Gem Analysis Rewrite- Design Document

Noah Wuerfel

October 6, 2018

Contents

1 Introduction

This is the design document for the rewrite/cleanup of the GEM analysis code for the MUSE experiment hosted at the $\Pi m1$ beamline at the Paul Scherrer Institut. The physics goals of the MUSE experiment require ingoing and outgoing particle tracking with fine spatial resolution (<100μm). The GEM telescope is comprised of 6 ”Gas Electron Multiplier” detectors: two GEMs located at the intermediate focal point, upstream from the final focusing magnets of the beamline, and 4 GEMs just upstream of the target. The GEM analysis code takes digitized pulses generated by APV-25 cards from raw strip data on the detector, identifies clusters of strips to identify hit location candidates, and finally fits tracks to the clusters to provide a most-likely ingoing particle track.

A cleanup of the code was motivated by multiple crashes/ inconsistent performance of the code during summer 2018 data taking. Crashes indicated memory bugs in the old code, which could cause data corruption. The primary goals of the rewrite are two-fold: first, to substantially improve the readability/maintainability of the analysis code; and secondly, to improve the performance of the code, both in optimization and veracity (no memory errors -> less possibility of data corruption).

This document contains brief descriptions of the key components of this software: the strategy of the framework, what calibrations are required for a GEM analysis, histograms produced by the analysis, structure of data in and data out, classes and methods in the framework, and a schedule of the work to be done.

Note on style: functions are camel case, variables underscore delimited. Hence, a function called ”my function” would be written ”myFunction()” and a variable called ”my variable” would be written ”my_variable”

2 Strategy

Since we’re using the MUSE cooker, it’s worthwhile to work within the abstraction of recipes. Previous versions of the analysis code lied on a heavily branched ”process” function. Control branches were chosen by a set of booleans set in various startup procedures but this makes recipes opaque. Instead, this time the design is based around leveraging the recipe format. We may also need some init file and definitely need a mapper, but those are details of writing the plugin so who cares?

3 ’Calibration’ Requirements of the GEMs

Before extracting data from the GEMs there are 3 components to cleaning the DAQ data, roughly called a calibration step:

1. common mode gain:
   The common mode gain is an APV specific characteristic. This accounts for some non-zero offset to the ADC output across all channels on a given APV. The gain is common only to that APV, hence in the raw data we often see a sharp step in the X or Y ADC spectra because the first half of strips in an axis are managed by one APV with its own common mode gain, while the other half is managed by another APV with its own common mode gain.
   Finding the common mode gain can be done by (this should be revised later with a better method):

```
For each APV:
```

```
          Find the maximum channel
          For each channel outside (max-some, max+some):
                    total += this_channel data
          APV_cmode = total/num_apv_channels
```

2. pedestal subtraction:
   The Pedestal is the average, non-zero offset to the ADC output across all GEM channels following common mode subtraction. (I still don't understand what this does or why there should be a non-zero offset if cmode subtraction has been performed properly). Finding the pedestal can be done by (revise later with better method):

```
For each gem:
          For each channel:
                    total_adc+=channel data
          gem_pedestal = total_adc/num_channels
```

3. bkg:
   The Bkg characterizes the statistical fluctuations of the dataset, specifically the RMS x and y ADC values given by averaging over the entire ADC spectrum for the GEM. Then, these statistics data may be used to identify peaks which are statistically significant and can then be fed to the cluster finder to generate most likely clusters to fit tracks to. Finding the bkg can be done by (definitely revise with a better method):

```
For each gem:
          For each channel:
                    bkg=RMS(ADC this channel plus minus range)
```

4 I/O Data Status

This section details the status of data coming in and being put out by the analysis code. Currently, the DAQ outputs MIDAS data, which is converted to a Root tree by the Midas to Root converter. The GEM data is stored in the 'data' leaf of the "GEM1" branch of the MMT tree in the run*.root file. The GEM data is currently stored as binary information, hence it is essential to understand the format of this binary data, as well as to have a binary blob reader capable of making this data human interact-able.

