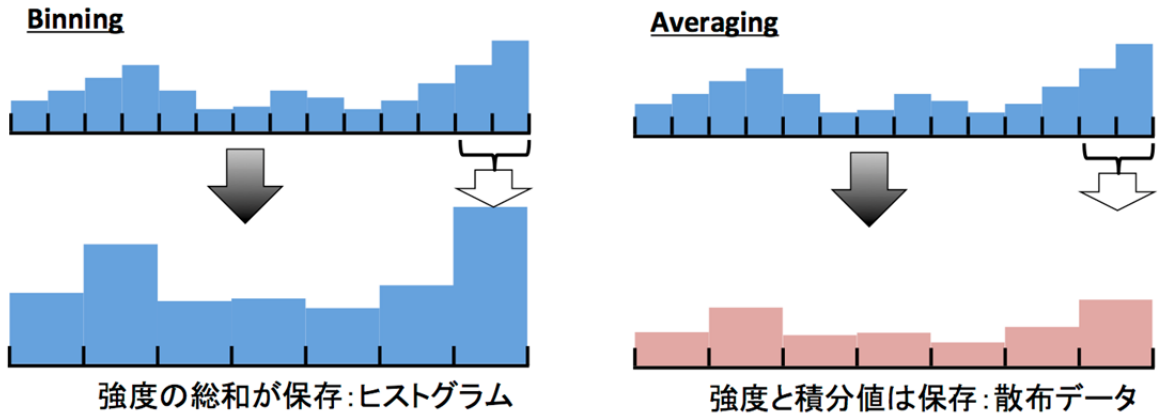

Fig.4 横軸 bin 変換の関数 *Binning* と *Averaging* による強度の変化

3.3.1. Binning, ReBin

ヒストグラムとして bin 変換を行うための関数である。Binning も ReBin も全く同一の動作を行う。 引数の与え方は 2 種類ある。単純に bin をまとめるために正の整数を与える方法と、変換したい配列を直接与える方法である。 変換結果は新しい ElementContainer が作成され納められ、それが戻り値となる。したがって関数を実行した ElementContainer は何も変化しない。

3.3.1.1. 引数が正の整数

隣り合う bin をまとめるのに使う。例えば、隣り合う 2 つの bin を 1 つにまとめる（強度は 2 倍）場合は以下のように記述する。

```
import Manyo
# ElementContainer 作成
ec = Manyo.ElementContainer()
ec.Add("x",[0,1,2,3,4,5])
ec.Add("y",[100,200,300,400,500])
ec.Add("e",[10.0,14.14,17.32,20.0,22.36])
ec.SetKeys( "x", "y", "e" )

new_ec = ec.Binning( 2 )
print "new_ec Intensity=", new_ec.PutYList()
```

【結果】

```
new_ec Intensity= [300.0, 700.0, 500.0]
```

3.3.1.2. 引数が bin の配列（vector<Double>）

新しい bin 配列を用いたいときに使用する。変更したい bin 配列を Python のリストではなく、C++ で定義されている vector<Double>という形式で与える必要がある。また先に述べたように bin 配列の変更で「強度の総和は変化しない」。

```
## new_x という vector<Double>を作成
new_x = Manyo.MakeDoubleVector()

## 0 から 200 まで 2.0 間隔の List 作成
v_list = [ float(i)*2.0 for i in range(100) ]

## v_list の中身を new_x(vector<Double>)へ入れる
for v in v_list:
        new_x.append( v )

## bin 変更の実行
new_ec = ec.Binning( new_x )

## 強度の出力
print "new_ec Intensity=", new_ec.PutYList()
```

【結果】

```
new_ec Intensity=[...]
```

また、しばしば用いられるのが別の ElementContainer の bin 配列を用いて bin 変換を行うやり方である。

```
new_ec = ec.Binning( ec_sample.PutX() )
```

ここでは、ec_sample の横軸 bin 配列を取り出し（PutX()は vector<Double>で取り出す）、それを用いて ec を Binning している。

3.3.2. Averaging

こちらも bin 配列を変更するが、Binning とは異なり、強度は単位あたりの平均値であるとして扱われる。したがって bin 幅を変更した場合でも、見た目の強度は変化しない。ただし当然ながら総和は保存されない。ElementContainer の強度が、ヒストグラムのカウント数（ヒストグラムデータ）ではなく、見た目ヒストグラムだが値は単位横軸あたりの強度（散布データ）に変換されている場合に使用できる。（将来、散布データを万葉ライブラリでサポート予定）引数の与え方や戻り値の扱いは、Binning とまったく同じである。

3.3.2.1. 引数が正の整数

```
new_ec = ec.Averaging( 2 )
```

3.3.2.2. 引数が bin の配列（vector<Double>）

```
new_x = Manyo.MakeDoubleVector()
for v in [float(i)*2.0 for i in range(100)]:
        new_x.append( v )
new_ec = ec.Averaging( new_x )
```

3.4. 四則演算するヒストグラムの横軸が異なる場合のルール

二つの ElementContainer 間における四則演算において、両者の横軸が異なる場合においては以下のようなルールが適応される。 横軸が異なるというのは、bin の幅と範囲の二点において、どちらか、もしくは両方異なることを言う。ヒストグラムの性質上「bin 幅が変わると見かけの強度（カウント数）も変わる」ことに注意。

ルールとしては以下の通り。Fig. 5 も参照のこと。

1. bin 幅が異なる場合、bin 幅の大きなほう（分解能が荒いほう）へ合わせる。（前項の Binning を参照）

2. 範囲が異なる場合、両者の重なる領域のみを演算し出力する。重複領域以外の演算データは失われる。

3. bin 幅もカバーする範囲も異なる場合、1.かつ 2.、すなわち両者が同時にカバーする領域で、かつ bin 幅の大きなほうにならう。

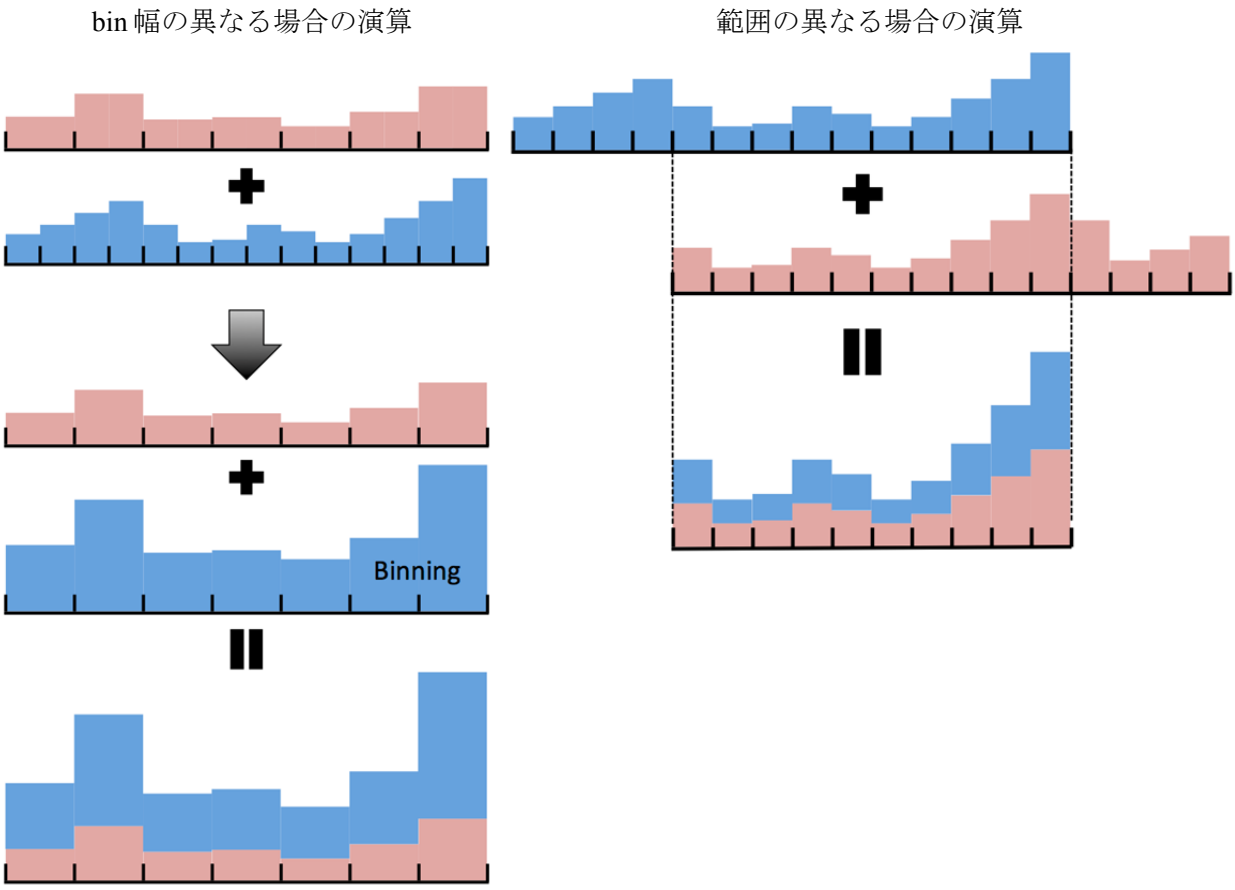

Fig.5 bin 幅や範囲の異なるヒストグラム同士の演算

4. ファイル入出力

万葉ライブラリにはデータコンテナをファイルとして保存したり、ファイルをデータコンテナに読み込んだりするファイル入出力機能がある。ファイル入出力には複数の方法があり、目的に応じて使い分けることができる。保存と読み込みはペアとなっており、ある方法で保存したファイルをデータコンテナに読み込むには対応した方法で行う必要がある。

本書では三つの方法について述べる。便宜上これら三つの方法を"テキスト形式"、"バイナリ形式"、"NeXus 形式"と呼ぶこととする。Table 21 にそれぞれの長所、短所を示す。

Table 21 万葉ライブラリのファイル入出力

	長所	短所
テキスト形式	他のソフト・ハードで保存ファイルを容易に利用可能	ElementContainer のみ対応。一部の情報を保存しない
バイナリ形式	保存、読み込みが高速	他のソフト、ハードでの保存ファイルの利用が非常に困難
NeXus 形式 [6)]	他のハードで保存ファイルを利用可能	保存、読み込みが若干遅い。他のソフトでの保存ファイルの利用が困難

4.1. テキスト形式

ElementContainer のヒストグラムデータをカンマやタブで区切った 3 列のテキストファイルとして保存し、また、その形式のファイルを読み込む。色々な表計算ソフトウェアなどで扱いやすく、保存したハードウェア以外でも利用しやすい形式である。ただし、ヘッダ領域の情報や、データ領域にあってもヒストグラムとして設定されてないデータは保存されない。また、保存に用いるコマンドよって、bin の両端の値をそれぞれ保存するか、両端の平均値を保存するかを選ぶことができる。両端の平均値を保存した場合には bin 幅の情報は失われることになる。

4.1.1. 保存

ファイル名の他に、桁数と区切り文字を指定できる。下記のように使用する。

```
import Manyo

ec1 = Manyo.ElementContainer()
ec1.Add("x",[0,1,2,3,4,5])
ec1.Add("y",[100,200,300,400,500])
ec1.Add("e",[10.0,14.14,17.32,20.0,22.36])
ec1.SetKeys( "x", "y", "e" )

# テキスト形式保存
ec1.SaveTextFile("ec1_av.dat", 5, ",") # bin の両端の平均値を保存
ec1.SaveHistTextFile("ec1_hist.dat", 5, ",") # bin の両端を保存。
                                             # 1 列目は 1 行多くなる。
```

この結果として、下記の内容のファイルが保存される。

ec1_av.dat

```
0.5,   100,    10
1.5,   200, 14.14
2.5,   300, 17.32
3.5,   400,    20
4.5,   500, 22.36
```

ec1_hist.dat

```
  0,   100,    10
  1,   200, 14.14
  2,   300, 17.32
  3,   400,    20
  4,   500, 22.36
  5
```

このように、第一引数にファイル名、第二引数に桁数、第三引数に区切り文字を指定する。この例の場合はファイル名が ec1_av.dat、ec1_hist.dat であり、桁数は 5 桁、区切り文字はカンマとなる。SaveTextFile()で保存されるファイルの 1 列目は bin の両端の平均値、2 列目は強度、3 列目はエラーとなる。SaveHistTextFile()では 2 列目、3 列目は同じく強度、エラーであるが 1 列目は bin の低い方の端となり、最後の bin の高い方の端が最後の行となる。

ファイル名は必ず必要だが、桁数と区切り文字は省略してもよい。

ec.SaveTextFile(ファイル名)
ec.SaveTextFile(ファイル名, 桁数)
ec.SaveTextFile(ファイル名, 区切り文字)

省略した場合には、桁数は 10 桁、区切り文字はカンマとなる。また、区切り文字としてタブを用いる場合には\t を区切り文字に指定する。

4.1.2. 読み込み

ElementContainer にカンマ、タブ、空白のいずれかで区切られた 3 列の数値からなるファイルを読み込む。下記のように使用する。

テキスト形式読み込み

```
import Manyo

ec1_av = Manyo.ElementContainer()
ec1_hist = Manyo.ElementContainer()

# テキスト形式読み込み
ec1_av.LoadTextFile("ec1_av.dat")
ec1_hist.LoadTextFile("ec1_hist.dat") # SaveHistTextFile も LoadTextFile で読み込む。
```

このように、第一引数にファイル名を指定する。区切り文字は自動で判別される。SaveTextFile()、SaveHistTextFile()いずれで保存したファイルも LoadTextFile()で読み込む。

4.2. バイナリ形式

データコンテナをバイナリファイルとして保存し、また、その形式のファイルを読み込む。保存、読み込みが最も早い形式であり、データコンテナの全ての情報が保存され、読み込み時にデータコンテナを完全に再現できる。ただし、保存したファイルを他のハードウェアに移した際に読み込めることを保証していない。また、他のソフトウェアで読み込むことも非常に難しい。

4.2.1. 保存

バイナリ形式保存には WriteSerializationFileBinary という名前の機能を、下記のように使用する。

```
import Manyo

ec1 = Manyo.ElementContainer()
ec1.Add("x",[0,1,2,3,4,5])
ec1.Add("y",[100,200,300,400,500])
ec1.Add("e",[10.0,14.14,17.32,20.0,22.36])
ec1.SetKeys( "x", "y", "e" )

# バイナリ形式保存
w = Manyo.WriteSerializationFileBinary("ec1.srlz")
w.Save(ec1)
```

まず、この機能を下から 2 行目のようにファイル名を引数として呼び出す。その戻り値（ここでは w とした）に対し Save()コマンドを実行する。引数には保存したいデータコンテナを指定する。バイナリ形式で保存するデータコンテナは ElementContainer、ElementContainerArray、ElementContainerMatrix のいずれでも良い。

4.2.2. 読み込み

バイナリ形式で保存されたファイルを読み込むには ReadSerializationFileBinary という名前の機能を、下記のように使用する。

```
import Manyo

ec1 = Manyo.ElementContainer()

# バイナリ形式読み込み
r = Manyo.ReadSerializationFileBinary("ec1.srlz")
r.Load(ec1)
```

まず、この機能を下から 2 行目のようにファイル名を引数として呼び出す。その戻り値（ここでは r とした）に対し Load()コマンドを実行する。引数には読み込み先のデータコンテナを指定する。指定するデータコンテナの種類は保存されているものと同じでなければならない。

4.3. NeXus 形式

NeXus とは中性子、X 線、ミュオンのデータをやりとりしやすくするために定められた共通データ形式である。NeXus は HDF5[7]という汎用データ形式に独自のルールを加える形で作られており、様々なハードウェア、ソフトウェアで扱うことができる。例えば Python では h5py というパッケージを利用して扱うことができ、IgorPro、Origin、MATLAB などの商用ソフトウェアでも標準でこの形式のファイルを扱うことができる。また、データコンテナの全ての情報を保存しており、読み込みで完全にデータコンテナを再現できる。ただし、バイナリ形式に比べて入出力が若干遅い。

4.3.1. 保存

NeXus 形式を扱う際には保存でも、読み込みでもまず NeXusFileIO という機能を用いる。下記に保存についての使用例を示す。

```
import Manyo

ec1 = Manyo.ElementContainer()
ec1.Add("x",[0,1,2,3,4,5])
ec1.Add("y",[100,200,300,400,500])
ec1.Add("e",[10.0,14.14,17.32,20.0,22.36])
ec1.SetKeys( "x", "y", "e" )

# NeXus 形式保存
nxio = Manyo.NeXusFileIO()
nxio.Write(ec1, "ec1.nx", "SIK", 0)
```

