AIDA Data Analysis Code

Oscar Hall *
University of Edinburgh
AIDASort — Version 2

Abstract

A quick start look at using the current package of AIDA code developed at the University of Edinburgh for the analysis of AIDA raw data.

The guide is split into two sections the first being **AIDASort** which deals with the sorting of raw AIDA data. The second is **AIDA-Analysis** which encompasses a group of root scripts designed to help in the analysis of the sorted code.

Contents

* Oscar.Hall@ed.ac.uk

1 AIDASort

The AIDASort repository can be found at: https://github.com/ohall1/AIDASort

1.1 Building the Repository

Once you have cloned the repository from github you will need to build the program. The program is built with CMAKE. I recommend making a directory called build within the repository and compiling and making from within there .

```
mkdir build
build
cmake ..
make
```

This will generate the file AIDASort which can be executed.

1.2 Configuring the Program

The program can be somewhat tailored to specific use cases with the addition of pre-compiler definitions and if statements. If you wish to change these you will need to remake the program afterwards.

1.2.1 Output Type

One of the main options is the declaration of the output type of the root file. If no output type is defined an output tree will not be generated.

The program is currently set to output with x and y set to be the geometric centre of a cluster area and the height and width of the cluster is stored in the **TFast** branch, this is covered in greater detail in section 1.5. The definition for this output type is:

```
#define MERGER_OUTPUT
```

I would not recommend changing this option as the other Merer options are for testing purposes only.

1.2.2 Histogramming

A second useful definition is:

```
#define HISTOGRAMMING
```

With this definition a number of histograms will be created and stored in the output file defined. These can be useful when trying to get a quick picture of how the data looks.

1.2.3 Offsets

The final definition that may be of use to users is:

```
#define OFFSETS
```

With this defined when a low energy pulser walkthrough dataset is run a histogram will be produced which can be analysed using offsetPeaks.cpp in the AIDA-Analysis repository to calculate offsets for the channels. **Do not have this defined when trying to analyse data other than pulser walkthroughs as it removes the data output of the program.**

1.3 Running the Program

Once the program has been built it is relatively straight forward to analyse data using it. The program can be run using the command:

```
./AIDASort -c PathToConfiFile/ConfigFile.csv -o
    PathToOutputFile/OutputFile.root
```

Where -c specifies the following argument will be the configuration file and -o specifies the following argument will be the output file. The program will either open a new file or overwrite an existing file with the same name.

1.3.1 Configuration File

The configuration file contains most of the options needed to run the program. In here the raw data files to be analysed are specified and the parameters file which stores the offsets and FEE layout is selected. An example file is included in the repository and lists the configuration options.

If wanting to sort a single file then the AIDAFile option can be used followed by the path to the file and the file itself e.g.

```
AIDAFile            /scratch/data/R6_237
```

If instead multiple sequential files want to be sorted the option AIDAList can be used. Where the format is "PathToFile/String before run number" <run number> <sub-run start> <sub-run end> e.g.

```
AIDAList /scratch/data/R 6 17 20
```

Which will sort R6_17, R6_18, R6_19 and R6_20 in the directory /scratch/data.

The final option specifies the parameters file to be used. Unless you are changing the offsets used by the program this option can likely be set to point to the ExampleParameters.csv file within the repository. If you do wish to change it the format is:

```
AIDAConfig PathToParameterFile/ParameterFile.csv
```

1.3.2 Parameters File

The parameters file lists all the key information needed for the program to run. Much of this will not change from experiment to experiment but certain aspects will. The first option is **dssdMap**; this option will map FEEs to their respective DSSDs. This is followed by **sideMap** which defines whether a FEE is dealing with a front or back side of the detector. This is followed by **stripMap**, this is based upon which part of the DSSD the FEE is responsible for. **adcPolarity** this again relates to the side of the DSSD which the FEE deals with. These options will be given to you if they are different to what they are as default in the example parameter file.

The last group of options are the **adcOffset**. These will vary experiment to experiment and define the offset which is applied to low energy ADC signals. These are typically calculated from a pulser run. The structure of the offsets in file is:

```
adcOffset       1       0       168.755  0.999946
```

Here **adcOffset** is the parameter name which tells the program what to expect, **1** is the FEE, **0** is the channelID, **168.755** is the offset for the channel and **0.999946** is a number which describes the quality of the fit generated by the offsetPeaks.cpp file.