The binary data input to the GEM analysis has the form:
Event Header: 1 word contains event_id, num_apvs, VME module id for the run
Data: 128 words of APV data
Footer: 1 word Hex signature (0xFFFFFFFF)

5 Constants, Data Types, and Histograms

There are a number of constants which are referenced in many places throughout the code, here's a list of them:

- NUM_GEMS 6

- NUM_APV_PER_GEM 4
- NUM_X_APV 2
- NUM_Y_APV 2
- NUM_MAX_CHAN_PER_APV 128
- NUM_CHAN_APV_1 122
- NUM_CHAN_APV_2 128
- NUM_CHAN_PER_AXIS (NUM_CHAN_APV_1 + NUM_CHAN_APV_2)
- NUM_CHAN_PER_GEM 2*(NUM_CHAN_PER_AXIS)
- NUM_NEIGHBOR_CHANNELS 2
- NUM_MAX_CLUSTER_CANDIDATES 5

There are a number of ancillary data types defined for this analysis, here's a list of them:

1.
   ```
   apv_common_mode_info{
             uint_32 max_channel_num;
             double peak_adc_value;
             double common_mode;
   };
   ```

2.
   ```
   enum Axis{"x","y"}        ;
   ```

3.
   ```
   struct gemRawData{
             uint_32 event_id;
             uint_32 num_apvs;
             uint_32 apv_header[NUM_GEMS * NUM_APV_PER_GEM]
             uint_32 apv_id;
             int apv_data[NUM_GEMS][NUM_MAX_CHAN_PER_APV]
   };
   ```

4.
   ```
   struct gemCluster{
             uint_32 event_id;
             int x_chan_num;
             int y_chan_num;
             double adc_xy_avg_weight;
   };
   ```

There are a number of histograms of interest in this analysis, here's a list of them:

1. gem_hitmap
2. gem_correlation

3. adc_spectrum_no_cmode_subtracted
4. adc_spectrum_cmode_subtracted
5. adc_spectrum_no_ped_subtracted
6. adc_spectrum_ped_subtracted

6 Classes and their Methods

6.1 gem

The gem class is declared in gem.h and defined in gem.cxx. It abstracts the GEM detectors in our experiment. Each gem has an associated set of apv items which read data from the detector.

Here's the class elements:
Note that the hitmaps being migrated to the gems means they get set in define histograms by iterating through the gems but enforces logical separation.

```
int  id;
const char* name;
std::vector<apv* > apv_list;
dH1 hitmap;
```

And the class methods:

1. addApv(axis)

   ```
   check apv
           apv_list.pushback(apv)
   ```

2. combineApvAxisData(axis)

   ```
   For each axis_apv on gem:
           For each channel on apv:
                   gem_axis_channel_data.pushback(apv_data_this_channel)
                   return gem_axis_channel_data
   ```

3. findPedestal(data)
   I dunno how good this is: looks for "local maxima" but how is that range defined? I wanna see rms values for resolution/ cluster size from building the gems.

   ```
   findAllPeaks()
           sum channels != peaks and neighbors
           total/channels counted is pedestal
   ```

4. subtractPedestals(data, pedestal)

   ```
   For each data channel
           subtract pedestal from bin value
   ```

5. findClusters()
   returns std::vector of clusters guarunteed to be in descending order of weight

```
For each "peak" in x:
        For each "peak in y:
                weight cluster by xy ADC sums
                take NUM_MAX_CLUSTER_CANDIDATES
                return vector of them as gemClusters
```

6. drawClusterToHitmap(cluster)

```
Take first cluster:
        draw x,y data to hitmap for this gem
```

7. drawClustersToHitmap(vector of clusters)
   for debugging/ interest

```
For each cluster
        drawClusterToHitmap()
```

6.2 apv

The apv class is declared in apv.h and defined in apv.cxx. It abstracts the APV-25 chips which digitize and read strip data from the GEM detectors. Each APV needs to be calibrated for common mode gain subtraction. In addition, it's unclear to me if this is necessary, but it seems there used to be a form of channel to channel gain calibration. Ie scaling each channel's data by some factor order 1 which is predetermined and seems out of date.