まず、NeXusFileIO を下から 2 行目のように呼び出す。その戻り値（ここでは nxio とした）に対し Write()コマンドを実行する。第一引数には保存するデータコンテナを指定し、第二引数としてファイル名を指定する。第三引数としては利用者名を指定する。利用者名は文字列であれば特に制限はない。第四引数は 0 もしくは 1 を指定する。1 の場合は圧縮して保存し、0 の場合は圧縮しない。第四引数は省略可能であり、省略すると 1、つまり圧縮しての保存となる。ただし、圧縮は速度が遅くなるため圧縮しないことを推奨する。

4.3.2. 読み込み

下記に読み込みについての使用例を示す。

```
import Manyo

# NeXus 形式読み込み
nxio = Manyo.NeXusFileIO()
ec1 = nxio.ReadElementContainer("ec1.nx")
```

保存の場合と同じく、まず NeXusFileIO を下から 2 行目のように呼び出す。その戻り値に対して読み込み用のコマンドをファイル名を引数として実行する。実行すると戻り値として読み込まれたデータコンテナが得られる。読み込み用のコマンドは ReadElementContainer()、ReadElementContainerArray()、ReadElementContainerMatrix()の三種類あり、保存したデータコンテナの種類に応じたものを使用する必要がある。保存ファイルの圧縮、非圧縮は自動的に判別される。

5. データ処理関数とデータコンテナ

一般にデータ処理を行う場合、様々なパラメータとともに多段階の処理を行う。各段階の処理はそれぞれ個別の関数として準備され、それら関数を順次実行することで処理を進めてゆく。これまで述べてきた万葉ライブラリのデータコンテナとそのデータの扱い方を用いれば、Python 上で必要な機能を持った関数を作成し実行できる。しかし処理速度などの問題から、主なデータ処理の関数は C++言語で書かれているはずである。

万葉ライブラリでは、C++で書かれた関数も基盤部分を共通化することを目指している。なぜなら、この共通化によって

1. 関数の使い方が統一される
2. 独自機能の関数を開発する時に、開発作業が軽減される

といった利点が生じるためである。

ここでは、万葉ライブラリの関数開発には踏み込まないが、データコンテナと万葉ライブラリで推奨する関数との関係を簡単に紹介する。

5.1. 想定するデータ処理

先に述べたような多段階の処理を実行するにあたり、万葉ライブラリが想定するデータ処理とは以下のようなものである（Fig.6 も参照）。

1. データコンテナにデータが入っている
2. 処理のステップで、関数に入力としてデータコンテナ、また処理に必要なパラメータを与える
3. 関数の実行
4. 出力としてデータコンテナを取り出す
5. これを新たに次の処理の入力として用いる（2 へ戻る）

このように、データコンテナを媒介に処理を進めてゆく。万葉ライブラリではこの関数を「演算子」と名付けている。

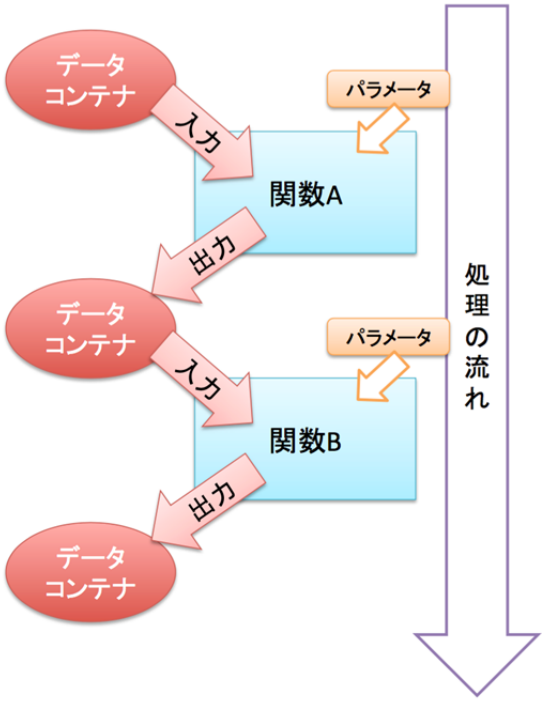

Fig.6 万葉ライブラリが想定するデータ処理（概念図）

5.1.1. データコンテナと処理パラメータ

ここで、データコンテナと処理のパラメータの違いを述べておく。

データコンテナは、処理をされる側のデータであり、同時にそのデータに関する情報（メタデータ）も持つ。一方、処理を行う際に必要となる情報のうち、処理ごとに変更される可能性のあるパラメータと、ある程度固定化されたパラメータが存在する。よって、ある程度固定化されたパラメータは、データコンテナのメタデータとして保存されておくほうが効率が良い。よってあらかじめ

データコンテナには、固定化されているパラメータをヘッダ部に登録しておくべきである。これらの具体的な例は、第 8 章の「MLF における利用例」にて簡単に触れる。

5.2. 関数（演算子）のタイプ

演算子には二つのタイプがある。

1. 入力したデータコンテナがそのまま出力になる（単純演算子：Fig.7 ）

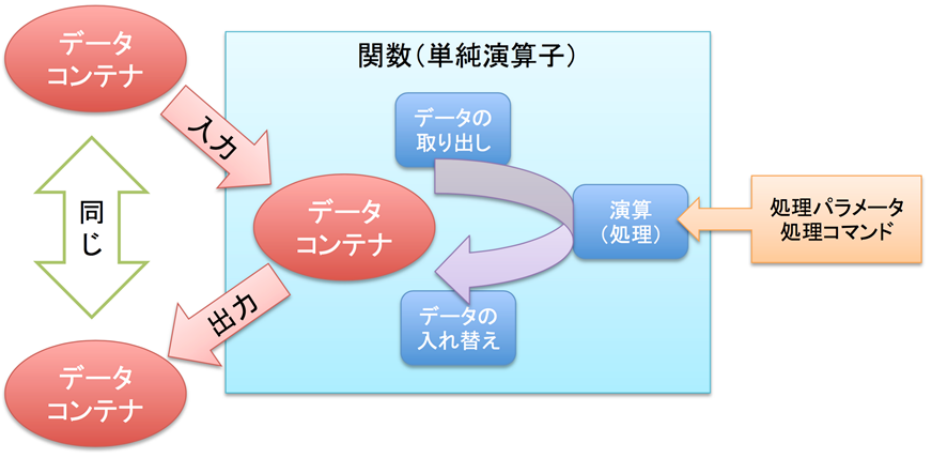

Fig.7 単純演算子

2. 入力したデータコンテナと異なるデータコンテナが出力される（汎用演算子：Fig.8）

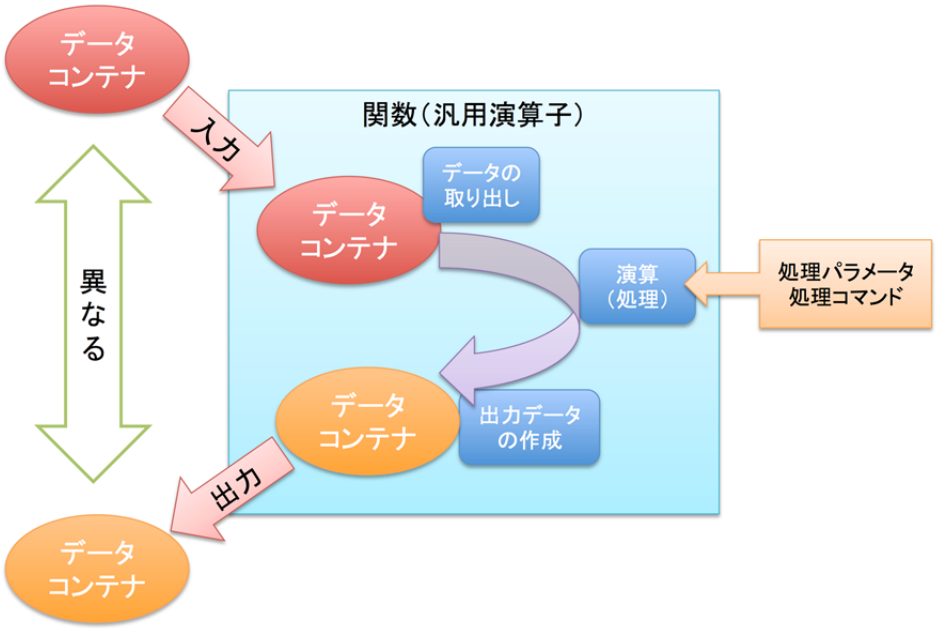

Fig.8 汎用演算子

実際に関数がどのような形で実装されているかは、その機能に依存する。データコンテナの構造が変わらない関数、例えば全データを特定の値で割る、全データの強度をある関数で補正

する、などの場合は 1.単純演算子が用いられる。一方、円環平均をとったり全データの積算値を得たりする場合、すなわち明らかに入力データと出力データの構造が異なる場合は、2.汎用演算子が用いられる。

ただし Table 22 に示すように、単純演算子と汎用演算子とでデータコンテナの入力コマンドが異なっているため、使用時には注意が必要である。

Table 22 関数演算子への入力・出力コマンド

	入力コマンド	出力コマンド
単純演算子	SetTarget(*T)	なし or Put()
汎用演算子	SetInput(T), SetInputP(*T)	Put()

使用例：

単純演算子型の関数の使用例を示す。

```
import Manyo.Utsusemi as mu
kikf = mu.KiKfCorrection()
kikf.SetTarget( dat )  #データの入力
kikf.KiKfCorrect()     #処理の実行 この時点で'dat'は処理済み
del kikf
```

汎用演算子型の関数の使用例を示す。

```
import Manyo.SAS as ms
esub = ms.EcmSubtruction()
esub.SetInputP( dat1 )      #データの入力
esub.Subtract( dat2 )       #パラメータ入力と処理の実行
dat3 = esub.Put()           #結果の出力
del esub
```

6. データコンテナの今後の開発

6.1. 散布データ用コンテナ

6.1.1. 従来のデータコンテナ(ヒストグラム)

現在、万葉ライブラリに実装されている ElementContainer は、 Fig.9(a)に示すようなヒストグラムのデータを扱うことを前提としている。 ElementContainer::SetKeys("x", "y", "e")でヒストグラムデータを 定義する。 その場合の各 vector の要素数は、x.size() = y.size() +1 = e.size() +1 と なっている必要があるが、これは、一次元のカウンターを想定したデータ構造であるためである。

ヒストグラムのデータは、四則演算などの 演算子でエラー伝搬を含めて取り扱うことができている。

6.1.2. 新規の散布データ用コンテナ

Fig.9(b)に示すように散布データ用コンテナに 登録されている vector の要素数は、 x.size()= y.size() = e.size()であり、 ヒストグラムデータとは異なる。

ElementContainer に登録されているデータは この違いによって、自動的にヒストグラムデータか 散布データかの区別を行う。

散布データ用コンテナは、ヒストグラムデータから 「bin 幅で規格化を行う」などして「bin 幅に依存しない 物理量」などに利用を想定している。

6.1.3. 開発項目

散布データの四則演算に適合したエラー伝搬の 計算アルゴリズムの実装を行う必要がある。 ユーザー/ビームラインによって望ましい計算アルゴリズムが 異なる可能性がある。

また２つの散布データを加算する場合でも、 両者で x の値が違う、または、要素数が異なる場合にどのように対応するのかよいか考慮する必要がある。

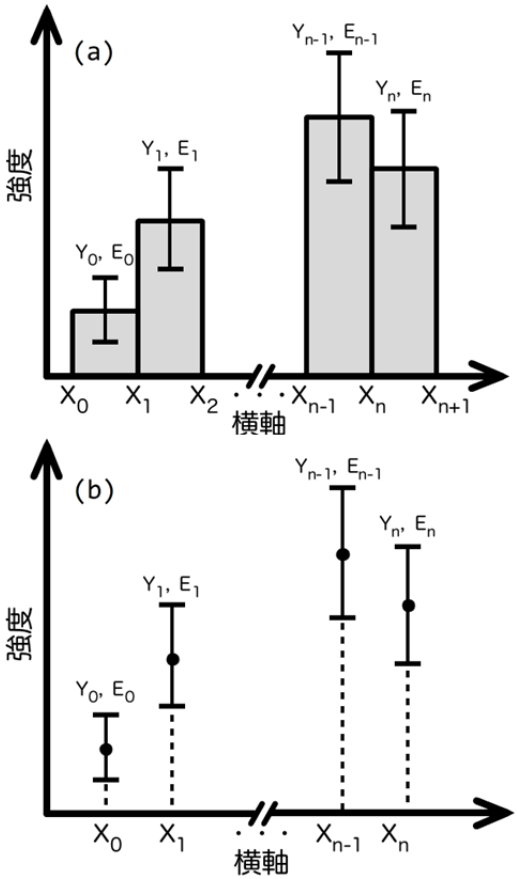

Fig.9 ヒストグラムデータ（a）と散布データ（b）

7. サンプルコード

7.1. はじめに

万葉ライブラリの具体的な使い方を、簡単な使用例やサンプルコードを用いて示す。前章までに ElementContainer などの機能を紹介する際に幾つかのコードを示したが、それらの内容も含め本章にまとめて示す。

基本的には Python での例を示すまた、本文中、あるいは Note で、本書で必要になる Python の基礎についても説明する。Python の基礎については、本書の Appendix A「Python イントロダクション」も参考して欲しい。Keyword では、そのサンプルで特に覚えてもらいたい言葉を取り上げる。

- 本章で示すソースコードはわかりやすさを優先しており、速度、メモリ使用量などの最適化についてはここでは述べない。
- 本章では変数名であることを表すために *斜体*（例： *ec_sample*）を用いている。
- 本章での Python コードで、字下げ（インデント）は空白 4 文字としている。
- Python の文中で#以降はコメントして扱われる。この本書では、例文の操作の結果などをコメントに記すことがある。

7.2. サンプルコード

1. コマンドのヘルプ機能
2. ElementContainer のテキストファイル読み書き
3. ElementContainer の実数配列データの取り出しと表示
4. MLF モジュールの定義定数
5. ElementContainerArray、ElementContainerMatrix へのアクセス
6. EC、ECA、ECM のバイナリファイル読み書き
7. ヘッダ情報へのアクセス
8. ElementContainerArray や Matrix 内 ElementContainer のヘッダ情報による検索
9. Python と C++間のデータ型変換
10. ElementContainer の横軸の入れ替え
11. MLF ターゲット情報の取得
12. 拡張パッケージ「空蝉」によるヒストグラム化とプロトンカレントによる規格化

7.2.1. コマンドのヘルプ機能

Python は、関数の機能を確認する方法として、**help** というコマンドが用意されている。この機能は、万葉ライブラリにも適応されており、下記のようにコマンドの機能などが簡単にではあるが表示される。

```
1    >>> import Manyo
2    >>> ec = Manyo.ElementContainer()
3    >>> help( ec.PutX )
```

【結果】

```
Help on method PutX in module Manyo.core:

PutX(self) method of Manyo.core.ElementContainer instance
    vector< Double > ElementContainer::PutX()

    Return the vector assigned to X-value.
```

7.2.2. ElementContainer のテキストファイル読み書き

二つのテキストファイルからデータを読み込んで、それらを引き算して保存する操作を示す。

```
1    import Manyo
2    ec_sample = Manyo.ElementContainer()
3    ec_sample.LoadTextFile("sample.dat")
4    ec_background = Manyo.ElementContainer()
5    ec_background.LoadTextFile("background.dat")
6    ec_result = ec_sample - ec_background
7    ec_result.SaveTextFile("result.dat")
```

- 1 行目: 万葉ライブラリを使うために必要な命令文。以降では記述を省略するが、これが行われているものとする。
- 2 行目: データコンテナ(*ec_sample*)を準備する。
- 3 行目: このデータコンテナに"sample.dat"というファイルからデータを読み込む。
- 4, 5 行目: もう一つデータコンテナ(*ec_background*)を準備して、そこに"background.dat"というファイルからデータを読み込む。
- 6 行目: 二つのデータの引き算をし、その結果を新たなデータコンテナ(*ec_result*)に入れる。
- 7 行目: この結果をファイルに保存する。

result.dat のエラーの列を見て、誤差伝搬がなされていることを確認する。

7.2.2.1. Keyword

ElementContainer

万葉ライブラリのデータを収めるための入れ物、データコンテナ。基本的に実数配列(vector)が入る。万葉ライブラリの根幹の一つ。ElementContainer 同士の演算をサポートしている。万葉ライブラリは ElementContainer に始まり ElementContainer に終わる。本稿ではしばしば *EC* と略す。HeaderBase も参照。より詳しくは第 2 章「データコンテナ」を参照。

vector

C++のデータ型の一種で、C 言語での配列を便利にしたようなもの。本書第 2 章「データコンテナ」に若干の説明を記した。詳しくは C++について書かれた書籍等を参照のこと。