1.4 Structure of the Program

Here I will just add a couple of notes on the structure of the code to give a general idea of what happens where.

1.4.1 Threads

The program is written as a multi-threaded program. In it's current iteration it runs on three threads with thread safe buffers in operation between each thread. The first thread is responsible for reading in the raw data and creating the two 32bit data words from this. The second thread is responsible for unpacking the data words and creating adc data items from them and then building the event from this. The final thread is responsible for calibrating the raw data (converting from FEE and channel to DSSD and strip and converting the raw adc data into keV) and then creating the clusters and matching them.

1.4.2 DataReader.cpp

This thread is responsible for reading in the raw data from the AIDA raw files as blocks. These blocks are 64kB in size. The blocks are then processed to split them into pairs of the two 32 bit words which make up the AIDA data items which are then stored in lists as pairs until the full block has been processed.

1.4.3 DataUnpacker.cpp and EventBuilder.cpp

This thread takes the list of data item pairs and processes them. DataUnpacker.cpp is responsible for taking the two 32 bit words and separating this into the item information such as FEE, Channel and ADC Value for ADC data items. These value are stored in C++ data objects. Throughout the analysis different data objects are used depending on the stage of the sorting and the type of information that is being stored in them. All the objects used in the analysis are defined in DataItems.cpp. Once the two data words have been unpacked and a data item unpacked if it is an ADC data item it is then passed onto EventBuilder.cpp.

As AIDA is a triggerless DAQ all data items come in with their own timestamps. It is therefor necessary to group the incoming data stream into events which are formed by groups of data items falling within a time window. Due to the effect of the multiplexer on ADC signals a fixed event window is not used, instead we search for gap in the data of greater than $2\mu s$ or 200 AIDA clock cycles or 2000 WR timestamp units and call all events that are in a chain with spacings of less than this time one event. Once an event has been found it is split based upon it's ADC range; if there is a high energy ADC data item present indicating an implant all low energy event's are discarded, likewise if the multiplicity is such that over half of the low energy system are triggered it is defined to be a pulser event and it is added to the pulser histograms, and if

neither of these conditions are met it is treated as a low energy decay event. This is done in EventBuilder::CloseEvent() beginning on line 71 of EventBuilder.cpp. Once an event is closed it is stored as a list of all the data items that are involved which is placed in a thread safe buffer.

In the EventBuilder.cpp file there are two function calls which are optional depending on your use of the program (**Note:** If you plan to merge the data using the BRIKEN merger these functions are required). The first function is CorrectMultiplexer. It is called on line 142:

```
CorrectMultiplexer(adcItem);
```

This function corrects the timestamp for the output of the multiplexer.

The second function is ApplyCorrelationScalerOffset which is called on line 144

```
ApplyCorrelationScalerOffset(adcItem);
```

This function syncs the AIDA timestamp with those that are used by BRIKEN and BigRIPS so correlations can be made between the three data acquisition systems.

1.4.4 Calibrator.cpp and EventClustering.cpp

This is the final thread of the program. Calibrator.cpp takes the list of data items that make up an event and calibrate the data items. It is here that we switch from using FEE and channel and move to DSSD, side of the detector and strip based on parameters which are stored in ExampleParameters.csv. With data items in terms of these values we can create our clusters.

A key thing of note here is that all we use in calibrating the strips is applying an offset to the adc value. This offset is calculated by carrying out a pulser walkthrough.

We use clusters as implant and decay events are rarely single pixel events and as such we need to take into account all active pixels in our events. A cluster is formed of adjacent strips that all fire within an event window. The energy of the cluster is the sum energy of all the strips. The cluster has two timestamps an upper and lower relating to the highest and lowest data item timestamps that make up the cluster. Similarly a cluster has two positions an upper and a lower relating to the highest and lowest strips involved. A timing condition can also be applied in cluster formation that all strips must be within a set time of each other, currently the program is configured for $4\mu s$. Up to this stage implants and decays are treated the same, however here there is an extra stage for implants.

