# MLF 向け DAQ Middleware 2009.07 版
# モニターコンポーネント実装解説

**高エネルギー加速器研究機構**
**素粒子原子核研究所**
**千代浩司**

2009 年 7 月 9 日

**概要**

MLF 中性子向け DAQ Middleware 2009.07 版モニターコンポーネントを改造して独自のモニターコンポーネントを開発するかたがた向けに 2009.07 版モニターコンポーネントの実装を解説する。

## 目次

# 1 実装したモニターコンポーネントの機能

このモニターコンポーネントは MLF 中性子で NEUNET モジュールを使用してデータ取得を行う実験で使用されることを念頭に置き開発したものであり、開発目的は DAQ-Middleware の枠組のなかでどのようにデータを上流コンポーネントから取得し、ヒストグラム化するかを示す点にある。

実装したモニターコンポーネントの機能は以下のとおりである。

- 各 PSD モジュール毎の 1 次元ポジションヒストグラム PNG ファイルの作成
- 各 PSD モジュール毎の 1 次元 TOF ヒストグラム PNG ファイルの作成
- 各 CPUDAQ の 2 次元ポジションヒストグラム PNG ファイルの作成

出力する PNG ファイルのファイル名は以下のようにした。

| | |
|---|---|
| 1 次元ポジションヒストグラム | `pos_<DaqId>_<modNo>_<PSD 番号>.png` |
| 1 次元 TOF ヒストグラム | `tof_<DaqId>_<modNo>_<PSD 番号>.png` |
| 2 次元ポジションヒストグラム | `pos_2d_<DaqId>.png` |

DaqId はその CPUDAQ の番号で 0 から始まる。modNo はその CPUDAQ がデータ収集を行う NEUNET モジュールの番号で 0 から始まる。一つの NEUNET モジュールが扱う PSD 検出器は 8 台あるので PSD 番号は 0〜7 とふることにした。

イベントデータのヒストグラム化は Manyo ライブラリを通じて gsl ヒストグラムを使用する。PNG ファイル作成は Manyo ライブラリで実装されている gnuplot interface を使って行っている。PSD ピクセル (後述) は等分割であり、各ヒストグラムのビン幅は一定で、TOF の最小値、最大値は全ての NEUNET モジュールを通じて共通である。

ユーザーが Condition データベースを通じて設定可能なパラメータは以下の通りである。

- TOF 最小値、最大値
- 1 次元ポジションヒストグラムのビン数 (2 次元ポジションヒストグラムのビン数もこれと同じに設定される)
- PNG ファイルを作るタイミング (1 daq_run() 単位で指定する)

タイムフォーカシング、$\Delta T/T$ でのヒストグラムの作成、あるいはビン幅を PSD ごとに変更する、あるいはビン幅を 1PSD 内で変更する等の機能は実装していない。

# 2 Manyo ライブラリ

Manyo ライブラリは MLF 中性子でデータ解析をするために作られたライブラリである。以下でモニターコンポーネントで使用した Manyo ライブラリの使いかた、およびそのデータ構造につ

いて簡単にまとめておく。

Manyo ライブラリでは PSD 検出器 1 台をその位置について多数の小片 (ピクセル) に分割し、ピクセル毎に TOF ヒストグラムを作る。位置ごとのイベント数を知りたい場合は TOF ヒストグラム内のイベント数の和を取ることになる。どのピクセルにイベントが発生したかはイベントデータ中の左右のパルスハイトから計算する。計算に用いるパラメータは Manyo ライブラリを使用するユーザーが SetPsdParam() を使って行う。

イベントデータのインクリメントはコードは

```
void EventDataToHistogramBase::
Increment(UInt4 Channel, UChar* data, UInt4 size) {

    UChar *CopyData = new UChar [ size * 8 ];
    for (UInt4 i = 0; i < size*8; i++) {
        CopyData[i] = data[i];
    }

    UInt4 index = 0;
    for (UInt4 i = 0; i < size; i++) {
        // 0x5a --- event data      = 90
        // 0x5b --- T0 data         = 91
        // 0x5c --- time stamp data = 92
        index = i * 8;

        if(*(data+index) == 90 && CheckFlag == 1) { // event data
            Decode_EventData(CopyData+index, &_psd_num, &_k, &_ph_l, &_ph_r, &_tof);
            UInt4 psd_id   = PutPsdId(Channel, _ModuleId, _psd_num );
            UInt4 pixel_id = PutPixelId(psd_id, PutPixelPosition(psd_id, _ph_l, _ph_r));
            ((*gslhist)[pixel_id])->Increment(
                ClockToMicrosec(_tof)*TimeFocParamC1(pixel_id) +
                TimeFocParamC0(pixel_id));
        }
        else if(*(data+index) == 91){ // T0 data
            CheckFlag = 1;
            Decode_T0Data(CopyData+index, &_CrateId, &_ModuleId, &_PulseId);
        }
        else if(*(data+index) == 92){ // time stamp data
            Decode_TimeStampData(CopyData+index, &_t1, &_t2, &_t3, &_t4, &_t5, &_t6);
        }
    }
    delete [] CopyData;
}
```