7.2.2.2. Note

- ファイル形式は、3 列（X、強度、エラー)、カンマ区切りを既定としている。
- より詳しい説明は「ファイル入出力」の章に示した。
- この保存方法では X、強度、エラー以外の情報は保存されない。それらも含めて保存する方法は「EC、ECA、ECM のバイナリファイル読み書き」に示す。

7.2.3. ElementContainer の実数配列データの取り出しと表示

ec という ElementContainer から色々なデータを取り出す方法を示す。 ここでは以下の例の前に ec にデータが入っているものとする。

```
1    x = ec.PutX()
2    intensity = ec.PutY()
3    error = ec.PutE()
4    header = ec.PutHeader()
5    pixel_id = header.PutInt4("PIXELID")
6    pixel_position = header.PutDoubleVector("PixelPosition")
7    ec.Dump()
```

- 1 行目: *ec* から横軸の vector を取り出し、*x* に代入する。
- 2 行目: *ec* から強度の vector を取り出し、*intensity* に代入する。
- 3 行目: *ec* から誤差の vector を取り出し、*error* に代入する。
- 4 行目: *ec* から HeaderBase を取り出し、*header* に代入する。
- 5 行目: *header* からピクセル ID を取り出し、*pixel_id* に代入する。PXIELID はピクセル ID を取り出すための Key。
- 6 行目: *header* からピクセル位置を取り出し、*pixel_position* に代入する。PixelPosition はピクセル位置を取り出すための Key。
- 7 行目: *ec* の内容全てを表示する。

7.2.3.1. Keyword

HeaderBase

万葉ライブラリのデータを収めるための入れ物。ElementContainer と異なり、様々なデータを入れられる。ElementContainer には必ず HeaderBase が付加されている。HeaderBase についても第 2 章「データコンテナ」の章でより詳しく述べた。

Dump

ElementContainer や HeaderBase についてこれを行うと、それぞれの内容を表示する。

7.2.4. MLF モジュールの定義定数

MLF モジュールには中性子の質量やボルツマン定数など幾つかの物理定数、および長さや重さなどの変換係数が定義されており、使用することができる。

```
1    import math
2    import Manyo.MLF as mm
3
4    Neutron_mass = mm.MLF_MASS_NEUTRON
5    print "Neutron Math = ",Neutron_mass
6    Neutron_E = 100.0 #[meV]
7    Neutron_E = Neutron_E*mm.MLF_MEV2J #[meV]->[J]
8    Neutron_v = math.sqrt( Neutron_E / Neutron_mass * 2.0 )
9    print " the Neutron speed at 100.0 meV  = %g [m/s] "%( Neutron_v )
```

- 4 行目: *Neutron_mass* に MLF モジュールの中性子質量定数（MLF_MASS_NEUTRON）を代入する。
- 5 行目: *Neutron_mass* を表示する。
- 6 行目: *Neutron_E* に中性子の速度として 100.0 を代入する（100.0 meV の意味）。
- 7 行目: *Neutron_E* の単位を[meV]から[J]に変換する定数（MLF_MEV2J）で変換する。
- 8 行目: 中性子速度を計算し、 *Neutron_v* に代入する(1/2*Mn*v*v から)。
- 9 行目: 中性子速度の計算結果を表示する。

その他の定数に関しては、本書の Appendix B「万葉ライブラリ関数リファレンス」を参照のこと。

7.2.5. ElementContainerArray、ElementContainerMatrix へのアクセス

ecm という ElementContainerMatrix と、 *ec_background* という ElementContainer があるとしよう。*ecm* 中の一つ目の ElementContainerArray に収められている ElementContainer 全てについて、*ec_background* を引いた結果を新たに作った *eca2* に収め、それを *ecm* に追加する。

```
1    eca1 = ecm.Put(0)
2    n = eca1.PutSize()
3    eca2 = Manyo.ElementContainerArray()
4    for i in range(n):
5        ec = eca1.Put(i)
6        ec -= ec_background
7        eca2.Add(ec)
8    ecm.Add(eca2)
```

- 1 行目： *ecm* から一つ目の ElementContainerArray を取り出し、 *eca1* に代入する。
- 2 行目： *eca1* 内の ElementContainer の数を *n* に代入する。
- 3 行目： *eca2* という空の ElementContainerArray を作る。
- 4 行目： *n* 回の for ループ。
- 5 行目： *eca1* から *i* 番の ElementContainer を取り出し、 *ec* に代入する。
- 6 行目： *ec* から *ec_background* を引き、その結果を *ec* とする。
- 7 行目： *eca2* に *ec* を追加する。
- 8 行目： *ecm* に *eca2* を追加する。

7.2.5.1. Keyword

ElementContainerArray
: ElementContainer を収めるためのデータコンテナ。ElementContainerArray 自身も HeaderBase を持つ。本節ではしばしば *ECA* と略す。第 2 章「データコンテナ」を参照。

ElementContainerMatrix
: ElementContainerArray を収めるためのデータコンテナ。ElementContainerMatrix 自身も HeaderBase を持つ。本節ではしばしば *ECM* と略す。第 2 章「データコンテナ」を参照。

range
: Python の関数。引数に整数を指定することで、整数のリストを返す。例えば次のようになる。

```
1    range(3) # [0, 1, 2]
2    range(2,5) # [2, 3, 4]
3    range(10, 30, 5) # [10, 15, 20, 25]
```

7.2.5.2. Note

- Python での for ループの基礎を例を挙げて示す。

```
1        for i in range(2,5):
2            print "i=",i
```

1 行めの *range(2,5)* は、先に示したように、 *[2,3,4]* を意味する。よってこのコードは

```
1        for i in [2,3,4]:
```

```
2           print "i=",i
```

という意味にになる。まず 1 行目の *range(2,5)* すなわち *[2,3,4]* の一つ目が *i* へ代入され、6 行目のブロック（ **for** よりもインデントされた行 ）に渡される。ブロック内の処理が終わると再び **for** の行へ戻ってくるが、次は *[2,3,4]* の二つ目が *i* へ代入されてブロックへ流される。これを繰り返し、 *[2,3,4]* のすべての値が *i* に代入し終わるまで処理が続く。

このように、いわゆる C 言語の for 文の for (int i=0; i<n; i++) と同じことが、この *range(n)* を用いれば使用できる。 また、あらかじめ与えておいた Python リストの通りに **for** 文を実行できるので、不規則な数列でも（ [2,53,4,100,1,...] ）順に処理されるのみならず、文字列の配列すら同様に渡すことができる。

```
1       ll = [ "an Apple", "a Banana", "an Orange" ]
2       for fruit in ll:
3           print "I Love ",fruit," too!"
```

また、配列の長さを知りたい場合、 **リスト** であれば len(x_list)とすると得られる。

7.2.6. EC、ECA、ECM のバイナリファイル読み書き

EC、ECA、ECM をバイナリファイルで入出力する方法を示す。

ここでは、*filepath_ecm* という **ECM** をバイナリで保存したファイルのパスがあり、これを読み込んでその中の **ECA** のうちヘッダの PSDID が 16～31 のものだけをそれぞれバイナリで保存する処理を示す。

```
 1  filepath_ecm = "/home/hogehoge/sample.srlz"
 2  ecm = Manyo.ElementContainerMatrix()
 3  read_binary = Manyo.ReadSerializationFileBinary(filepath_ecm)
 4  read_binary.Load(ecm)
 5  del read_binary
 6  filepath_template = "psdid_%04d.srlz"
 7  for i in range(ecm.PutSize()):
 8      eca = ecm.Put(i)
 9      psd_id = eca.PutHeader().PutInt4("PSDID")
10      if psd_id in range(16,32):
11          out_filepath= filepath_template%psd_id
12          write_binary= Manyo.WriteSerializationFileBinary(out_filepath)
13          write_binary.Save(eca)
14          del write_binary
```

- 1 行目： *filepath_ecm* にバイナリファイルの場所を代入する。
- 2 行目： 空っぽの ElementContainerMatrix を作成し、*ecm* と名付ける。
- 3 行目： バイナリファイル読み込み機能である *ReadSerializationFileBinary* を作成し、*read_binary* と名付ける。その時に引数として *ecm_filepath* を指定する必要があり、これでファイルが開かれる。
- 4 行目： 空っぽの *ecm* にファイルの中身を代入するコマンドである、 *Load* コマンドを *ecm* を引数にして実行する。
- 5 行目： 不要となったバイナリファイル読み込みコマンドを消去する。
- 6 行目： 次に、データを書き出すためのファイル名のテンプレートを用意する。
- 7 行目： ループ処理の開始。ここでは、 *i* を 0 から *ecm* の内部コンテナの数-1 だけ変化させるループとなる。
- 8 行目： *i* 番目の ElementContainerArray を取り出す。
- 9 行目： 取り出した ElementContainerArray ののヘッダを取り出し、"PSDID"の情報を整数で引き出し *psd_id* に代入する。
- 10 行目： もし、 *psd_id* の値が、16～32 であれば、次のブロックを実行する。ここで psd_id in range(16,32)は Python でよく使われる記法で、psd_id in [16,17,18,19,20,...31]と同じ意味であり、すなわち psd_id がリスト[16,17,18,19,20,...31]に入っていれば True となるものである。
- 11 行目： 6 行目のファイル名テンプレートを利用して、 *psd_id* が挿入されたファイル名を *out_filepath* に代入する。
- 12 行目： バイナリファイル保存機能である *WriteSerializationFileBinary* を作成し、*write_binary* という名前をつける。この時に引数として先ほど作成した保存ファイル名 *out_filepath* 利用している。
- 13 行目： 保存を実行する。 *Save* コマンドは引数として、ElementContainer、ElementContainerArray、ElementContainerMatrix などを与えることができるが、今回は ElementContainerArray であり、*ecm.Put(i)* として *ecm* の *i* 番目の ElementContainerArray を保存対象としている。

- 14 行目： 不要なコマンドを消去する。

7.2.6.1. Note

- **バイナリファイルへの読み書き**

  このバイナルファイルへの書き出しは、テキストファイルでの入出力より処理が早い。また、テキストファイルの場合と異なり、ヘッダ情報も含めた全情報を保存、読み込みできる。ただし、書き出されたファイルには環境依存性、すなわち、ハードウェア（PC の CPU）やソフトウェア（特にウィンドウズや Linux など）の違いによって、互換性がなくなる可能性が高いが、別の環境へ持ち出したり持ち込んだりしないのであれば問題はない。したがって処理の途中経過のデータを一時的にファイルに保存する際に最も力を発揮する。

- Python リストの役割

  Python に置いて、「変数 in リスト」、すなわち

  ```
  psd_id in [1,2,3,4,5]
  ```

  という表記は、「変数の値と同じものがリストの中にあるか」のチェックに使用される。これ自体が True や False という値を持ち、if 文なのどの条件に使用される。

- 不要な関数、コマンドは消去

  Python は、メモリをよく消費する。また C++のコードを使うためにメモリの管理も難しくなっている。よって使い終わった関数やコマンドはできるだけすぐに消去することをお勧めする。

7.2.7. コンテナのヘッダ情報へのアクセス

ヘッダは、HeaderBase と呼ばれるデータコンテナが利用されている。ElementContainer には、AddToHeader という入力用コマンドと PutHeader()もしくは InputHeader()という取り出し・置き換え（上書き）コマンドがある。

コンテナのヘッダに情報を加える、取り出す場合の例を以下に示す。

```
 1   import Manyo
 2   ec=Manyo.ElementContainer()
 3   ec.AddToHeader( "L1", 20.193 )
 4   ec.AddToHeader( "NumOfPixels", 100 )
 5   ec.AddToHeader( "Sample", "CuGeO3" )
 6
 7   hh = ec.PutHeader()
 8   L1=hh.PutDouble( "L1" )
 9   NumOfPixels=hh.PutInt4( "NumOfPixels" )
10   SampleName=hh.PutString( "Sample" )
11   print "L1=",L1
12   print "NumOfPixels=",NumOfPixels
13   print "Sample Name=",SampleName
```

- 2 行目： 空っぽの ElementContainer を作成し、*ec* と名付ける。
- 3 行目： *ec* のヘッダに "L1" というキーで実数を収納する。
- 4 行目： 同様に *ec* のヘッダに "NumOfPixels" というキーで整数を収納する。
- 5 行目： 同様に *ec* のヘッダに "Sample" というキーで文字列を収納する。
- 6 行目： 空行
- 7 行目： *ec* のヘッダのコピーを取り出す。
- 8 行目： 取り出したヘッダから "L1" というキーの情報を実数として取り出し *L1* に代入する。
- 9 行目： 取り出したヘッダから "NumOfPixels" というキーの情報を整数として取り出し *NumOfPixels* に代入する。
- 10 行目： 取り出したヘッダから "Sample" というキーの情報を文字列として取り出し *SampleName* に代入する。
- 11-13 行目： 代入された３つの変数を表示する。

コンテナのヘッダの情報を書き換える場合の例を以下に示す。

```
 1   hh = ec.PutHeader()
 2   if hh.CheckKey( "L1" )==1:
 3       hh.Remove( "L1" )
 4   hh.Add( "L1", 30.0 )
 5   ec.InputHeader( hh )
 6
 7   hh_new = ec.PutHeader()
 8   L1=hh_new.PutDouble( "L1" )
 9   NumOfPixels=hh_new.PutInt4( "NumOfPixels" )
10   SampleName=hh_new.PutString( "Sample" )
11   print "L1=",L1
12   print "NumOfPixels=",NumOfPixels
13   print "Sample Name=",SampleName
```

- 1 行目： *ec* のヘッダを取り出す（一つ前のスクリプトと同じ *ec* ）
- 2 行目： *ec* のヘッダに "L1" というキーがあるかどうかを確認し、
- 3 行目： もしキーがあれば "L1" の情報を削除する。
- 4 行目： *ec* のヘッダに "L1" というキーで実数を収める（ "L1"の情報の書き換え)。
- 5 行目： *ec* のヘッダを上で書き換えたヘッダで置き換える。
- 6 行目： 空行
- 7 行目： 改めて *ec* のヘッダのコピーを取り出し、 *hh_new* とする。
- 8 行目： 取り出したヘッダ *hh_new* から "L1" というキーの情報を実数として取り出し *L1* に代入する。
- 9 行目： 取り出したヘッダ *hh_new* から "NumOfPixels" というキーの情報を整数として取り出し *NumOfPixels* に代入する。
- 10 行目： 取り出したヘッダ *hh_new* から "Sample" というキーの情報を文字列として取り出し *SampleName* に代入する。
- 11-12 行目： 代入された３つの変数を表示する。

7.2.7.1. Note

- PutHeaer()と InputHeader()

  AddToHeader は新しい情報を加えることしかできないため、ヘッダから情報を引き出したり変更したりするには、一旦 HeaderBase を取り出す必要がある。まず PutHeader()にて HeaderBase のコピーを取り出し、そこからさらに情報を引き出すことになる。あるいはヘッダ情報を編集（追加、置き換え）しいなら、PutHeaer()で取り出した後で編集し、InputHeader(HeaderBase)で ElementContainer のヘッダを置き換える必要がある。

- ヘッダ情報の置き換え（上書き）

  コマンドの紹介の意味もあり、スクリプトでは *CheckKey* 及び *Remove* を使って情報を置き換えたが、そのまま *OverWrite* というコマンドもある。Appendix B の「万葉ライブラリ関数リファレンス」を参照のこと。

7.2.8. ElementContainerArray や Matrix 内 ElementContainer のヘッダ情報による検索

ElementContainerArray や ElementContainerMatrix に収められている ElementContainer や ElementContainerArray のヘッダ内容に対して検索をかけ、適合するコンテナのみを取り出す方法を示す。例として、以下のようなデータがあるとする。

- ElementContainer には位置敏感型検出器（PSD）の検出単位（Pixel）ごとのデータ
  - ヘッダには *PIXELID* というキーで検出単位の ID 番号（整数値）が収められている。
- ElementContainerArray には、PSD ごとにまとめられた複数の ElementContainer
- ElementContainerMatrix には複数の ElementContainerArray
- ElementContainerMatrix は *dat* という変数名

ここから特定の検出単位の ID 番号（キーは *PIXELID* ）の ElementContainer を取り出し、新しく用意した ElementContainerArray に追加する作業を行う。