When analysing the data we care most about the layer that the implant particle stops in the DSSD stack and don't need information on the track it takes. In EventClustering.cpp there is therefore a function called EventClustering::ImplantStoppingLayer(). This function uses a list of conditions to determine an implants stopping layer and determines whether an implant event can be defined. The conditions currently in use are

- Must be no high energy clusters downstream of the stopping layer
- Must be a cluster in at least one side of every upstream detector

- Must be a cluster in both sides of the stopping layer detector

If these conditions are not met the event is deemed to not be an implant and is discarded as correlations would not be able to be carried out.

With stopping layers for the implant calculated the final stage is to match front and back clusters. This is done with an equal energy cut. Currently in use x and y cluster must be within 150keV of each other for decays and 300MeV for implants. An additional condition can be added that clusters must also fall within a time window of each other, in this case it is 4000ns. When clusters are paired the timestamp is taken to be the lowest timestamp in either cluster. x and y are defined to be the geometric centre of the cluster and dx and dy are stored as the difference in x and y between the upper and lower values for each. With clusters paired they are added to a final list which once the full event window has been processed is written to file.

1.5 Final Output

The final output of the sort program is a ROOT file containing a single tree object. The branches of the tree are set out as such:

- T - 64bit int Timestamp of the data events in ns
- TFast - 64bit int bottom 16bits used to store dx and dy
- E - 64bit double The average of the x and y cluster energies
- Ex - 64bit double The energy of the x side cluster
- Ey - 64bit double The energy of the y side cluster
- x - 64bit double The geometric x centre of the cluster
- y - 64bit double The geometric y centre of the cluster
- z - 64bit double The stopping layer of the detector (0 Ordered)
- nx - Int The number of x strips in the cluster
- ny - Int the number of y strips in the cluster
- nz - Int A pointless variable currently required by the merger
- ID - Int Declares whether an event is an implant (4) or beta (5) to the merger

2 AIDA-Analysis

For RIKEN based experiments the files produced by AIDASort are passed onto a merger in which all the correlations are calculated. In this section I will break down how implants and decays are correlated.

2.1 Correlations

Correlations are calculated using the positional information of implantation and decay events. As each implant or decay item has an area specified to it we look for overlapping or adjacent areas in the implant and decay items to ascertain correlations. The information is stored in the form of x, y, dx and dy where x and y are defined to be the geometric centres of the cluster and dx and dy are calculated by taking the difference in the highest and lowest strips of the cluster. To check if there is overlap one can therefore do:

```
if((imp.Y + (idy/2.0) >= (beta.Y-((dy/2.0)+B)) &&
        (imp.Y - (idy/2.0) <= (beta.Y+((dy/2.0)+B)){
if((imp.X + (idx/2.0) >= (beta.X-((dx/2.0)+B)) &&
        (imp.X - (idx/2.0) <= (beta.X+((dx/2.0)+B)){
```

Where B is an optional border variable that lets the use choose between searching for a strict overlap in area or if adjacent areas are correlated. For example with $B = 0$ the correlation condition is restricted such that a beta must overlap an implant whereas with $B = 1$ a beta now only needs to be touching an implant.

In terms of timescales it is important that you correlate implants with betas forwards and backwards in time. This is done as the backwards time correlation gives a measure of the random noise that is associated with a give area in the detector, See Figure 1.

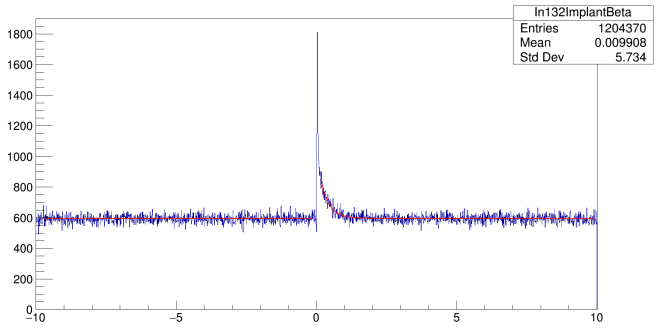

Figure 1: Fit of ^{132}In using the solution to the Bateman equations as the fit function. $x-$axis is time in seconds.