のようになっていて、T0 データにより入力イベントデータストリームのモジュール番号 (_ModuleId) を判定する。T0 データがくるまでのイベントデータはインクリメントされない。モジュール番号がわかったらそのモジュール番号はその後そのデータストリームで変化しないと仮定し、_ModuleId は一定値をとり続ける。モジュール番号確定後、イベントデータがやってきたときにはデコードして

1. PutPsdId() によりそのイベントデータの psd_id を判定する。
2. PutPixelId() によりそのイベントデータの pixel_id を判定する。

3. pixel_id の確定によりインクリメントするべきヒストグラムが判定できたのでそのヒストグラムの該当する TOF ビンのカウントをインクリメントする

という動作を行う。Increment() に関し Manyo ライブラリを使用するユーザーが定義しなければならないメソッドは

- PutPsdId()
- PutPixelId()

のふたつである。

上でのべたように、Manyo ライブラリで実装されている Increment() はインクリメントするデータストリームはひとつの NEUNET モジュールから送られてくるもので途中で他の NEUNET モジュールからのデータが混じることはないという仮定のもとで実装されている。今回実装したモニターコンポーネントでは、上流の Gatherer は複数の NEUNET モジュールからデータを取得し、そのデータは Disptcher コンポーネントを通じて Monitor コンポーネントに送られてくる。したがって Monitor コンポーネントで受け取るデータストリーム中には複数の NEUNET モジュールからのデータが存在する。このため Manyo ライブラリで実装されている Increment() はそのままでは使用できなかった。そこで、Increment() をオーバーライドしてモニターコンポーネント用 Increment() メソッドを実装した。実装当時はオリジナルの Manyo ライブラリの Increment() に上に見られる TimeForcParamC0()、TimeForcParamC1() は入っていなかったのでこれらはモニターで使用されている Increment() には入っていない。この TimeForcParamC0() の機能を実装したい場合は後述 EventDataToHistogramEmulator.cpp の Increment() を変更し、TimeForcParamC0() メソッド、TimeForcParamC1() メソッドを実装すればよい。

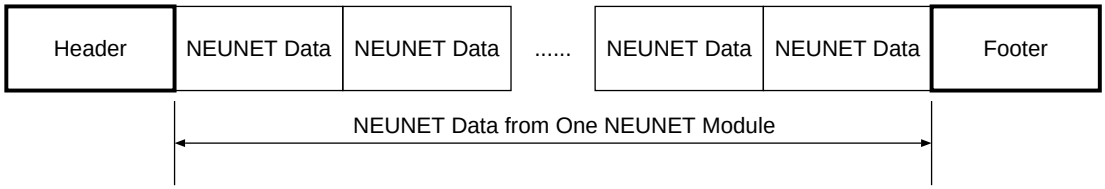


**図** 1　Dispatcher **から送られてくるデータフォーマット。ヘッダとフッタは** Gatherer **が付ける。ヘッダ、フッタのフォーマットについては図** 2 **を参照。**Monitor **コンポーネントで上流からのデータを読むと** 1 **回のリードでこの図の** Header **から** Footer **までのデータ** (Header **および** Footer **を含む**) **が読める。**Header **と** Footer **に挟まれた** NEUNET **データは必ず** 1 NEUNET **モジュールからのデータである。次にリードした場合は通常違う** NEUNET **モジュールからのデータである。**