Python だけを利用したスクリプトを以下に示す。

```
 1  target_id = 245
 2  result_eca = Manyo.ElementContainerArray()
 3  num_of_array = dat.PutSize()
 4  for i in range( num_of_array ):
 5      eca = ecm(i)
 6      num_of_hist = eca.PutSize()
 7      for j in range( num_of_hist ):
 8          header = eca(j).PutHeader()
 9          if header.CheckKey( "PIXELID" )==1:
10              pixel_id = header.PutInt4( "PIXELID" )
11              if pixel_id == target_id:
12                  result_eca.Add( eca.Put(j) )
13
14  if result_eca.PutSize()==0:
15      print ( "We cannot found ElementContainer with id=",target_id )
```

- 1 行目： *target_id* に 245 を代入する（これを目的の ID とする）。
- 2 行目： *result_eca* に空の ElementContainerArray を代入しておく（これに見つけた ElementContainer を入れる）。
- 3 行目： *num_of_array* に ElementContainerMatrix に含まれている ElementContainer の個数を代入する。
- 4 行目： ループ処理の開始。ここでは、 *i* を 0 から *ecm* の内部コンテナの数 *num_of_array* だけ変化させるループとなる。
- 5 行目： *eca* に *ecm* の *i* 番目のコンテナ（そのもの）を取り出す（ Put(i) を使わない方法：Note を参照）。
- 6 行目： *num_of_hist* に *eca* に含まれている ElementContainer の個数を代入する。
- 7 行目： ループ処理の開始。ここでは、 *j* を 0 から *eca* の内部コンテナの数 *num_of_hist* だけ変化させるループとなる。
- 8 行目： *header* に *eca* の *j* 番目のコンテナのヘッダを取り出す。
- 9 行目： もし、 *header* 内に *PIXELID* と言うキーがあれば
- 10 行目： *pixel_id* にそのキーの値を整数で取り出して代入する。

- 11 行目： もし *pixel_id* の値が *target_id* と同じであれば
- 12 行目： *result_eca* に、この見つけたコンテナ（ *eca* の *j* 番目のコンテナ）を加える。
- 13 行目： 空行
- 14 行目： もし *result_eca* が空であったら（検索に何一つかからなかったら）
- 15 行目： メッセージを表示する。

このように Python コードと ElementContainer やヘッダの扱いのみでコードを書くことができる。しかしループを行うのは Python では遅いし、検索ごとにこのコードを書くのも大変であるので、万葉ライブラリに入っている **SearchInHeader** 関数を用いることも可能である。その例を示す。

SearchInHeader を利用したスクリプトを以下に示す。

```
1    target_id = 245
2    result_eca = None
3    SIH = Manyo.SearchInHeader( dat )
4    if SIH.Search( "PIXELID", target_id ):
5        result_eca = SIH.PutResultAsArray()
6
7    if result_eca==None or result_eca.PutSize()==0:
8        print ( "We cannot found ElementContainer with id=",target_id )
```

- 1 行目： *target_id* に 245 を代入する（これを目的の ID とする）。
- 2 行目： *result_eca* に None を代入しておく（これに見つけた ElementContainer が入った ElementContainerArra を入れるつもり。見つからなければ None のまま）。
- 3 行目： *SIH* に *SearchInHeader* 関数を準備する（引数に目的の ElementContainerMatrix の *dat* を与える）。
- 4 行目： キー "PIXELID" の値が *target_id* である ElementContainer の検索を実行し、実行自体に問題がなかったなら
- 5 行目： 検索結果を *result_eca* に代入する（ElementContainerArray に見つかった ElementContainer が入る）。
- 6 行目： 空行
- 7 行目： もし *result_eca* が *None* もしくは、サイズが 0 だったら（検索にかからなかったら）
- 8 行目： メッセージを表示する。

7.2.8.1. Note

- ElementContainerArray からの高速な ElementContainer の取り出し
  ElementContainerArray には *Put* と言うコマンドがあり、収納している。ElementContainer のコピーを取り出すことができる。しかしコピーを取るという作業はそれだけでも時間がかかるため、今回のようにヘッダを見るだけの作業で使用すると大幅に時間がかかる。そこで ElementContaierArray には、コピーではなくそのものを取り出すやり方（C++言語で言うところのポインタ）がある。ただしそのものを取り出す場合、高速ではあるが取り扱いには厳重に注意する必要がある。Table 23 にその差を示しておく。

Table 23 ElementContainerArray からの ElementContainer 取り出し

コマンド	作用	速度	危険度
eca.Put(i)	i 番目のコンテナ (ElementContainer）のコピーを取り出す	低速	低い（取り出したコンテナを消しても大丈夫）
eca(i)	i 番目のコンテナそのもの（ポインタ）を取り出す	高速	高い（取り出したコンテナを消すと元の ElementContainerArray も壊れる）

- ElementContainerMatrix からの高速な ElementContainer, -Array の取り出し
  ElementContainerMatrix も *Put* で ElementContainerArray の取り出しを行うが、そのものを取り出す方法もある。また直接に ElementContainer そのものを取り出す方法もある。どちらも高速であるが、大変に危険であるため取り扱いには注意が必要である。その違いを Table 24 に示す。

Table 24 ElementContainerMatrix からの ElementContainerArray 取り出し

コマンド	作用	速度	危険度
eca.Put(i).Put(j)	i 番目の ElementContainerArray の j 番目の ElementContainer のコピーを取り出す	低速	低い（取り出したコンテナを消しても大丈夫）
eca(i,j)	i 番目の ElementContainerArray の j 番目の ElementContainer そのもの（ポインタ）を取り出す	高速	高い（取り出したコンテナを消すと 元の ElementContainerArray も壊れる）

7.2.9. Python と C++間のデータ型変換

Python で配列を扱うために用いる **リスト** から、C++で配列を扱うために用いる **vector** への変換、及びその逆の変換を行う方法を示す。

事前に、数値の **リスト** である *x_list*、数値の **vector** である *y_vector* 、及び、文字列の **リスト** である *string_list* が用意されているものとする。

```
1   print type(x_list)
2   x_vector = Manyo.ListToDoubleVector(x_list)
3   print type(x_vector)
4   y_list = Manyo.DoubleVectorToList(y_vector)
5   string_vector = Manyo.StringVector()
6   for s in string_list:
7       string_vector.push_back(s)
8   print type(string_vector)
```

- 1 行目： *x_list* のデータ型を表示する。ここでは **list** と表示される。
- 2 行目： **リスト** を **vector** に変換し *x_vector* に代入する。厳密には **vector<Double>** に変換している。
- 3 行目： *x_vector* のデータ型を表示する。
- 4 行目： **vector** を **リスト** に変換し、 *y_list* に代入する。
- 5 行目：空の vector<string> を作る。
- 6, 7 行目： *string_list* から一個ずつ文字列を *s* に取り出し、 *string_vector* に追加していく。
- 8 行目： *string_vector* のデータ型を表示する。

万葉ライブラリは C++で作られており、Python 上で作成したデータをそのまま万葉ライブラリで使うことができない場合がある。例えば **リスト** は本来ならそのままでは ElementContainer に入力できない。これは、**list** は Python のデータ型であり、この類のデータは万葉ライブラリでは **vector** で扱わなければならないからである。それができるのは、ElementContainer 内部でリストから **vector** への変換を行っているからである。そのほか、ElementContainer やヘッダから直接に配列を取り出す場合も **リスト** ではなく **vector** で取り出される（ Put(KEY) コマンドなど)。このような場合には、ここにあるような操作でデータ型変換を行う必要がある。

7.2.9.1. Keyword

list

Python でのデータ列の入れ物の一種。万葉ライブラリ環境では *intensity* という vector に対して、list(*intensity*)とすると list に変換される。7 行目では、vector のままでは表示できないために vector から list に変換された結果を表示している。

type

引数のデータ型を文字列で返す関数。なお、ある変数が、あるデータ型であるかを調べるためには、 **isinstance** という真偽を返す関数がある。これは例えば、 *ec* が ElementContainer であるかを調べるためには次のように使う。

```
1   isinstance(ec, Manyo.ElementContainer)
```

7.2.9.2. Note

第 2 章「データコンテナ」も参照のこと。

7.2.10. ElementContainer の横軸を入れ替える

ElementContainer に入っているヒストグラムデータの横軸を取り出して、他の単位への変換を行う。このサンプルでは、以下のような情報が入った ElemnetContianer がすでに *ec* という名前で存在するものとする。

軸	内容
X 軸	Time of Flight [micro-sec]
Y 値	強度 [counts]
E 値	エラー

このスクリプトでは、X 軸の配列を取り出すのに、PutXList や PutX ではなく、Put を利用した。これは ElementContainer に収められている複数の実数配列（ vector<double> ）へのアクセスの一つの方法である。計算された新しい横軸も vector<double> を用いて作成し、ElementContainer に加えている。また汎用性を高めるために、用意された *ec* のヒストグラム情報の **キーが具体的にわからなく** ても行えるように作成した。

```
 1  x_key = ec.PutXKey()
 2  y_key = ec.PutYKey()
 3  e_key = ec.PutEKey()
 4  tof_vector = ec.Put( x_key )  # ec.PutX()と同じ
 5
 6  d_list_vec = Manyo.MakeDoubleVector()
 7  for i in range( tof_vector.size() ):
 8      d_list_vec.append( (tof_vector[i]-7.7)/15040.2 )
 9
10  new_x_key = "d-spacing"
11  ec.Add( new_x_key, d_list_vec )
12  ec.SetKeys(new_x_key, y_key, e_key)
```

- 1 行目： *ec* の X 軸のキーを取り出して *x_key* に代入する。
- 2 行目： *ec* の Y 軸のキーを取り出して *y_key* に代入する。
- 3 行目： *ec* の E 軸のキーを取り出して *e_key* に代入する。
- 4 行目： *ec* から *x_key* をキーとする配列（ヒストグラムの X 軸）を **vector** 形式で取り出し *tof_vector* に代入する。
- 5 行目： 空行。
- 6 行目： *d_list_vec* という空の C++の実数配列である vector<double> を作る。
- 7 行目： *tof_vector* のサイズ（ *tof_vector.size()* ）を用いて、for 文を *i* を使ってループさせる。
- 8 行目： *tof_vector* の *i* 番目の値を取り出し、7.7 引いて 15040.2 で割って d 値を計算し、*d_list_vec* に追加する。これを *tof_vector.size()* 回だけ繰り返す。
- 9 行目： 空行。
- 10 行目： 新しく利用する x 軸のキーとして、 *new_x_key* を用意し、"d-spacing"を代入する。
- 11 行目： *ec* に *new_x_key* というキーで *d_list_vec* を追加する。
- 10 行目： *ec* のデータを *new_x_key*, *y_key*, *e_key* の配列を用いてヒストグラムとして構成する。

7.2.10.1. Keyword

vector の作成

C++の配列である vector は、万葉ライブラリでは主に実数配列、整数配列、正の整数配列、文字列の配列に用いられる。表記は vector<double>, vector<Int4>, vector<UInt4>, vector<string> である。

vector のサイズ

C++の汎用配列である vector のサイズを知るのは *.size* というコマンドである。

7.2.10.2. Note

- ElementContainer をヒストグラムとして使う利点

  ElementContainer の構成のうち、どのような配列がどのようなキーで登録されているかは一般的には不明である。しかし、ヒストグラムとして設定する（**SetKeys(X-KEY, Y-KEY, E-KEY)**）ことで、特にスクリプト中にキーをあらわに書くことなくヒストグラムデータの要素にアクセスできる。このように **ElementContainer を新しく作ったり、処理を行ったら、必ず SetKeys を行うこと**。

- 横軸の問題

  サンプルでは、Time of Flight [micro-sec]の値を単に d 値に変換しただけであった。よって横軸の bin 区切りは等間隔ではないことに注意。ただし強度はヒストグラムの性質上正確なものである。等間隔にしたい場合は、新しい横軸の配列を C++の実数配列 vector<double> で作成して、それを用いて *Binning* を実行する必要がある。

```
1   import Manyo
2   new_x_vec=Manyo.MakeDoubleVector()
3   for i in range(1000):
4       new_x_vec.append( float(i)*0.01 )
5   ec.Binning( new_x_vec )
```

7.2.11. MLF ターゲット情報の取得

ここでは中性子源直前の陽子数計測器の積算値を表示する方法を示す。ただしこのコマンドは J-PARC サイト内の LAN 接続を行う必要があり、通常は機能しない。ここではあくまで使用例だけを示し、具体的な使い方などの記述は省く。

```
1    HOST = 'www-cont.j-parc.jp'
2    API = Manyo.AcquireNeutronSourceTextInformation(HOST)
3    begin = '2011/03/10 12:00:00'
4    end = '2011/03/10 13:00:00'
5    values = API.PutValueInformation('CT9', begin, end, -1)
6    print values[0] # values は vector<Double>で返ってくる。単位は テラ個。
```

- 1 行目: 中性子源データベースのホスト名を変数 *HOST* に代入する。
- 2 行目: MLF ターゲット情報取得用のインスタンス(*API*)を作る。引数は先ほどの *HOST* を指定する。
- 3 行目: 開始時刻を変数 *begin* に代入する。
- 4 行目: 終了時刻を変数 *end* に代入する。
- 5 行目: *API* の PutValueInformation を使って、陽子数を取り出し *values* に代入する。'CT9' は中性子源直前の陽子数計測器を示す。最後の引数 -1 は begin から end までの総和を返すという指示。
- 6 行目: values は vector<Double>になっており、今回は中身は一つだけ(0 番目)でありそれを表示する。

7.2.12. 拡張パッケージ「空蝉」によるヒストグラム化とプロトンカレントによる規格化

ここでは拡張パッケージ「空蝉」によるヒストグラム化を例として、具体的なスクリプトの実用例を示す。ただし、このスクリプトの実行には「空蝉」のインストールが必要であり本書の範囲を逸脱するため、参考として見ていただきたい。使用するのは万葉ライブラリ 0.3 と空蝉（リリース 0.3.3.0）である。

本スクリプトの主な内容

- ヒストグラム化に必要なパラメータファイル（あらかじめ作成済み）を指定し
- イベントデータのヒストグラム化を行い ElementContainerMatrix に収納し
- ElementContainerMatrix のヘッダ情報から測定開始時刻と終了時刻情報を取り出し
- それを用いてプロトンカレントを線源のサーバーから取得し、
- その値でヒストグラムの強度をノーマライズする。

```
 1   import Manyo as mm
 2   import Manyo.Utsusemi as mu
 3
 4   wfile = "/home/hoge/xml/WiringInfo.xml"
 5   dfile = "/home/hoge/xml/DetectorInfo.xml"
 6   runNo = 100
 7   dd=mu.UtsusemiEventDataConverterNeunet()
 8   dd.LoadParamFiles( wfile, dfile )
 9   dd.SetHistAllocation()
10
11   ecm = mm.ElementContainerMatrix()
```

```
12  dd.SetElementContainerMatrix( ecm, runNo, "/data", "" )
13
14  date_v = ecm.PutHeader().PutDoubleVector("MEASPERIOD")
15  HOST = 'www-cont.j-parc.jp'
16  API = Manyo.AcquireNeutronSourceTextInformation(HOST)
17  begin = "%4d/%02d/%02d %02d:%02d:%02d" % ( date_v[0], date_v[1], date_
18  v[2], date_v[3], date_v[4], date_v[5] )
    end = "%4d/%02d/%02d %02d:%02d:%02d" % ( date_v[7], date_v[8], date_v
19  [9], date_v[10], date_v[11], date_v[12] )
20  values = API.PutValueInformation('CT9', begin, end, -1)

21  ecm.MulMySelf( 1.0/values[0] )
```

- 1 行目： 万葉ライブラリのコア関数の読み込み
- 2 行目： 万葉ライブラリの空蝉関数の読み込み
- 3 行目： 空行
- 4 行目： Wiring Info ファイルの場所の指定（ *wfile* に代入）
- 5 行目： Detector Info ファイルの場所の指定（ *dfile* に代入）
- 6 行目： *runNo* に目的の Run Number を代入
- 7 行目： ヒストグラム化を行う関数の準備
- 8 行目： WiringInfo と DetectorInfo のファイルを読み込み
- 9 行目： WiringInfo の情報から必要なヒストグラムを内部に準備する。
- 10 行目： 空行
- 11 行目： 空の ElementContainerMatrix（ *ecm* ）の用意
- 12 行目： 与えられた *RunNo* やデータの場所の引数（ "/data"など）を利用しイベントデータを読み込んで *ecm* へ代入する。
- 13 行目： 空行
- 14 行目： ElementContainerMatrix のヘッダにはイベントデータから読み出した測定開始時刻と終了時刻が入っているので、それを取り出す（キー： *MEASPERIOD* ）。
- 15 行目： 中性子源データベースのホスト名を変数 HOST に代入する。
- 16 行目： MLF ターゲット情報取得用関数(API)を準備する。引数は先ほどの HOST を指定する。
- 17 行目： 開始時刻を変数 *begin* に代入する。
- 18 行目： 終了時刻を変数 *end* に代入する。
- 19 行目： API の PutValueInformation を使って、陽子数を取り出し *values* に代入する。 'CT9'は中性子源直前の陽子数計測器を示す。 最後の引数 -1 は *begin* から *end* までの総和を返すという指示。
- 20 行目： 空行
- 21 行目： ElementContainerMatrix の強度を陽子数(*values*[0])で割る。*values* は vector<Double>になっている。今回は陽子数だけが必要なため、一つ目の値だけ(*values*[0])取り出している。