Here's the class elements:

```
bool data_loaded;
bool gain_calib_loaded;
int id;
int num_channels;
const char* name;
apv_axis axis;
std::vector<double> apv_channel_data;
std::vector<double> channel_gain_factors;
std::bitset<NUM_MAX_CHAN_PER_APV> validChannelSet:
apv_common_mode_info cmode_info;
```

And the class methods:

8. clearApvData()

```
apv_channel_data.clear()
```

2. loadApvData()

```
check this chan data
apv_channel_data.pushback(this chan data)
```

3. determineCommonModeGain()

   ```
   For all channels on this apv:
           if this data isnt a max:
                   total_adc += this_apv_chan_data
   this apv cmode info = total_adc/strips_counted
   ```

4. subtractCommonModeGain()

   ```
   For each channel:
           subtract common mode from channel
   ```

5. parsePerChanGain()
   Not implemented in current codebase, but seems like an alternative to common mode subtraction? The original idea was to look at the the average maximum adc value for the gems as a way of calibrating the channels. Not implemented but an interesting idea. Now we need a good metric to evaluate gain factors per channel

   ```
   From gain calib vals:
           channel_gain_factors.pushback(this chan gain)
   ```

6. scaleDataPerChanGain()

   ```
   check data loaded
   check gain_calib loaded
   For each channel:
           scale chan data by gain factor
   ```

6.3 gemControl

The gemControl class is declared in gemControl.h and defined in gemControl.cxx. It processes and manages data from all the gems and holds definitions for "global" GEM analysis methods like defineHistograms and process which will be called by most GEM recipes.

Here's the class elements:

```
gem_list std::vector<gem*>;
```

And class methods:

1. defineHistograms():
   The defineHistograms function sets up output histos for the analysis, it is called before looping over data.

   ```
   for each 1d histo:
           add to 1d histo list
   for each 2d histo:
           etc.
   for each gem:
           gem->hitmap = new dH1()
   ```

2. startup():
   looks simple but it won't be, this function is gonna be a pain in the butt.

```
For NUM_GEMS:
        make gem
        configure gem
        configure all apvs on gem (part of above)
```

3. process(): The process function is called for every event in the data.

```
for each gem:
        for each apv:
                clear apv data
                load apv data for this event
                validate apv data
                calibrate data
        clusters = findClusters()
        drawClustersToHitmap( clusters )
```

4. readDaqRawEventData()

```
for each gem:
        for each apv on gem:
                validate/order data from daq_tree
                populate apv data from daq_tree:MMT
```

5. initInTrees()

6. initOutTrees()

6.4 binaryBlobReader

Here are the class elements:

```
gemRawData raw_data;
```

Here are the class methods:

1. readBank

```
iterate over data
        set raw_data event_id
        set raw_data apv_id
        set raw_data num_apvs
        fill raw_data apv_header
```

6.5 histogramList

The histogram list keeps track of all histograms for easy

6.6 Helpers

Sometimes helper functions are useful here are some:

- findAllPeaks(data)
  unfortunately finding all peaks is always O(n)

  ```
  for all channels
          if chan bigger than both neighbors
                  peaklist.pushback(this channel)
  ```

- findMax(data)
  unfortunately finding Max is O(n)

  ```
  for all channels
          if this is the biggest one keep track
  return biggest one
  ```

7 Major changes

Here's major structural or algorithmic changes:

1. removing all depricated references to telescopes and L/R which is a relic from OLYMPUS
2. moved all init steps to startup
3. removing all repeated, unnecessary loops over data
4.

8 Schedule

Week:
Oct 8 - 14: Writing gem and apv classes, Ryan unit tests them
Oct 15 - 21: Write gemControl and load real data functional code by 2 weeks
Oct 22 - 28: HAWAII
Oct 29 - Nov 5: findClusters studies, common mode studies
Nov 6 -12: unit testing
Nov 13 - 20: unit testing and complete