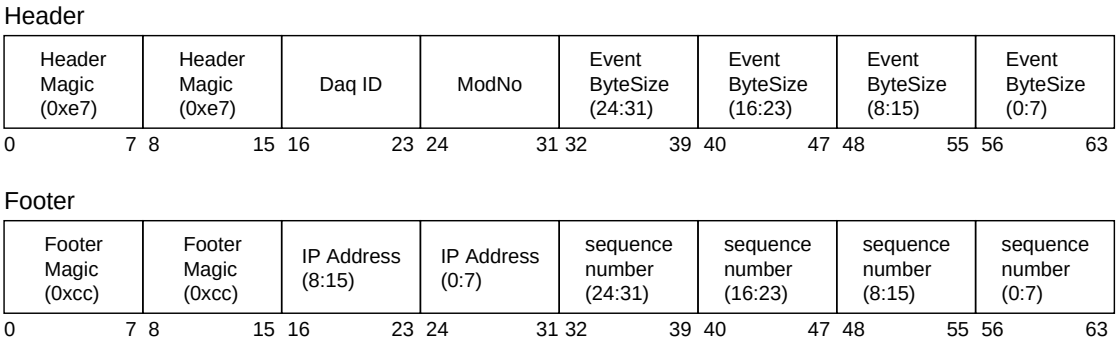


**図** 2　Gatherer **コンポーネントが付加する** Header**、**Footer **データフォーマット。**Header**、**Footer **とも** 8 **バイトである。それぞれ先頭** 2 **バイトがマジック。**

# 3　上流 DAQ コンポーネントからのデータフォーマット

NEUNET **モジュールからのデータには** 4 **種類のデータ** (**イベントデータ、**T0 **データ、装置時刻データ、外部** time clock **データ**) **がある。これらをここでは総称して** NEUNET **データと呼ぶことにする。**

Gatherer **コンポーネントで取得したデータは** Dispatcher **コンポーネントを通じて** Monitor **コンポーネントに送られてくる。**Monitor **コンポーネントで読んだときのデータフォーマットを図** 1 **に示す。また図中の** Header**、**Footer **のデータフォーマットを図** 2 **に示す。**

Monitor **コンポーネントで上流からのデータを読むと** 1 **回のリードで図** 1 **の** Header **から** Footer **までのデータ** (Header **および** Footer **を含む**) **が読める。**

Header **と** Footer **に挟まれた** NEUNET **データは必ず** 1 NEUNET **モジュールからのデータである。次にリードしたときの** NEUNET **データは通常は違う** NEUNET **モジュールからのデータである。**

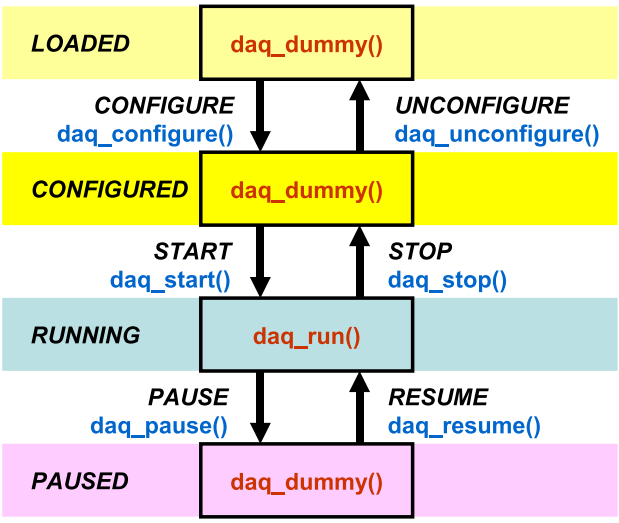


図 3　DAQ-Middleware で実装されている状態、およびその遷移。daq_dummy()、daq_run() はその状態中で繰り返す呼ばれるメソッドを示している。各状態から次の状態に遷移するときに呼ばれるメソッドには daq_configure()、daq_start() 等がある。これらのメソッドを実装することで DAQ コンポーネントを作ることができる。

# 4　実装方針

DAQ-Middleware で用意されている状態遷移図およびその際に呼ばれるメソッドを図 3 に示す。図中の daq_configure()、daq_start() 等の各メソッドを実装することで DAQ コンポーネントを作ることができる。

Monitor コンポーネントの実装では `daq_configure()` でヒストグラムを作成するためのオブジェクトの作成、各パラメータの取得を行い、`daq_run()` 内で上流コンポーネントからのデータの読み取り、ヒストグラムのインクリメント、ヒストグラム PNG ファイルの書きだしを実装し、`daq_stop()` でヒストグラムデータのリセットを行うことにした。