8. MLF における利用例

8.1. MLF モジュール

万葉ライブラリには、J-PARC の物質・生命科学実験施設（MLF）で利用できる汎用のツールや関数、物理定数などを収めたモジュールが同梱されている。これを MLF モジュールと呼ぶ。

8.1.1. 役割と目的

万葉ライブラリ自身は MLF での基盤解析環境として整備が進められているが、万葉ライブラリのコア部分は非常に汎用的なデータコンテナやツールのみで構成されている。そこで万葉ライブラリのコア部分を利用し、MLF でビームラインに寄らず使用できるツールや関数、物理定数などを含めたモジュールが必要である。それが MLF モジュールである。

万葉ライブラリを利用した開発では、すでに派生パッケージ（後述）がいくつか存在し、MLF 内のビームラインごとにコードが作成されている。それらのコードのうち汎用で使えるものをこの MLF モジュールに移植することで、広く他のビームラインにて使用できるようにすることが MLF モジュール作成の目的である。

8.1.2. 何ができるか

MLF モジュールには、下記のようなツールや関数が収められている。

- MLF 標準イベントデータ処理に使用される基礎関数やツール
- XML 用ツール
- MLF の線源情報の取得ツール（J-PARC サイト内でのみ有効）
- 幾つかの物理定数の定義
- 中性子散乱に関する補正を計算するための基礎的関数

8.1.3. 使用方法

万葉ライブラリがインストールされていれば

```
1    import Manyo.MLF
```

で MLF モジュールを扱えるようになる。 例えば、中性子の質量を得る場合は以下のような定数を呼び出す。

```
1    >>> import Manyo.MLF as mm
2
3    >>> print "Neutron Mass=%g [%s]"%( mm.MLF_MASS_NEUTRON, mm.MLF_MASS_NE
     UTRON_UNIT)
4    Neutron Mass=1.67493e-27 [kg]
```

MLF モジュールに含まれる定数やツールのいくつかはサンプルコードの章で具体的な使い方を示す。また Appendix B「万葉ライブラリ関数リファレンス」も参照のこと。

8.2. データコンテナとデータ処理の利用

ElementContainer を中心とした万葉ライブラリであるが、MLF における具体的な利用のコンセプトや例をいくつか挙げておく。

8.2.1. ElemetContainer

MLF において万葉ライブラリの ElementContainer は、装置のデータ収集システムが生み出すデータを収めるのに広く使用されている。ここに収めるべきデータは装置に設置された検出器と密接に結びつけられている。装置の検出器は、通常一つの検出器内に複数の検出部もしくは広い検出域があり、それらを適当な単位で区切ることで、中性子を検知した場所を識別する。得られたデータはこの検出単位ごとにヒストグラムデータとして扱われる必要がある。

そこで個々の ElementContainer は、「中性子検出器のうちの一つの検出単位（Pixel）に来た中性子の情報」を収めるのに使用される。

もう少し詳しく見ると、データ処理に必要な中性子の情報は

1. 検出中性子の状態と頻度
   - いくらのエネルギーもしくは波長の中性子がどれくらい検知されたか
   - 実際には中性子の検知時間と頻度（検知した中性子の個数）から計算
2. その他の情報
   - 検出単位の位置情報（検知時間から中性子のエネルギーや波長の情報を計算するのに使う）
   - 測定の情報（Run Number、試料情報、試料環境など）

である。これらを ElementContainer に収めるときには

1. 検出中性子の状態と頻度：ヒストグラム部
2. その他の情報：ヘッダ部

に分けて、それぞれ収められる。

ヒストグラム部では単位変換などで、様々な横軸のヒストグラムデータを持たせている。例えば弾性散乱では、横軸が時間、波長（λ）、エネルギー、運動量遷移（Q）、d 値といった単位のヒストグラムを持ち、非弾性散乱実験では時間やエネルギー遷移（hw）などを横軸の単位とするヒストグラムを持つ。

一方ヘッダ部では、その他の情報（メタ情報）を納めており、例えば測定番号（Run Number）、検出器および検出単位の ID、検出単位の位置情報（X,Y,Z）、散乱角度（θ,φ）、立体角の大きさ、マスク情報（このデータを使用する・しない）など、検出単位ごとの補正や解析に必要となる情報を納めている。その例を Table 25 にまとめた。

Table 25 「空蝉」パッケージにおける ElementContainer ヘッダ情報（例）

KEY	型	内容
PIXELID	Int4	検出単位（Pixel）の ID
PSDID	Int4	検出器(Detector)の ID
MASKED	Int4	マスク情報（1 なら Masked, 0 はなし）
PixelPosition	vector<double>	検出器の位置情報（試料位置を原点とする）単位[mm]
PixelSolidAngle	vector<double>	検出単位の立体角　単位[str]
TotalCounts	double	ヒストグラムの強度の和
PixelPolarAngle	vector<double>	検出単位の Polar 角度（2θ）
PixelAzimAngle	vector<double>	検出単位の Azimath 角度

ここで Int4 は signed long を表す。

8.2.2. ElementContainerArray

ElementContainer をまとめて扱うためのコンテナである。複数の ElementContainer を一度に扱うことができるので、主に検出単位(Pixel)を多く含んだ検出器 1 つ分の情報を収めるのに使用している。

複数の ElementContainre を保持し、ヘッダ部には主に、検出器の ID、スク情報、検出器の種類や位置情報（実位置情報やバンク情報）、検出器固有情報（位置敏感型検出器の Pulse Height 情報など）を管理している。その情報は Table 26 にまとめている。

Table 26「空蝉」パッケージにおける ElementContainerArray ヘッダ情報（例）

KEY	型	内容
DETID	Int4	検出器(Detector)の ID
MASKED	Int4	マスク情報（1 なら Masked, 0 はなし）
TYPE	string	検出器の種別（"PSD","N2MON"など）

8.2.3. ElementContainerMatrix

ElementContainerArray をまとめて扱うためのコンテナである。複数の ElementContainerArray を一度に扱うことできるので、検出器の集合体（バンク）や、装置全体のデータおよび情報などを収めるのに使用している。

複数の ElementContainerArray を保持し、ヘッダ部には主に、測定装置の識別文字、測定番号（Run Number）、試料情報（試料名や結晶、パウダーの区別など）、測定時間、ビームモニター強度などが含まれている。Table 27 を参照のこと。

Table 27「空蝉」パッケージにおける ElementContainerMatrix ヘッダ情報（例）

KEY	型	内容
RUNNUMBER	string	Run Number
MONITORCOUNT	Int4	ビームモニターのカウント数
INSTRUMENT	string	装置のコード（"SIK", "SAS"など）
SAMPLETYPE	string	試料の状態（"Powder", "SingleCrystal"など）
MEASPERIOD	vector<double>	測定の時刻情報（YYYY,MM,DD,hh,mm,ss,sub-sec）
DATAPROCESSED	vector<string>	データ処理のステップ（任意）
L1	double	L1 [mm]
Ei	double	非弾性散乱データ用入射中性子エネルギー

8.2.4. データ処理について

前の章で触れたように、万葉ライブラリとその派生パッケージ（「空蝉」「絵巻」など、詳細は後述）は多くのツールやデータ処理関数を持っている。特にデータ処理については、その処理機能ごとに関数が準備されており、処理のステップごとに処理関数の入力 -> 出力 ->次の処理関数の入力 -> 出力 -> ...を繰り返す形である。

検出単位ごとにヒストグラム化されたデータは、同じく万葉ライブラリを利用して作成された関数で処理される。基本的に万葉ライブラリの関数は、「ElementContainer, -Array, -Matrix を入力」し、幾つかの「パラメータやヘッダ情報を利用した処理」を行い、再び「ElementContainer, -Array, -Matrix として出力」するという形である。

しかし、MLF では装置や測定手法、ユーザーのサイエンスに応じてデータ処理が千差万別となる。そのため独自の関数を実装する必要に迫られることも多いであろう。これらの準備された関数だけで対応できない場合、まずは次項に示す万葉ライブラリの派生パッケージである「空蝉」や「絵巻」の関数を用いて Python でコードを書いてみることをお勧めする。

8.3. 派生パッケージ「空蝉」

万葉ライブラリを利用し、J-PARC MLF で使用されているデータ収集システムの DAQ ミドルウェアの MLF コンポーネントが出力するイベントデータに対し、ヒストグラム化や補正といったデータリダクション、および可視化を行うためのコード群である。特に柔軟なヒストグラム化や TrigNET を使用した高度な解析を目指す。

8.3.1. 何ができるか

- MLF のイベントデータのヒストグラム化
  - DAQ ミドルウェアが出力するイベントデータをヒストグラム化
  - 変換単位（TOF, λ, Q(momentum transfer), Energy, EnergyTransfer, d 値）
  - PSD のパルス波高の取り出し

  - フィルタリングの使用：時間分解、TrigNET を使用したイベント分別
  - 解析可能な検出器
    - 位置敏感型検出器(PSD) + NEUNET
    - 窒素ビームモニター検出器 + GATENET
    - ガンマ線検出器 + APV8008
    - 二次元検出器各種（WLSF, MWPC, RPMT）
- 簡単な補正
  - 線源強度による規格化
  - 立体角補正
  - 非弾性用補正（ki/kf など）
- 可視化
  - 1D プロッタ、2D プロッタ、3D プロッタ（作成中）
  - 単結晶非弾性散乱測定の可視化
  - 多次元測定の可視化

8.3.2. 空蝉の MLF ビームラインへの対応

空蝉は MLF のデータ処理の基盤環境、特に「ヒストグラム化部分の共通化」を目指している。 万葉ライブラリ＋空蝉に対し、さらに各装置用の拡張パッケージを組み合わせることで装置固有の機能の実現が図られている（Table 28）。

Table 28 空蝉の MLF ビームラインへの導入状況

ビームライン	導入状況
BL01 (4SEASONS)	空蝉で全対応
BL02 (DNA)	空蝉＋DNA 用拡張で対応
BL11 (Planet)	空蝉で全対応
BL14 (AMATERAS)	空蝉で全対応
BL15 (SAS)	空蝉＋SAS 用拡張＋絵巻で対応
BL18 (Senju)	空蝉＋StarGazer で対応
BL19 (Takumi)	空蝉＋絵巻で対応
BL21 (NOVA)	空蝉＋NOVA 用拡張で対応

参考文献

1) J. Suzuki, T. Nakatani, T. Ohhara, Y. Inamura, M. Yonemura, T. Morishima, T. Aoyagi, A. Manabe, T. Otomo : "Object-oriented data analysis framework for neutron scattering experiments", Nucl Instrum Methods Phys Res A, 600 (2009), pp.123-125.
2) Y. Inamura, T. Nakatani, J. Suzuki, and T. Otomo : "Development status of software 'Utsusemi' for Chopper Spectrometers at MLF, J-PARC", J. Phys. Soc. Jpn., 82 (2013), SA031-1 – SA031-9.
3) Y. Inamura, J.-Y. So, K. Nakajima, J. Suzuki, T. Nakatani, et al. : "Technical Report on the Korea-Japan Software Collaboration", JAEA-Technology 2010-047, J-PARC 10-03, 74p.
4) T. Ito, T. Nakatani, S. Harjo, H. Arima, J. Abe, K. Aizawa and A. Moriai : "Application software development for the engineering materials diffractometer, TAKUMI", Materials Science Forum, vol. 652 (2010) pp.238-242.
5) T. Ito, S. Harjo, Y. Inamura, T. Nakatani, T. Kawasaki, J. Abe, and K. Aizawa : "Utilization of an Event-Recording System for Neutron Diffraction Experiments", Materials Science Forum, vol. 783-786 (2014) pp.2071-2074.
6) NeXus : A common data format for neutron, x-ray and muon science, available from http://www.nexusformat.org/ （参照：2016 年 1 月 15 日）.
7) HDF5, available from https://www.hdfgroup.org/HDF5/ （参照：2016 年 1 月 15 日）.

Appendix

A. Python イントロダクション

A.1. はじめに

本書でで初めて Python に触れる読者のために、Python の基本的な機能を簡単に説明する。

なお、入力例の行頭の「$」「>>>」はそれぞれ、ターミナルと Python のプロンプトを示しており、入力する必要はない。また「#」以降はコメントを表している。

A.2. help, ipython

A.2.1. help

Python の実行中に次のように使用することで説明文を表示する。

```
>>> help('Manyo.ElementContainer')
```

もし、使い方を知らない変数（仮に *something* としよう）が出てきた場合も、次のようにして説明文を表示できる。

```
>>> help(something)
```

A.2.2. ipython

ipython は Python の高機能なインターフェースである。その機能の一つにタブキーによる補完機能がある。メソッド名やファイル名、変数名などを途中まで入力しタブキーを押せば、補完、あるいは、候補の列挙を行う。例えば ElementContainer のメソッド名を忘れてしまっても、

```
>>> ec = Manyo.ElementContainer()
>>> ec.
```

まで、入力しタブキーを押せば、ElementContainer のメソッド全てが表示される。

```
>>> ec = Manyo.ElementContainer()
>>> ec.
ec.Add                      ec.InputUnitHeader          ec.PutE
         ec.PutTableSize             ec.ReduceColumnList
ec.AddBlankVector           ec.Integrate                ec.PutEKey
         ec.PutUnit                  ec.Remove
ec.AddToHeader              ec.LoadTextFile             ec.PutEList
         ec.PutUnitHeader            ec.Replace
ec.AppendValue              ec.MaskedDiv                ec.PutHeader
         ec.PutUnitHeaderPointer     ec.Reverse
ec.Ave                      ec.MaskedMul                ec.PutHeaderPointer
         ec.PutX                     ec.SaveTextFile
ec.Averaging                ec.MaskedPlus               ec.PutHistKeyList
         ec.PutXKey                  ec.SaveToBinFile
```

```
（以下略）
```

もう一つの便利な機能はログ機能である。ipython を利用中に

```
>>> %logstart
```

と入力すると、それ以降の入力がファイルに保存される。そのログファイルは、ほとんどそのままでスクリプトとして再利用できる。ipython を終了するには、「**Ctrl+d**」を入力する。

A.3. import

Python は様々なライブラリ（パッケージやモジュールと呼ばれる）の機能を利用できる。ライブラリは次のように **import** することで利用出来るようになる。

```
>>> import math
```

これにより、数学関数モジュール **math** が読み込まれる。

モジュールに含まれる関数を用いるにはモジュール名に「.」（ドット）と関数名を続ける。

```
>>> math.sqrt(2)
```

また、import の際に、**as** を使ってモジュール名に別名を付けることができる。

```
>>> import os.path as pth
```

これでパス名操作用モジュール os.path を読み込み **pth** という別名を付ける。利用方法は次のようになる。

```
>>> pth.abspath('.') # カレントディレクトリの絶対パス
'/home/hogehoge/fugafuga/~'
```

以下によく使うモジュールの表を挙げる。

モジュール名	説明
math	数学関数、定数
os.path	パス名操作
sys	Python に関する変数、関数
time	時間に関する関数

A.4. print と文字列

A.4.1. 文字列その 1

Python での文字列は「'」（シングルクォート）、「"」（ダブルクオート）、「'''」もしく「"""」トリプルクォートで囲む。

```
>>> s = 'Tokai'
```

文字列中にシングルクォートやダブルクオートを含めたい場合には、それぞれ他方のクォートで囲む。

A.4.2. print

Python で文字列を表示するために用いるのが **print** である。例えば、次のように用いる。

```
>>> print 'Hello, world'
Hello, world
```

「,」（カンマ）で区切ると、空白区切りで連続して表示される。

```
>>> print 'Tokai', 'Naka', 'Ibaraki'
Tokai Naka Ibaraki
```

A.4.3. 文字列その２

文字列中に変数の値を代入するには文字列の後に「%」を付け、その後に変数を羅列する。

```
>>> t = 12.345
>>> unit = 'K'
>>> print 'Temperature: %.2f %s'%( t, unit)
Temperature: 12.35 K
```

A.5. データ型

Python の主なデータ型を次に示す。

文字列	str	
整数	int	signed、 unsinged の区別は無い。
浮動小数	float	
リスト	list	配列
タプル	tuple	変更不可能な配列
辞書	dict	連想配列

Python では変数を新たに用いる際に、C 言語のようにデータ型を明示的に示す必要がない。

変数の型を知りたいときには **type** 関数を用いる。

```
>>> x = 10
>>> type(x)
int
```

また、型の変換は次のようにして行える。

```
>>> y = float(x)
>>> type(y)
float
```

また、ある変数が、求める型であるかの真偽を確認するには **isinstance** 関数を用いる。

```
>>> isinstance(ec, Manyo.ElementContainer)
True
```

A.6. list

リストは次のように「[」と「]」で囲うことで作成する。

```
>>> l = ['SIK', 'DNA', 'BIX', 'NRI']
```

リストから値を取り出す方法は C 言語の配列と同じである。

```
>>> l[1]
'DNA'
```

また、次のような方法で、リストの一部を得られる。

```
>>> l[1:3]
['DNA', 'BIX']
>>> l[2:]
['BIX', 'NRI']
```

要素を指すためのインデックスに負の数字を使うこともできる。

```
>>> l[-1]
'NRI'
>>> l[-2:]
['BIX', 'NRI']
```

リストに関係する関数などを紹介しておく。

A.6.1. len(*list*)

リストの長さを返す。

```
>>> len(l)
4
```

A.6.2. append(*item*)

リストの最後に要素を追加する。

```
>>> l.append('NOP')
>>> print l
['SIK', 'DNA', 'BIX', 'NRI', 'NOP']
```

A.6.3. remove(*item*)

リストの中身から *item* を削除する。*item* がリスト中に無い場合はエラーになる。

```
>>> l.remove('DNA')
>>> print l
['SIK', 'BIX', 'NRI', 'NOP']
```

A.6.4. sort()

昇順でリストの中身を並び替える。

```
>>> l.sort()
>>> print l
['BIX', 'NOP', 'NRI', 'SIK']
```

A.6.5. reverse()

リストの中身を逆順に並び替える。

```
>>> l.reverse()
>>> print l
['SIK', 'NRI', 'NOP', 'BIX']
```

A.6.6. range(*start*, *end*, *step*)

start から *end*（の手前）まで *step* おきの、整数リストを返す。*start*、*step* は省略できる。

```
>>> range(5)
[0, 1, 2, 3, 4]

>>> range(-2,2)
[-2, -1, 0, 1]

>>> range(0, 100, 20)
[0, 20, 40, 60, 80]
```

A.7. if, for, while, try

プログラミングに欠かせない制御文を紹介する。

A.7.1. 制御の区切り

制御文の文末は「:」（コロン）で示す。また、制御文の影響下に置かれるまとまりをインデントの数で区別する。インデントは空白でもタブでもよいが、一区切りのインデント数は一つのソースコードの中で一定でなければならない。

A.7.2. if...elif...else...

```
import random # random は様々な乱数を扱うモジュール

a = random.randint(-10, 10) # randint は二つの引数の間の整数をランダムに返す
if a < 0:
        print 'Negative'
elif a == 0:
        print 'Zero'
else:
        print 'Positive'
```

elif は何回でも使える。**else** は一回のみ使える。どちらについても、使わなくても良い。

A.7.3. for...else...

for*item***in***list*:で始める。*list* の内容を一つずつ取り出して、変数 *item* に代入し、それ以降の処理を行う。**break** されないでループを終えた場合、**else** の処理を行う。**else** は使わなくても良い。

```
l = ['SIK', 'DNA', 'BIX', 'NRI']
for bl_name in l:
        print bl_name
else:
        print 'Iteration ends.'
```

A.7.4. while...else...

条件が成り立っている間、ループを続ける。成り立たたずにループを抜けた後に **else** の処理を行う。**break** されてループを抜けた場合は **else** の処理は行わない。**else** は使わなくても良い。

```
bl_code = raw_input('Choice your BL. %s >'%(l))
while bl_code not in l:
        print "Not found."
        bl_code = raw_input('Choice your BL. %s >'%(l))
else:
        print 'Good!'
```

A.7.5. try...except...

エラーが起こった場合に別処理を行う。

```
a = random.randint(0,3)
try:
        b = 10/a
except:
        print "Error"
```

B. 万葉ライブラリ関数リファレンス

この節では、本書内で使用された万葉ライブラリの関数のリファレンスを記述する。ただし万葉ライブラリは C++で作成されていることもあり、本節の記述は C++での表現を使用している。本節の表現については、第２章「データコンテナ」の「万葉ライブラリの変数・配列について」を参照のこと。

1. ElementContainer
2. ElementContainerArray
3. ElementContainerMatrix
4. HeaderBase
5. SearchInHeader
6. WriteSerializationFile
7. ReadSerializationFile
8. NeXusFileIO
9. CppToPython
10. MLF constants

B.1. ElementContainer

***class* ElementContainer**

万葉ライブラリのデータコンテナとして、最も基本的なコンテナである。 一次元のヒストグラムを保存することができる。 ElementContainer を束ねることで二次元検出器 (ElementContainerArray), さらに、ElementContainerArray を束ねると ElementContainerMatrix とすることができる。

B.1.1. メソッド　主要一覧

ElementContainer()
コンストラクタ。

ElementContainer(HeaderBase &*Header*)
コンストラクタ。既存の HeaderBase を使用する。

~ElementContainer()
デストラクタ。

ElementContainer(*const* ElementContainer &*ob*)
コピーコンストラクタ。引数のコンテナを複製する。

ElementContainer operator=(ElementContainer *ob*)
"="演算子

ElementContainer operator+(ElementContainer &*r*)
"+"演算子。これら四則演算子は実行前に SetKeys(string,string,string)を 実行しておく必要がある。 演算されるヒストグラムの bin の数が異なる場合は、 演算されるヒストグラムが重なっている部分だけ抽出してから bin の数が少ない方の bin にリフォーマットされる。 戻り値の ElementContainer には、SetKeys()で示されている vector の計算値のみが登録されている。 HeaderBase の内容はコピーされず、 戻り値の ElementContainer の HeaderBase は初期化されたものである。

ElementContainer operator-(ElementContainer &*r*)
"-"演算子。以下、"+"演算子と同じ。

ElementContainer operator*(ElementContainer &*r*)
"*"演算子。以下、"+"演算子と同じ。

ElementContainer operator/(ElementContainer &*r*)
"/"演算子。 割り算をする場合、分母がゼロである場合は、以下に従って処理される。 C++でエラーメッセージを表示した後、 ヒストグラムの値を 0.0, エラーを 1.0 の値を入力する設定になっている。 以下、"+"演算子と同じ。

ElementContainer Mul(Double *d*)
SetKeys(string,string,string)で指定したヒストグラムが d 倍された ElementContainer のインスタンスを新たに生成して返す。 オリジナルのヒストグラムは変更されない。 HeaderBase の内容はそのまま引き継がれる。

void pMul(ElementContainer **result*, Double *d*)
ElementContainer::Mul() が内部的に利用。 生成するオブジェクトを引数で渡されたポインタへ書き込むので return に伴なうコピーの分オーバーヘッドが少ない。

ElementContainer ReBin(ElementContainer *E*, string *Key*)
ElementContainer のインスタンスを新たに生成し、 引数の E と Key で指定された vector<Double>を新しいヒストグラムのビンとして、 自分の SetKeys(string,string,string) で指定したヒストグラムの bin の幅を調整した新規のヒストグラムを作成し そのコンテナを返す。 ヒストグラムは、そのヒストグラムで表現される面積が変更されないように調整される。

ElementContainer ReBin(vector<Double> *v*)
ReBin(ElementContainer E, string Key)と同様の働き。 bin を直接引数に指定する。

void SetKeys(string *X*, string *Y*, string *E*)
ElementContainer が一次元のヒストグラムを形成するための キーを設定する。 設定するためには あらかじめこのメソッドの引数の名前で vector<Double>が すべて入力されている必要がある。 必要に応じて何時でもこのメソッドで キーワードを変更することが出来る。

X はヒストグラムの bin をなす vector<Double>の名前、 Y は、ヒストグラムの高さを示す vector<Double>の名前、 E は、二つ目の値のエラーを示す vector<Double>名前である。 ヒストグラムを定義するためには、 Y と E のサイズが同じで、X のサイズは、 Y と E のサイズよりひとつ大きい必要がある。

string PutXKey()
SetKeys(string,string,string)で設定した、 X の名前を返す。

string PutYKey()
SetKeys(string,string,string)で設定した、 Y の名前を返す。

string PutEKey()
SetKeys(string,string,string)で設定した、 Z の名前を返す。

vector<string> PutKeys()
データ領域に登録された vector<Double>の名前の一覧を返す。

vector<string> PutKeysList()
データ領域に登録された vector<Double>の名前の一覧を Python の list にして返す。

vector<string> PutHistKeyList()
SetKeys(string,string,string)で設定した、vector<Double>の名前を Python の list にして返す。list には X、Y、Z の順に収められている。

vector<Double> PutX()
SetKeys(string,string,string)で設定した X のキーに相当する vector<Double>を返す。

vector<Double> PutY()
SetKeys(string,string,string)で設定した Y のキーに相当する vector<Double>を返す。

vector<Double> PutE()
SetKeys(string,string,string)で設定した E のキーに相当する vector<Double>を返す。

UInt4 PutTableSize()
現在登録済みの、vector<Double>の数を返す。

UInt4 PutSize(UInt4 *IndexNumber*)
IndexNumber の順番に登録された vector<Double> のサイズを返す。

UInt4 `PutSize`(string *Key*)
Key の名前で登録された vector<Double> のサイズを返す。

UInt4 `PutIndexNumber`(string *Key*)
Key で登録されている vector の Index-Number を返す

string `PutName`(UInt4 *IndexNumber*)
IndexNumber の順番に登録された vector<Double>の 名前を返す。

HeaderBase `PutHeader`()
その ElementContainer の持つ HeaderBase オブジェクトを返す。

void `InputHeader`(HeaderBase &*pHeader*)
コンストラクタで HeaderBase を登録しなった場合に使用する。 HeaderBase をヘッダオブジェクトとして入力する。 HeaderBase が登録済みの場合は、 既存のオブジェクトは消去される。

void `DumpKey`()
登録済みの vector<Double>の名前一覧を標準出力に 表示する。

void `Add`(string *Key*, vector<Double> *value*)
Key の名前で "vector<Double> value"を登録する。 Key がすでに登録済みの場合はエラーメッセージを表示し 登録が拒否される。

void `Add`(string *Key*, PyObject **value*)
Key の名前で "value"を登録する。value は、要素が double の Python リストである。 (要素が整数である場合は double に変換される) Key がすでに登録済みの場合はエラーメッセージを表示し 登録が拒否される。

PyObject *`PutXList`()
SetKeys()で X として指定してある vector を Python の list にして返す。

PyObject *`PutYList`()
SetKeys()で Y として指定してある vector を Python の list にして返す。

PyObject *`PutEList`()
SetKeys()で Err として指定してある vector を Python の list にして返す。

void `Add`(string *Key*, vector<Float> *value*)
value は、vector<Double>に変換された後、 Key の名前で登録される。 Key がすでに登録済みの場合はエラーメッセージを表示し 登録が拒否される。

void `Add`(string *Key*, vector<UInt4> *value*)
value は、vector<Double>に変換された後、 Key の名前で登録される。 Key がすでに登録済みの場合はエラーメッセージを表示し 登録が拒否される。

void `Add`(string *Key*, vector<UInt2> *value*)
value は、vector<Double>に変換された後、 Key の名前で登録される。 Key がすでに登録済みの場合はエラーメッセージを表示し 登録が拒否される。

void Add(string *Key*, Double *&*Array*, UInt4 *ArraySize*)
配列 Array(サイズは ArraySizeArray)は、 vector<Double>に変換された後、 Key の名前で登録される。 Key がすでに登録済みの場合は エラーメッセージを表示し 登録が拒否される。

void Add(string *Key*, Float *&*Array*, UInt4 *ArraySize*)
配列 Array(サイズは ArraySizeArray)は、 vector<Double>に変換された後、 Key の名前で登録される。 Key がすでに登録済みの場合は エラーメッセージを表示し 登録が拒否される。

void Add(string *Key*, UInt4 *&*Array*, UInt4 *ArraySize*)
配列 Array(サイズは ArraySizeArray)は、 vector<Double>に変換された後、 Key の名前で登録される。 Key がすでに登録済みの場合は エラーメッセージを表示し 登録が拒否される。

void Add(string *Key*, UInt2 *&*Array*, UInt4 *ArraySize*)
配列 Array(サイズは ArraySizeArray)は、 vector<Double>に変換された後、 Key の名前で登録される。 Key がすでに登録済みの場合は エラーメッセージを表示し 登録が拒否される。

vector<Double> Find(string *Key*)
Key で登録されている vector<Double>を返す。

vector<Double> Find(UInt4 *IndexNumber*)
IndexNumber 番目に登録されている vector<Double>を返す。

Double Find(string *Key*, UInt4 *Index*)
Key で登録されている vector<Double>の Index 番目の値を返す。 (Find(Key))[Index]と同じ。

vector<Double> Put(UInt4 *IndexNumber*)
Find(UInt4)のエイリアス

vector<Double> Put(string *Key*)
Find(string)のエイリアス

Double Put(string *Key*, UInt4 *Index*)
Find(string,UInt4)のエイリアス

UInt4 CheckKey(string *Key*)
Key の名前で登録されている vector<Double>の個数を返す。

void Copy(string *Old*, string *New*)
Old の名前で登録されている vector<Double>をコピーして 新規に New の名前で登録する。 Old の方は変更されない。

void Remove(string *Key*)
Key の名前で登録されている vector<Double>を削除する。 IndexNumber は、Key より後ろの番号は、消去した分を詰める。

void Remove(UInt4 *IndexNumber*)
IndexNumber で登録されている vector<Double>を削除する。 IndexNumber は、消去した分を詰める。

void Replace(string *Key*, vector<Double> *value*)
Key で登録されている vector<Double>を、二番目の引数の vector<Double>に 入れ替える。

void AppendValue(string *Key*, Double *value*)
Key で登録されている vector<Double>に、value を追加する。

void AppendValue(string *Key*, vector<Double> *value*)
Key で登録されている vector<Double>の末尾に value を連結する。

void SetValue(string *Key*, UInt4 *Number*, Double *value*)
Key で登録されている vector<Double>の Numer 番目の値を value にする。

void Dump()
登録されている内容を標準出力に全て表示する。 ただし、vector のサイズが大きすぎると表示の内容が大きくなりすぎる。 その場合は、Dump(UInt4)を用いる。

void Dump(UInt4 *size*)
登録されている内容を標準出力に表示する。 size 以上のサイズの vector は、size 以上の部分は省略表示になる。 size より小さい vector は全て表示される。

void SaveTextFile(string *FileName*)
ヒストグラムをカンマなどで区切った 3 列のテキストファイルとして保存する。
FileName がファイル名となる。

void SaveTextFile(string *FileName*, Int4 *prec*)
ヒストグラムをカンマなどで区切った 3 列のテキストファイルとして保存する。
FileName がファイル名、prec が桁数となる。

void SaveTextFile(string *FileName*, Char *deli*)
ヒストグラムをカンマなどで区切った 3 列のテキストファイルとして保存する。
FileName がファイル名、deli が区切り文字となる。

void SaveTextFile(string *FileName*, Int4 *prec*, Char *deli*)
ヒストグラムをカンマなどで区切った 3 列のテキストファイルとして保存する。
FileName がファイル名、prec が桁数、deli が区切り文字となる。

void SaveHistTextFile(string *FileName*)
ヒストグラムをカンマなどで区切った 3 列のテキストファイルとして保存する。bin の両端の値が保存される。FileName がファイル名となる。

void SaveHistTextFile(string *FileName*, Int4 *prec*)
ヒストグラムをカンマなどで区切った 3 列のテキストファイルとして保存する。bin の両端の値が保存される。FileName がファイル名、prec が桁数となる。

void SaveHistTextFile(string *FileName*, Char *deli*)
ヒストグラムをカンマなどで区切った 3 列のテキストファイルとして保存する。bin の両端の値が保存される。FileName がファイル名、deli が区切り文字となる。

void SaveHistTextFile(string *FileName*, Int4 *prec*, Char *deli*)
ヒストグラムをカンマなどで区切った 3 列のテキストファイルとして保存する。bin の両端の値が保存される。FileName がファイル名、prec が桁数、deli が区切り文字となる。

B.2. ElementContainerArray
class ElementContainerArray

以下のメソッドの紹介は、ElementContainerArray に実装されているもののみで あるので、NeutronVector< ElementContainer, HeaderBase >の public カテゴリの メソッドも使用できる。