具体的には `daq_configure()` で DAQ オペレータから送られてくる config.xml に書かれたパラメータの読みだし、および condition.json ファイルに書かれたパラメータの読みだしを行う。`daq_run()` で上流 DAQ コンポーネントから送られてきたデータ中の header および footer からモジュール番号を取り出し Manyo ライブラリを使ってヒストグラムデータをインクリメントする。一定回数 `daq_run()` したらヒストグラムデータからデータを取り出し Manyo ライブラリの gnuplot インターフェイスを使用して PNG ファイルを作成する、とした。

## 4.1　DAQ システムナンバリングスキーム

NEUNET モジュール等の識別には以下のようなスキームを採用した。

`DaqId`

CPUDAQ を識別する番号。0 から始まり、各 CPUDAQ ごとに 1 づつインクリメントされる。

`modNo`

各 NEUNET モジュールの番号。DaqId で識別される CPUDAQ の最初の NEUNET モジュールの modNo は 0。以下 1 づつ増えていく。
20 枚の NEUNET モジュールがあるならその modNo は 0〜19 である。CPUDAQ が複数ある場合には、modNo が同じになる NEUNET モジュールが複数存在することになる。したがってこの番号は NEUNET モジュールの identify には使えないので modNo と No(ナンバー) を使うことにした。

## 4.2 各パラメータの情報源

各パラメータの情報源をまとめておく。

- daqId は、DAQ オペレータを通じて config.xml に書かれたものが取得できるので Gatherer が header に追加するものと比較し正しい値であることを確認の上、header にある値を使用することにした。daqId は PNG ファイルのファイル名決定にのみ使用している。
- モジュール番号 `ModNo` については Gatherer が header に追加するモジュール番号を使用することにした。
- 各 NEUNET モジュールに接続されている 8 個の PSD 検出器の識別はイベントデータ中に入っている値をデコードして使用した。
- ポジションヒストグラムのピクセル数、TOF ヒストグラムの最小値、最大値、ビン数、PNG ファイルを作るタイミングの指定は Condition データベースから取得することにした。
- PNG ファイル出力先ディレクトリ、gnuplot の実行プログラムのフルパスは DAQ オペレータを通じて config.xml から取得することにした。

次節で具体的にソースコードの解説を行う。

# 5 ソースコード解説

## 5.1 ファイル名

モニターコンポーネントのソースコードは以下のファイルである。

- EventDataToHistogramEmulator.h
- EventDataToHistogramEmulator.cc
- ConditionMonitor.h
- ConditionMonitor.cpp
- MonitorComp.cpp
- Monitor.cpp
- MonitorCreatePng.cpp
- cformat.cpp

EventDataToHistogramEmulator.{h,cpp}内で Manyo ライブラリでヒストグラムを作成するのに必要な PutPsdId()、PutPixelId() を実装している。また前述の理由により Manyo ライブラリで実装されているイベントデータをインクリメントするメソッド Increment() を使用することができなかったので Monitor コンポーネント用の Increment() を実装し、これらのファイル内に置いた。ランごとに変わるかもしれないパラメータは Condition データベースとして設定可能なようにした。これらを ConditionMonitor.{h,cpp}内で実装している。モニターコンポーネント本体のコードが MonitorComp.cpp および Monitor.cpp である。このうち MonitorComp.cpp はユーザーが変更する必要はほとんどない。ヒストグラム PNG ファイルの作成は Monitor.cpp 内の daq_run() で create_png_files() を呼ぶことで行っている。create_png_files() の実装は MonitorCreatePng.cpp 内にまとめた。gnuplot インターフェイスを使用する上で、printf() 関数のように呼び出せて std::string に変換する関数があると便利だったので cformat() という関数を cformat.cpp 内で実装した。

## 5.2 EventDataToHistogramEmulator.h および.cpp

EventDataToHistogramEmulator クラスは EventDataToHistogramBase クラスを継承している。

コンストラクタ内で SetParam() を読んでヒストグラムに必要な各パラメータをセットしている。

ピクセルの定義は単純に

```
// set PSD parameters for each PSD
vector<Double> A(NumOfPsd);
vector<Double> B(NumOfPsd);
vector<Double> C(NumOfPsd);
```

```
for (UInt4 i = 0; i < NumOfPsd; i++) {
    A[i] = (Double) (num_of_pixel_per_psd - 1.0);
    B[i] = 1.0;
    C[i] = 0.0;
}
for(UInt4 i = 0; i < NumOfPsd; i++) {
    SetPsdParam(i, A[i], B[i], C[i]);
}
```

とした。また、Manyo ライブラリを使用する上で必ず実装しなければならない PutPsdId() および PutPixelId() は単純に

```
//PutPsdId()
UInt4 EventDataToHistogramEmulator::
PutPsdId(UInt4 DaqChannel, UInt4 module, UInt4 psd, UInt4 modNo) {
    return DaqChannel + m_num_of_psd_per_module * modNo + psd;
}
```

とした。modNo は Gatherer が追加する Header に含まれているものを get_modNo() を使って取得している。

```
//PutPixelId()
PutPixelId(UInt4 PsdId, UInt4 PixelPosition) {
    return PsdId * m_num_of_pixel_per_psd + PixelPosition;
}
```

とした。

Increment() は Manyo ライブラリの実装は

```
void Increment( UInt4 DaqChannel, UChar* data, UInt4 size );
```

であったが、モジュール番号を入力として与える必要があったので

```
void Increment(UInt4 Channel, UChar* data, UInt4 size, UInt4 modNo)
```

とした。

## 5.3 ConditionMonitor.h および.cpp

Condition データベースを使って以下のパラメータを指定できるようにした。Condition データベースに関するコードは「Condition データベースの開発マニュアル」に書かれているコードを、パラメータ名をおきかえただけでそのまま流用している。各パラメータの値は monitorParam 構造体のメンバーとしてセットしている。

tof_min

TOF 最小値 (単位 usec)

`tof_max`

TOF 最大値 (単位 usec)

`num_of_tof_bin`

各 PSD TOF ヒストグラムのビン数

`num_of_pixel_per_psd`

1 次元ポジションヒストグラム、および 2 次元ポジションヒストグラムのピクセル数。

`monitor_update_rate`

png ファイルをアップデートするレート。ここで指定された回数 daq_run() が回ると png ファイル生成を行う。どの png ファイルを生成するかはさらに create_2d_position_file(), new_create_1d_position_files(), create_tof_files(), の関数内で決めている。

## 5.4 Monitor.cpp

### 5.4.1 config.xml で指定可能なパラメータの取得

以下のパラメータに関しては run 毎に変更する必要があまりないと考え config.xml 内で指定するようにした。各パラメータの取得は daq_configure() の parse_params() で行っている。

`daqId`

そのコンポーネントが稼働している DAQ システムの DAQ ID を指定する。

`samplingRate`

上流の Dispatcher コンポーネントのポート名、および送られてくるデータの対データ全体の割合を指定する。指定する数値は全て送られてくる場合は 1 と指定し、1/10 送られてくるときには 10 と指定する。1/50 送られてくるときには 50 と指定する。

`gnuplot_path`

ヒストグラム PNG ファイルの作成に使用する gnuplot のフルパスを指定する。

`png_output_dir`

gnuplot が作成したヒストグラム PNG ファイルの出力先ディレクトリをフルパスで指定する。

`num_of_psd_per_module`

NEUNET モジュール毎の PSD 検出器数。通常 8。

### 5.4.2 データのインクリメント、PNG ファイルの作成

データのインクリメント、PNG ファイルの作成は daq_run() の最後のほうで

```
m_evs->Increment(0, &m_cdata[0], event_byte_size / EVENT_BYTE_SIZE, modNo);
if (m_loop % m_monitorParam.monitor_update_rate == 0) {
```

```
        create_png_files();
}
```

として行っている。

## 5.5 MonitorCreatePng.cpp

ヒストグラム PNG ファイルの作成を行うメソッドを MonitorCreatePng.cpp ファイルにまとめた。

ファイルを書くタイミングについてはあまり頻繁に作ると計算機に負荷がかかりすぎ、上流からのデータを受け取ることができずその影響が Gatherer まで波及し、データ読み取りに影響するので適当に間引く必要があった。そこで 1 daq_run() ごとにインクリメントされる変数 m_loop を元に

- m_loop % monitorParam.monitor_update_rate が 0 なら 2 次元ヒストグラムを書く。
- m_loop % monitorParam.monitor_update_rate が 0 のとき 1 次元ポジション PNG ファイルおよび TOF PNG ファイルを書くが、どのモジュールのヒストグラムを書くかの決定は、m_loop / monitorParam.monitor_update_rate を計算すると 1, 2, 3, $\cdots$ という自然数列が得られるので、この自然数列を総モジュール数で割り、その余りの数のモジュール番号のヒストグラムを書くことにした。