B.2.1. メソッド　主要一覧

ElementContainerArray(UInt4 *pdepth* = 1)
コンストラクタ。引数は通常は省略される。

ElementContainerArray(HeaderBase *pheader*, UInt4 *pdepth* = 1)
コンストラクタ。既存の HeaderBase を登録する場合に使用する。

ElementContainer *operator()(UInt4 *index*)
ElementContainerArray の index 番目に保存されている ElementContainer のポインタを返す。 このメソッドは、ElementContainer の内部のメソッドにアクセス する目的以外に使用してはならない。 また取り出したポインタの値を ElementContainerArray の外部に保存してはならない。ポインタでなくコピーを得るには、親クラスのメソッドを使用する。

vector<Double> *operator()(UInt4 *index*, string *key*)
ElementContainerArray の index 番目に保存されている ElementContainer から、key の名前で保存されている vector のポインタを返す。 このメソッドは、vector<Double>の内部のメソッドにアクセスする目的以外に使用してはならない。 また取り出したポインタの値を ElementContainerArray の外部に保存してはならない。ポインタでなくコピーを得るには、親クラスのメソッドを使用する。

vector<Double> *operator()(UInt4 *index*, UInt4 *C_index*)
ElementContainerArray の index 番目に保存されている ElementContainer から、C_index 番目に保存されている vector のポインタを返す。 このメソッドは、vector<Double>の内部のメソッドにアクセス する目的以外に使用してはならない。 また取り出したポインタの値を ElementContainerArray の外部に保存しては ならない。ポインタでなくコピーを得るには、親クラスのメソッドを使用する。

void AddToHeader(string *Key*, Int4 *value*)
親クラスである NeutronVector< ElementContainer, HeaderBase >の HeaderBase へ直接値を入力する。

void AddToHeader(string *Key*, Double *value*)
親クラスである NeutronVector< ElementContainer, HeaderBase >の HeaderBase へ直接値を入力する。

void AddToHeader(string *Key*, string *value*)
親クラスである NeutronVector< ElementContainer, HeaderBase >の HeaderBase へ直接値を入力する。

void AddToHeader(string *Key*, vector<Int4> *value*)
親クラスである NeutronVector< ElementContainer, HeaderBase >の HeaderBase へ直接値を入力する。

void AddToHeader(string *Key*, vector<Double> *value*)
親クラスである NeutronVector< ElementContainer, HeaderBase >の HeaderBase へ直接値を入力する。

void AddToHeader(string *Key*, vector<string> *value*)
親クラスである NeutronVector< ElementContainer, HeaderBase >の HeaderBase へ直接値を入力する。

B.3. ElementContainerMatrix
***class* ElementContainerMatrix**

以下のメソッドの紹介は、ElementContainerMatrix に実装されているもののみで あるので、NeutronVector< ElementContainerArray, HeaderBase >の public カテゴリの メソッドも使用できる。

B.3.1. メソッド　主要一覧
ElementContainerMatrix(UInt4 *pdepth* = 1)
コンストラクタ。このメソッドの引数は、親クラスのコンストラクタに渡される。

ElementContainerMatrix(HeaderBase *pheader*, UInt4 *pdepth* = 1)
コンストラクタ。このメソッドの引数は、親クラスのコンストラクタに渡される。

ElementContainerArray *operator()(UInt4 *index*)
index 番目の ElementContainerArray のポインタを返す。 このメソッドは ElementContainerArray の内部のメソッドにアクセスする目的 以外に使用してはならない。取り出したポインタの値を ElementContainerMatrix の外部に保存してはならない。

ElementContainer *operator()(UInt4 *Mindex*, UInt4 *Aindex*)
Mindex 番目の ElementContainerArray の内部の Aindex 番目の ElementContainer のポインタを返す。 このメソッドは ElementContainer の内部のメソッドにアクセスする目的 以外に使用してはならない。取り出したポインタの値を コンテナの外部に保存してはならない。

vector<Double> *operator()(UInt4 *Mindex*, UInt4 *Aindex*, UInt4 *Cindex*)
Mindex 番目の ElementContainerArray の内部の Aindex 番目の ElementContainer の Cindex 番目の vector<Double>のポインタを返す。 このメソッド vector の内部のメソッドにアクセスする目的 以外に使用してはならない。取り出したポインタの値を コンテナの外部に保存してはならない。

vector<Double> *operator()(UInt4 *Mindex*, UInt4 *Aindex*, string *Key*)
Mindex 番目の ElementContainerArray の内部の Aindex 番目の ElementContainer の 名前:Key の vector<Double>のポインタを返す。 このメソッド vector の内部のメソッドにアクセスする目的 以外に使用してはならない。取り出したポインタの値を コンテナの外部に保存してはならない。

void AddToHeader(string *Key*, Int4 *value*)
親クラスである NeutronVector< ElementContainerArray, HeaderBase >の HeaderBase へ直接値を入力する。

void AddToHeader(string *Key*, Double *value*)
親クラスである NeutronVector< ElementContainerArray, HeaderBase >の HeaderBase へ直接値を入力する。

void AddToHeader(string *Key*, string *value*)
親クラスである NeutronVector< ElementContainerArray, HeaderBase >の HeaderBase へ直接値を入力する。

void AddToHeader(string *Key*, vector<Int4> *value*)
親クラスである NeutronVector< ElementContainerArray, HeaderBase >の HeaderBase へ直接値を入力する。

void AddToHeader(string *Key*, vector<Double> *value*)
親クラスである NeutronVector< ElementContainerArray, HeaderBase >の HeaderBase へ直接値を入力する。

void AddToHeader(string *Key*, vector<string> *value*)
親クラスである NeutronVector< ElementContainerArray, HeaderBase >の HeaderBase へ直接値を入力する。

B.4. HeaderBase
class HeaderBase

HeaderBase は、種々の型のデータをキーワードで管理するクラスである。 主に、ElementContainer などに組み込まれ、 そこに保存されている数値データの集合体のかたまりを記述する データヘッダ情報を保持するヘッダオブジェクトに用いられる。

HeaderBase には、多くの種類の型を保存できることを目的に設計されているので 高速な検索や、巨大なデータの保存には向いていない。 入力できる型は、Int4, Double, String, vector<Int4>, vector<Double>, vector<string>である。 6 種のデータ型が入力できるが、全てのデータは string の key を付して管理する。 key は HeaderBase の中において別のものをつける必要がある。 これは入力時と出力時で型が異なっていてもクラス内部で出来うる限り変換を行って 出力に対応出来るようにするためである。

HeaderBase はクラスの内部に、データを保持するクラス、Map<T>を複数持つ。

B.4.1. メソッド　主要一覧

HeaderBase()
コンストラクタ。

~HeaderBase()
デストラクタ。全てのデータを消去しメモリを解放する。

void Add(string *key*, Int4 *value*)
void Add(string *key*, double *value*)
void Add(string *key*, string *value*)
void Add(string *key*, vector<Int4> **values*)
void Add(string *key*, vector<double> **values*)
void Add(string *key*, vector<string> **values*)
データをキーワード付きで入力する。 *key* はキー、 *value* は Int4, Double, string のいわゆる通常の数、 *values* は、vector を入力する。

void OverWrite(string *key*, Int4 *value*)
void OverWrite(string *key*, double *value*)
void OverWrite(string *key*, string *value*)
void OverWrite(string *key*, vector<Int4> **values*)
void OverWrite(string *key*, vector<double> **values*)
void OverWrite(string *key*, vector<string> **values*)
データを置き換える。 *key* はキー、 *value* は Int4, Double, string のいわゆる通常の数、 *values* は、vector を入力する。もし *key* が存在しなかった場合、ワーニングメッセージが表示されるが、 **Add** と同じように追加される。

Map<type> *PutTypeMap()
以下のいずれかのメソッドが呼び出される。

Map< Int4 > *PutInt4Map()
Map< Double > *PutDoubleMap()
Map< string > *PutStringMap()
Map< vector< Int4 >* > *PutInt4VectorMap()

Map< vector< Double >* > *PutDoubleVectorMap()
Map< vector< string >* > *PutStringVectorMap()

データを保存しているクラス Map<T>のポインタを返す。 より詳細なデータの取り扱いが必要なときに用いる。

void Erase(string *Key*)
Key で登録されているデータを消去する。

void Dump()
ユーザーに分かりやすいフォーマットでディスプレーに現在の登録状況を表示する。

PutType(string)
以下のいずれかのメソッドが呼び出される。

Int4 PutInt4(string Key)
Double PutDouble(string Key)
string PutString(string Key)
vector<Int4> *PutInt4Vector(string Key)
vector<Double> *PutDoubleVector(string Key)
vector<string> *PutStringVector(string Key)

Key で登録されているデータを各データ型に変換して返す。 例えば、Double で登録して、Int4 で返す、 string で"3"と登録してあれば、Double で 3.0 で返す、 Double を string で返すなどが可能である。

B.5. SearchInHeader

***class* SearchInHeader**

万葉ライブラリのデータコンテナのヘッダは、そのデータに対するメタデータを収めるために使用されている。ElementContainerArray や ElementContainerMatrix が持つメタデータに対して検索をかけるためのクラスである。

B.5.1. メソッド　主要一覧

SearchInHeader(bool *flag* = false)
SearchInHeader(ElementContainerMatrix **target*)
SearchInHeader(ElementContainerArray **target*)

コンストラクタ。flag はデバッグフラグである。通常は false でよい。target は対象となるデータコンテナであり、ElementContainerMatrix か ElementContainerArray である。ここで target を指定しない場合は、SetTarget メソッドで対象となるデータコンテナを与えること。

~SearchInHeader()

デストラクタ。

void SetTarget(ElementContainerMatrix **target*)
void SetTarget(ElementContainerArray **target*)

検索したいデータコンテナを与える。target は対象となるデータコンテナであり、ElementContainerMatrix か ElementContainerArray である。

bool Search(string *key*, Int4 *val*)
bool Search(string *key*, Int4 *val_lower*, Int4 *val_upper*)
bool Search(string *key*, Double *val_lower*, Double *val_upper*)

対象となるデータコンテナに含まれるすべての ElementContainer のヘッダ情報の整数値と実数値に対し検索を行う。key は検索したいヘッダ情報の名前(Key)である。 引数として val（整数）のみ与えられている場合、ヘッダ情報の中で val の型（整数）を持つ key の値と一致したものを検索結果に加える。val_lower, val_upper が与えられている場合（整数、実数）はそれぞれ値の下限、上限であり、ヘッダ情報の同じ型の key の値が、その範囲内にあれば検索結果に加える。 戻り値は対象となるデータコンテナが存在しない場合のみ false である。検索結果が 0 の場合（Key が見つからない場合も含む）でも true である。

bool Search(string *key*, string *val*, bool *isStrict* = false)

対象となるデータコンテナに含まれるすべての ElementContainer のヘッダ情報に対し、文字列情報への検索を行う。key は検索したい文字列情報の名前(Key)であり、val で検索する文字列を指定する。isStrict は、false ならばヘッダの文字列情報に val が含まれていれば一致したとみなし、true ならばとまったく同じ文字列の場合のみ一致したとみなす。戻り値は対象となるデータコンテナが存在しない場合のみ false である。検索結果が 0 の場合（Key が見つからない場合も含む）でも true である。

bool SearchArray(string *key*, Int4 *val*)
bool SearchArray(string *key*, Int4 *val_lower*, Int4 *val_upper*)
bool SearchArray(string *key*, Double *val_lower*, Double *val_upper*)

対象となるデータコンテナに含まれるすべての ElementContainerArray のヘッダ情報に対し、整数の情報への検索を行う。それぞれ Search(args)と同じ挙動を示す。

bool SearchArray(string *key*, string *val*, bool *isStrict* = false)
対象となるデータコンテナに含まれるすべての ElementContainerArray のヘッダ情報に対し、文字列の情報への検索を行う。Search(string key, string val, bool isStrict=false)と同じ挙動を示す。

ElementContainerArray PutResultAsArray()
Search による検索結果である ElementContainer をコピーして、ElementContainerArray に収めて戻す。

```
import Manyo
SIH = Manyo.SearchInHeader( dat0 )  # dat0 は ElementContainerMatrix
SIH.Search( "DETID", 100, 120 )     # ヘッダで"DETID"が 100 から 120 までの E
lementContainer を探す
dat2 = SIH.PutResultAsArray()       # 見つかった ElementContainer のコピー
を Array に収めて戻す
```

ElementContainerMatrix PutResultSearchArray()
SearchArray による検索結果である ElementContainerArray をコピーして、ElementContainerMatrix に収めて戻す。

vector<vector<UInt4>> PutResults()
Search および SearchArray の結果を取り出す。戻り値は検索結果のデータコンテナの個数の vector である。この vector は特定のデータコンテナを示すのに必要な index 情報を含んだ vector であり、そのフォーマットは以下のようになっている。

target	メソッド	戻り値	内容
ElementContainerArray	Search	大きさ 1 の vector	ElementContainerArray 内の index
ElementContainerMatrix	Search	大きさ 2 の vector	ElementContainerMatrix 内の index, ElementContainerArray 内の index
ElementContainerMatrix	SearchArray	大きさ 1 の vector	ElementContainerMatrix 内の index

サンプルコード

```
import Manyo
SIH = Manyo.SearchInHeader( dat0 )  # dat0 は ElementContainerMatrix
SIH.Search( "DETID", 100 )     # ヘッダで"DETID"が 100 から 120 までの
ElementContainer を探す
result_vec = SIH.PutResults()
if result_vec.size()!=0:
    index_ecm = result_vec[0][0]
    index_eca = result_vec[0][1]
    ec = dat0.Put( index_ecm ).Put( index_eca )
```

vector<UInt4> `PutResultIndex`(UInt4 *index* = 0)

Search および SearchArray の結果を、vector<UInt4>で取り出す。検索結果と index との関係は以下のとおり。

SetTarget(XXX)	検索に使用したメソッド	引数 (index)	戻り値の内容
ElementContainerArray	Search	0	ターゲット内の ElementContainer の index
ElementContainerMatrix	Search	0	ターゲット内の ElementContainerArray の index
ElementContainerMatrix	Search	1	上の ElementContainerArray 内の ElementContainer の index
ElementContainerMatrix	SearchArray	0	ElementContainerArray の index

B.6. WriteSerializationFile
class WriteSerializationFile

Boost C++のシリアル化の機能を使用してデータコンテナをファイルに保存する。 ファイルをオープン(コンストラクタ)してから複数回 Save()を呼び出せば、 1 つのファイルに複数のコンテナを書き込むことができる。 読み出すときには、書き込んだ順に読み出さなければならない。

B.6.1. メソッド　主要一覧

WriteSerializationFile(*const* char **filename*)
コンストラクタ。保存するファイル名を指定する。

void Save(*const* HeaderBase &*data*)
HeaderBase を保存する。

void Save(*const* ElementContainer &*data*)
ElementContainer を保存する。

void Save(*const* ElementContainerArray &*data*)
ElementContainerArray を保存する。

void Save(*const* ElementContainerMatrix &*data*)
ElementContainerMatrix を保存する。

B.7. ReadSerializationFile
class ReadSerializationFile

WriteSerializationFile で保存したファイルを読み出す。

2.7.1. メソッド　主要一覧

ReadSerializationFile(*const* char **filename*)
コンストラクタ。読み出すファイル名を指定する。

void Load(HeaderBase &*data*)
引数で指定した空のコンテナに、ファイルから読み出した内容を 書き込む。

void Load(ElementContainer &*data*)
引数で指定した空のコンテナに、ファイルから読み出した内容を 書き込む。

void Load(ElementContainerArray &*data*)
引数で指定した空のコンテナに、ファイルから読み出した内容を 書き込む。

void Load(ElementContainerMatrix &*data*)
引数で指定した空のコンテナに、ファイルから読み出した内容を 書き込む。

B.8. NeXusFileIO

***class* NeXusFileIO**

ElementContainer, ElementContainerArray, ElementContainerMatrix を 最小限の手順でファイルに保存したり、 保存したファイルを読み込んだりできる。

ただし、ファイルの内容の変更の機能は実装していない。

B.8.1. メソッド　主要一覧

NeXusFileIO()

コンストラクタ

void Write(ElementContainerMatrix *M*, string *FileName*, string *UserName*, UInt4 *CompMode* = 1)

第 4 引数 CompMode=1 であると圧縮モードが有効になる。 1 以外を指定すると圧縮しない。また、第四引数を省略すると自動的に 1 が入り 圧縮モードとなる。 非圧縮で使用するのが望ましい。

void Write(ElementContainerArray *A*, string *FileName*, string *UserName*, UInt4 *CompMode* = 1)

同上。ElementContainerArray を保存する。

void Write(ElementContainer *C*, string *FileName*, string *UserName*, UInt4 *CompMode* = 1)