たとえば monitorParam.monitor_update_rate が 1000 と設定され、モジュールが 3 個あった場合は次のように PNG ファイルを書く。

| m_loop | 書くヒストグラム |
|---|---|
| 1000 | 2 次元ポジション、ModNo 1 のポジションおよび ModNo 1 の TOF ヒストグラム |
| 2000 | 2 次元ポジション、ModNo 2 のポジションおよび ModNo 2 の TOF ヒストグラム |
| 3000 | 2 次元ポジション、ModNo 0 のポジションおよび ModNo 0 の TOF ヒストグラム |
| 4000 | 2 次元ポジション、ModNo 1 のポジションおよび ModNo 1 の TOF ヒストグラム |
| 5000 | 2 次元ポジション、ModNo 2 のポジションおよび ModNo 2 の TOF ヒストグラム |
| 6000 | 2 次元ポジション、ModNo 0 のポジションおよび ModNo 0 の TOF ヒストグラム |

MonitorCreatePng.cpp ファイルにまとめてある。gsl ヒストグラムにインクリメントされたデータからヒストグラム図を作成するために Manyo ライブラリで実装されている GnuplotInterface を利用した。

利用したメソッドは、1 次元ポジションヒストグラムについては PutSum()(各 Pixel ごとにある TOF ヒストグラム中に含まれるカウント数の和を取り出すメソッド) であり、1 次元 TOF ヒストグラムについては TofVsCount()(gsl ヒストグラムからデータを抽出し TOF ヒストグラム PNG

ファイルを作成するメソッド) である。これらは gnuplot に対してパイプでデータを投入する形になっている。

2 次元ポジションヒストグラムについては、データ数が多くなりパイプでデータを投入すると 1 枚の PNG ファイルを作成するのに数秒程度かかることがわかった。ファイルからデータを投入すると 0.3 秒程度で PNG ファイルを作成できることが判明したので、2 次元ヒストグラムについては tmpfs の/dev/shm に一度データファイルを書き、gnuplot がそのデータを読んで PNG ファイルを作成するという実装を行った。データの書きだしは Create2DHistogramDataFile() というメソッドを、Manyo ライブラリの配布ファイルに追加する形で実装した。この Create2DHistogramDataFile() は本家 Manyo ライブラリには入っていない。

# 6 実験実用モニターへの改造指針

作成するヒストグラムを改造するには

1. ソースコードをコピーして別の名前にする。
2. Makefile をそれにしたがって書き換える。
3. ソースコード書き換え。
4. make

という手順になる。具体的にソースコードのどこを書き換えるかは、実現したい機能に依存するが、ヒストグラムの部分を変更したいだけであれば

- EventDataHistogramBase クラスを継承している部分 (EventDataToHistogram.h,cpp) でヒストグラムパラメータの取得と Manyo ライブラリを使用するうえで必要なパラメータの設定を行い、
- ConditionMonitor.h,cpp で必要なパラメータをセットするように変更し、
- Monitor.cpp のうち
  `////////// Manyo Monitor //////////`および
  `/////// Manyo Monitor End /////////`で囲まれた部分を書き換え、
- MonitorCreatePng.cpp 内の gnuplot へのコマンドを書き換える

という手順で行えると思う。

# 付録 A　gsl のメモリー消費量

gsl では bin の range として double(x 軸の値)、ヒストグラムのカウントとしても double(ウェイトを付けてインクリメントできるように) を使用する。dobule が 8 バイトのシステムで 1024 ビンのヒストグラムを 1 つ作るとメモリー 16kB が必要になる。これを確認するためにテストプログラムを書いて gsl ヒストグラムを一つ、ビンの数をいろいろ変えて ps コマンドで VSZ, RSZ を取得してみた。

| ビン数 | VSZ (kB) | RSZ (kB) |
|---|---|---|
| 1 * 1024 * 1024 | 20,232 | 17,068 |
| 2 * 1024 * 1024 | 36,616 | 33,460 |
| 3 * 1024 * 1024 | 53,000 | 49,840 |
| 4 * 1024 * 1024 | 69,380 | 66,228 |

次にヒストグラムの数を変えて試してみる。1024 ビンのヒストグラムを複数作って ps コマンドでメモリー消費量をみる:

| ヒストグラム数 | VSZ (kB) | RSZ (kB) |
|---|---|---|
| 1024 | 20,252 | 17,104 |
| 2048 | 36,676 | 33,532 |
| 3072 | 53,100 | 49,960 |
| 4096 | 69,524 | 66,388 |

# 付録 B　モニターコンポーネントのメモリー使用量

註: 以下の測定は各パラメータについて config.xml で指定していた時代に測定したものであるが、メモリー消費量はそんなに変化していないはずである。

10 台 NEUNET モジュールを読む config.xml でメモリー消費量を見た。gsl ヒストグラムに必要なメモリー量は 1 ピクセル毎に TOF ビン数のヒストグラムができるので 1 NEUNET モジュールあたりに必要なメモリー量は普通に考えると 16bytes * (1NEUNET モジュール総ピクセル数) * (TOF ビン数) * (NEUNET モジュール数) となる (1PSD 検出器あたりのピクセル数が 100 なら 1NEUNET モジュール総ピクセル数は 800 になる)。

Manyo ライブラリを使用する場合はレンジの指定で gsl 以外のところで確保したベクタを差すことができる場合もあるのでその場合は gsl で確保されるレンジは free() されている。この場合、ヒストグラムに必要なメモリー量の計算は 8bytes * (1NEUNET モジュール総ピクセル) * (TOF ビン数) * (NEUNET モジュール数) になる。

```
<manyo/gsl/GslHistogram.cc>
```

```
void GslHistogram::
Set( Double *Bin, UInt4 Size ){
  h = gsl_histogram_alloc( Size-1 );
  gsl_histogram_set_ranges( h, Bin, Size );
  free (h->range);
  h->range = Bin;
}
```

```
<param pid="tof_min">0</param>
<param pid="tof_max">30000</param>
<param pid="num_of_tof_bin">500</param>
<param pid="num_of_psd_per_module">8</param>
<param pid="num_of_pixel_per_psd">500</param>
<param pid="monitor_update_rate">500</param>
```

**とセットして** top **で見て** VIRT/RES **が** 244MB/166MB (**上の計算式で** gsl **ヒストグラムに必要なメモリー量を計算すると** 8bytes * (500 * 8) * 500 * 10 = 152.5MB)**。**

**次に** config.xml **でパラメータを変えて測定してみた。**

```
 <param pid="tof_min">0</param>
 <param pid="tof_max">30000</param>
!<param pid="num_of_tof_bin">1000</param>
 <param pid="num_of_psd_per_module">8</param>
 <param pid="num_of_pixel_per_psd">500</param>
 <param pid="monitor_update_rate">500</param>
```

(!**が変更点**) **で** VIRT/RES 397/318MB**。**

**さらに変更**:

```
 <param pid="tof_min">0</param>
 <param pid="tof_max">30000</param>
!<param pid="num_of_tof_bin">2000</param>
 <param pid="num_of_psd_per_module">8</param>
 <param pid="num_of_pixel_per_psd">500</param>
 <param pid="monitor_update_rate">500</param>
```

VIRT/RES 702MB/624MB

```
 <param pid="tof_min">0</param>
 <param pid="tof_max">30000</param>
 <param pid="num_of_tof_bin">500</param>
 <param pid="num_of_psd_per_module">8</param>
!<param pid="num_of_pixel_per_psd">1000</param>
 <param pid="monitor_update_rate">500</param>
```

VIR/RES 1314MB 1.2GB

# 付録 C　gsl の性能

gsl **ヒストグラムからデータを取り出すスピードを計ってみた。**

```
/* synopsis */
gsl_histogram *h = gsl_histogram_alloc(n_bin);
gsl_histogram_set_ranges_uniform(h, x_min, x_max);
read_file_and_increment(datafile, h);
gettimeofday(&start, NULL);
    for (n = 0; n < n_loop; n ++) {
        for (i = 0; i < n_bin; i++) {
            /* 全てのビンのデータの取り出し */
            y = gsl_histogram_get(h, i);
        }
    }
gettimeofday(&end, NULL);
```

Xeon 2.5GHz の計算機で n_loop=1,000,000 回、回してみた。

| n_bin | sec |
|---|---|
| 100 | 0.534335 |
| 1000 | 5.234554 |
| 10000 | 52.253156 |

gsl_histogram_get() で 8 バイト引き出すとして 1460MB/s で引き出せる (メモリー転送スピードか?)。