同上。ElementContainer を保存する。

ElementContainerMatrix ReadElementContainerMatrix(string *FileName*)

ElementContaierMatrix が記録されたファイルを読み出す。

ElementContainerArray ReadElementContainerArray(string *FileName*)

ElementContaierArray が記録されたファイルを読み出す。

ElementContainer ReadElementContainer(string *FileName*)

ElementContaier が記録されたファイルを読み出す。

B.9. CppToPython

***class* CppToPython**

万葉ライブラリで主に使用されるデータコンテナは、vector である。しかし使用する環境は Python であるため、特に配列のデータは Python ではリスト、万葉ライブラリ内では vector と、両者間の変換が必要である。そのためのクラスである。

B.9.1. メソッド　主要一覧

CppToPython()

コンストラクタ。

~CppToPython()

デストラクタ。

PyObject *VectorToList(vector<Int4> *v*)
PyObject *VectorToList(vector<UInt4> *v*)
PyObject *VectorToList(vector<Double> *v*)

C++の、整数配列 vector<Int4>、正の整数配列 vector<UInt4>、実数配列 vector<Double>、を Python リストへ変換する。*PyObject* はここでは Python リストの意味。

PyObject *VectorStringToList(vector<string> *v*)

C++の文字列の配列 vector<string>を Python リストに変換する。 *PyObject* はここでは Python リストの意味。

vector<Int4> ListToInt4Vector(PyObject **List*)

Python リストを C++の整数配列 vector<Int4>へ変換する。 *PyObject* はここでは Python リストの意味。

vector<UInt4> ListToUInt4Vector(PyObject **List*)

Python リストを C++の正の整数配列 vector<UInt4>へ変換する。 *PyObject* はここでは Python リストの意味。

vector<Double> ListToDoubleVector(PyObject **List*)

Python リストを C++の実数配列 vector<Double>へ変換する。 *PyObject* はここでは Python リストの意味。

vector<string> ListToStirngVector(PyObject **List*)

Python リストを C++の文字列の配列 vector<string>へ変換する。 *PyObject* はここでは Python リストの意味。

B.10. MLF constants

MLF モジュールで定義されている定数を紹介する。Python から使用するときは以下のように **Manyo.MLF** モジュールをインポートすれば使うことができる。

```
>>> import Manyo.MLF as mm
>>> mm.MLF_MASS_NEUTRON
```

B.10.1. 主な定数

***const* double MLF_PI**
円周率

***const* double MLF_MASS_NEUTRON**
***const* double MLF_Mn**
中性子の質量 [kg]

***const* double MLF_PLANCK**
プランク定数 [Js]

***const* double MLF_HBAR**
プランク定数／２＊円周率 [Js]

***const* double MLF_NA**
アボガドロ数 [1/mol]

***const* double MLF_kB**
ボルツマン定数 [J/K]

***const* double MLF_KG2G**
1 [kg] = 1000 [g]の単位変換用定数

***const* double MLF_MM2M**
1 [mm] = 1e-03 [m]の単位変換用定数

***const* double MLF_M2MM**
1 [m] = 1000 [mm]の単位変換用定数

***const* double MLF_ANGSTROM2M**
1 [A] = 1.e-10 [m]の単位変換用定数

***const* double MLF_M2ANGSTROM**
1 [m] = 1e10 [Å]の単位変換用定数

***const* double MLF_FM2M**
1 [fm] = 1.e-15 [m]の単位変換用定数

***const* double MLF_M2FM**
1 [m] = 1.e15 [fm]の単位変換用定数

***const* double MLF_SEC2MICROSEC**
1 [sec] = 1.e+06 [microsec]の単位変換用定数

***const* double MLF_MICROSEC2SEC**
1 [microsec] = 1.e-06 [sec]の単位変換用定数

***const* double `MLF_EV2J`**
1 [eV] = 1.6 [J]の単位変換用定数

***const* double `MLF_J2EV`**
1 [J] = 1/1.6 [eV]の単位変換用定数

***const* double `MLF_MEV2J`**
1meV=1.e-03eV の単位変換用定数

***const* double `MLF_J2MEV`**
1 [eV] = 1.e03 [meV]の単位変換用定数

***const* double `MLF_BARN2M2`**
1 [barn] = 1.e-24 [cm^2] = 1.e-28[m^2]の単位変換用定数

***const* double `MLF_M22BARN`**
1 [cm^2] = 1.e4 [m^2] = 1.e24 [barn]の単位変換用定数

***const* double `MLF_ATM2PA`**
1 [atm] = 101325 [Pa]の単位変換用定数

***const* double `MLF_PA2ATM`**
1 [Pa] = 1.-/101325.0 [atm]の単位変換用定数

***const* double `MLF_DEGREE2RADIAN`**
180 [degree] = PI [radian]の単位変換用定数

***const* double `MLF_RADIAN2DEGREE`**
PI [radian] = 180 [degree]の単位変換用定数

This is a blank page.

国際単位系（SI）

表１．SI 基本単位

基本量	SI 基本単位	
	名称	記号
長さ	メートル	m
質量	キログラム	kg
時間	秒	s
電流	アンペア	A
熱力学温度	ケルビン	K
物質量	モル	mol
光度	カンデラ	cd

表２．基本単位を用いて表されるSI組立単位の例

組立量	SI 組立単位	
	名称	記号
面積	平方メートル	m^2
体積	立方メートル	m^3
速さ，速度	メートル毎秒	m/s
加速度	メートル毎秒毎秒	m/s^2
波数	毎メートル	m^{-1}
密度，質量密度	キログラム毎立方メートル	kg/m^3
面積密度	キログラム毎平方メートル	kg/m^2
比体積	立方メートル毎キログラム	m^3/kg
電流密度	アンペア毎平方メートル	A/m^2
磁界の強さ	アンペア毎メートル	A/m
量濃度(a)，濃度	モル毎立方メートル	mol/m^3
質量濃度	キログラム毎立方メートル	kg/m^3
輝度	カンデラ毎平方メートル	cd/m^2
屈折率(b)	（数字の） 1	1
比透磁率(b)	（数字の） 1	1

(a) 量濃度（amount concentration）は臨床化学の分野では物質濃度（substance concentration）ともよばれる。
(b) これらは無次元量あるいは次元１をもつ量であるが、そのことを表す単位記号である数字の１は通常は表記しない。

表３．固有の名称と記号で表されるSI組立単位

組立量	SI 組立単位			
	名称	記号	他のSI単位による表し方	SI基本単位による表し方
平面角	ラジアン(b)	rad	1 (b)	m/m
立体角	ステラジアン(b)	sr(c)	1 (b)	m^2/m^2
周波数	ヘルツ(d)	Hz		s^{-1}
力	ニュートン	N		$m\ kg\ s^{-2}$
圧力，応力	パスカル	Pa	N/m^2	$m^{-1}\ kg\ s^{-2}$
エネルギー，仕事，熱量	ジュール	J	N m	$m^2\ kg\ s^{-2}$
仕事率，工率，放射束	ワット	W	J/s	$m^2\ kg\ s^{-3}$
電荷，電気量	クーロン	C		s A
電位差（電圧），起電力	ボルト	V	W/A	$m^2\ kg\ s^{-3}\ A^{-1}$
静電容量	ファラド	F	C/V	$m^{-2}\ kg^{-1}\ s^4\ A^2$
電気抵抗	オーム	Ω	V/A	$m^2\ kg\ s^{-3}\ A^{-2}$
コンダクタンス	ジーメンス	S	A/V	$m^{-2}\ kg^{-1}\ s^3\ A^2$
磁束	ウエーバ	Wb	Vs	$m^2\ kg\ s^{-2}\ A^{-1}$
磁束密度	テスラ	T	Wb/m^2	$kg\ s^{-2}\ A^{-1}$
インダクタンス	ヘンリー	H	Wb/A	$m^2\ kg\ s^{-2}\ A^{-2}$
セルシウス温度	セルシウス度(e)	℃		K
光束	ルーメン	lm	cd sr(c)	cd
照度	ルクス	lx	lm/m^2	m^{-2} cd
放射性核種の放射能(f)	ベクレル(d)	Bq		s^{-1}
吸収線量，比エネルギー分与，カーマ	グレイ	Gy	J/kg	$m^2\ s^{-2}$
線量当量，周辺線量当量，方向性線量当量，個人線量当量	シーベルト(g)	Sv	J/kg	$m^2\ s^{-2}$
酸素活性	カタール	kat		s^{-1} mol

(a)SI接頭語は固有の名称と記号を持つ組立単位と組み合わせても使用できる。しかし接頭語を付した単位はもはやコヒーレントではない。
(b)ラジアンとステラジアンは数字の１に対する単位の特別な名称で、量についての情報をつたえるために使われる。実際には、使用する時には記号radおよびsrが用いられるが、習慣として組立単位としての記号である数字の１は明示されない。
(c)測光学ではステラジアンという名称と記号srを単位の表し方の中に、そのまま維持している。
(d)ヘルツは周期現象についてのみ、ベクレルは放射性核種の統計的過程についてのみ使用される。
(e)セルシウス度はケルビンの特別な名称で、セルシウス温度を表すために使用される。セルシウス度とケルビンの単位の大きさは同一である。したがって、温度差や温度間隔を表す数値はどちらの単位で表しても同じである。
(f)放射性核種の放射能（activity referred to a radionuclide）は、しばしば誤った用語で"radioactivity"と記される。
(g)単位シーベルト（PV,2002,70,205）についてはCIPM勧告2（CI-2002）を参照。

表４．単位の中に固有の名称と記号を含むSI組立単位の例

組立量	SI 組立単位		
	名称	記号	SI 基本単位による表し方
粘度	パスカル秒	Pa s	$m^{-1}\ kg\ s^{-1}$
力のモーメント	ニュートンメートル	N m	$m^2\ kg\ s^{-2}$
表面張力	ニュートン毎メートル	N/m	$kg\ s^{-2}$
角速度	ラジアン毎秒	rad/s	$m\ m^{-1}\ s^{-1}=s^{-1}$
角加速度	ラジアン毎秒毎秒	rad/s^2	$m\ m^{-1}\ s^{-2}=s^{-2}$
熱流密度，放射照度	ワット毎平方メートル	W/m^2	$kg\ s^{-3}$
熱容量，エントロピー	ジュール毎ケルビン	J/K	$m^2\ kg\ s^{-2}\ K^{-1}$
比熱容量，比エントロピー	ジュール毎キログラム毎ケルビン	J/(kg K)	$m^2\ s^{-2}\ K^{-1}$
比エネルギー	ジュール毎キログラム	J/kg	$m^2\ s^{-2}$
熱伝導率	ワット毎メートル毎ケルビン	W/(m K)	$m\ kg\ s^{-3}\ K^{-1}$
体積エネルギー	ジュール毎立方メートル	J/m^3	$m^{-1}\ kg\ s^{-2}$
電界の強さ	ボルト毎メートル	V/m	$m\ kg\ s^{-3}\ A^{-1}$
電荷密度	クーロン毎立方メートル	C/m^3	m^{-3} s A
表面電荷	クーロン毎平方メートル	C/m^2	m^{-2} s A
電束密度，電気変位	クーロン毎平方メートル	C/m^2	m^{-2} s A
誘電率	ファラド毎メートル	F/m	$m^{-3}\ kg^{-1}\ s^4\ A^2$
透磁率	ヘンリー毎メートル	H/m	$m\ kg\ s^{-2}\ A^{-2}$
モルエネルギー	ジュール毎モル	J/mol	$m^2\ kg\ s^{-2}\ mol^{-1}$
モルエントロピー，モル熱容量	ジュール毎モル毎ケルビン	J/(mol K)	$m^2\ kg\ s^{-2}\ K^{-1}\ mol^{-1}$
照射線量（X線及びγ線）	クーロン毎キログラム	C/kg	kg^{-1} s A
吸収線量率	グレイ毎秒	Gy/s	$m^2\ s^{-3}$
放射強度	ワット毎ステラジアン	W/sr	$m^4\ m^{-2}\ kg\ s^{-3}=m^2\ kg\ s^{-3}$
放射輝度	ワット毎平方メートル毎ステラジアン	$W/(m^2\ sr)$	$m^2\ m^{-2}\ kg\ s^{-3}=kg\ s^{-3}$
酵素活性濃度	カタール毎立方メートル	kat/m^3	$m^{-3}\ s^{-1}$ mol

表５．SI 接頭語

乗数	名称	記号	乗数	名称	記号
10^{24}	ヨタ	Y	10^{-1}	デシ	d
10^{21}	ゼタ	Z	10^{-2}	センチ	c
10^{18}	エクサ	E	10^{-3}	ミリ	m
10^{15}	ペタ	P	10^{-6}	マイクロ	μ
10^{12}	テラ	T	10^{-9}	ナノ	n
10^9	ギガ	G	10^{-12}	ピコ	p
10^6	メガ	M	10^{-15}	フェムト	f
10^3	キロ	k	10^{-18}	アト	a
10^2	ヘクト	h	10^{-21}	ゼプト	z
10^1	デカ	da	10^{-24}	ヨクト	y

表６．SIに属さないが、SIと併用される単位

名称	記号	SI 単位による値
分	min	1 min=60 s
時	h	1 h =60 min=3600 s
日	d	1 d=24 h=86 400 s
度	°	1°=(π/180) rad
分	’	1’=(1/60)°=(π/10 800) rad
秒	”	1”=(1/60)’=(π/648 000) rad
ヘクタール	ha	1 ha=1 $hm^2=10^4m^2$
リットル	L, l	1 L=1 l=1 $dm^3=10^3cm^3=10^{-3}m^3$
トン	t	1 t=10^3 kg

表７．SIに属さないが、SIと併用される単位で、SI単位で表される数値が実験的に得られるもの

名称	記号	SI 単位で表される数値
電子ボルト	eV	1 eV=1.602 176 53(14)×10^{-19}J
ダルトン	Da	1 Da=1.660 538 86(28)×10^{-27}kg
統一原子質量単位	u	1 u=1 Da
天文単位	ua	1 ua=1.495 978 706 91(6)×10^{11}m

表８．SIに属さないが、SIと併用されるその他の単位

名称	記号	SI 単位で表される数値
バール	bar	1 bar=0.1MPa=100 kPa=10^5Pa
水銀柱ミリメートル	mmHg	1 mmHg≈133.322Pa
オングストローム	Å	1 Å=0.1nm=100pm=10^{-10}m
海里	M	1 M=1852m
バーン	b	1 b=100fm^2=$(10^{-12}cm)^2$=$10^{-28}m^2$
ノット	kn	1 kn=(1852/3600)m/s
ネーパ	Np	SI単位との数値的な関係は、対数量の定義に依存。
ベル	B	
デシベル	dB	

表９．固有の名称をもつCGS組立単位

名称	記号	SI 単位で表される数値
エルグ	erg	1 erg=10^{-7} J
ダイン	dyn	1 dyn=10^{-5}N
ポアズ	P	1 P=1 dyn s cm^{-2}=0.1Pa s
ストークス	St	1 St =1$cm^2\ s^{-1}$=$10^{-4}m^2\ s^{-1}$
スチルブ	sb	1 sb =1cd cm^{-2}=10^4cd m^{-2}
フォト	ph	1 ph=1cd sr cm^{-2} =10^4lx
ガル	Gal	1 Gal =1cm s^{-2}=$10^{-2}ms^{-2}$
マクスウエル	Mx	1 Mx = 1G cm^2=10^{-8}Wb
ガウス	G	1 G =1Mx cm^{-2} =10^{-4}T
エルステッド(a)	Oe	1 Oe ≙ $(10^3/4\pi)$A m^{-1}

(a) ３元系のCGS単位系とSIでは直接比較できないため、等号「 ≙ 」は対応関係を示すものである。

表10．SIに属さないその他の単位の例

名称	記号	SI 単位で表される数値
キュリー	Ci	1 Ci=3.7×10^{10}Bq
レントゲン	R	1 R = 2.58×10^{-4}C/kg
ラド	rad	1 rad=1cGy=10^{-2}Gy
レム	rem	1 rem=1 cSv=10^{-2}Sv
ガンマ	γ	1 γ=1 nT=10^{-9}T
フェルミ		1 フェルミ=1 fm=10^{-15}m
メートル系カラット		1 メートル系カラット= 0.2 g = 2×10^{-4}kg
トル	Torr	1 Torr = (101 325/760) Pa
標準大気圧	atm	1 atm = 101 325 Pa
カロリー	cal	1 cal=4.1858J（「15℃」カロリー），4.1868J（「IT」カロリー），4.184J（「熱化学」カロリー）
ミクロン	μ	1 μ =1μm=10^{-6}m

（第8版，2006年）