NPS Software Updates

NPS Collaboration Meeting

Feb 02, 2021

Carlos Yero
Steve Wood

Jefferson Lab
Exploring the Nature of Matter

NPS

Hall C

What has been done?
(Part I: Create NPS software)

- NPS Software is currently under development (e.g., we have a GitHub repository, `NPSApp`, for version control)

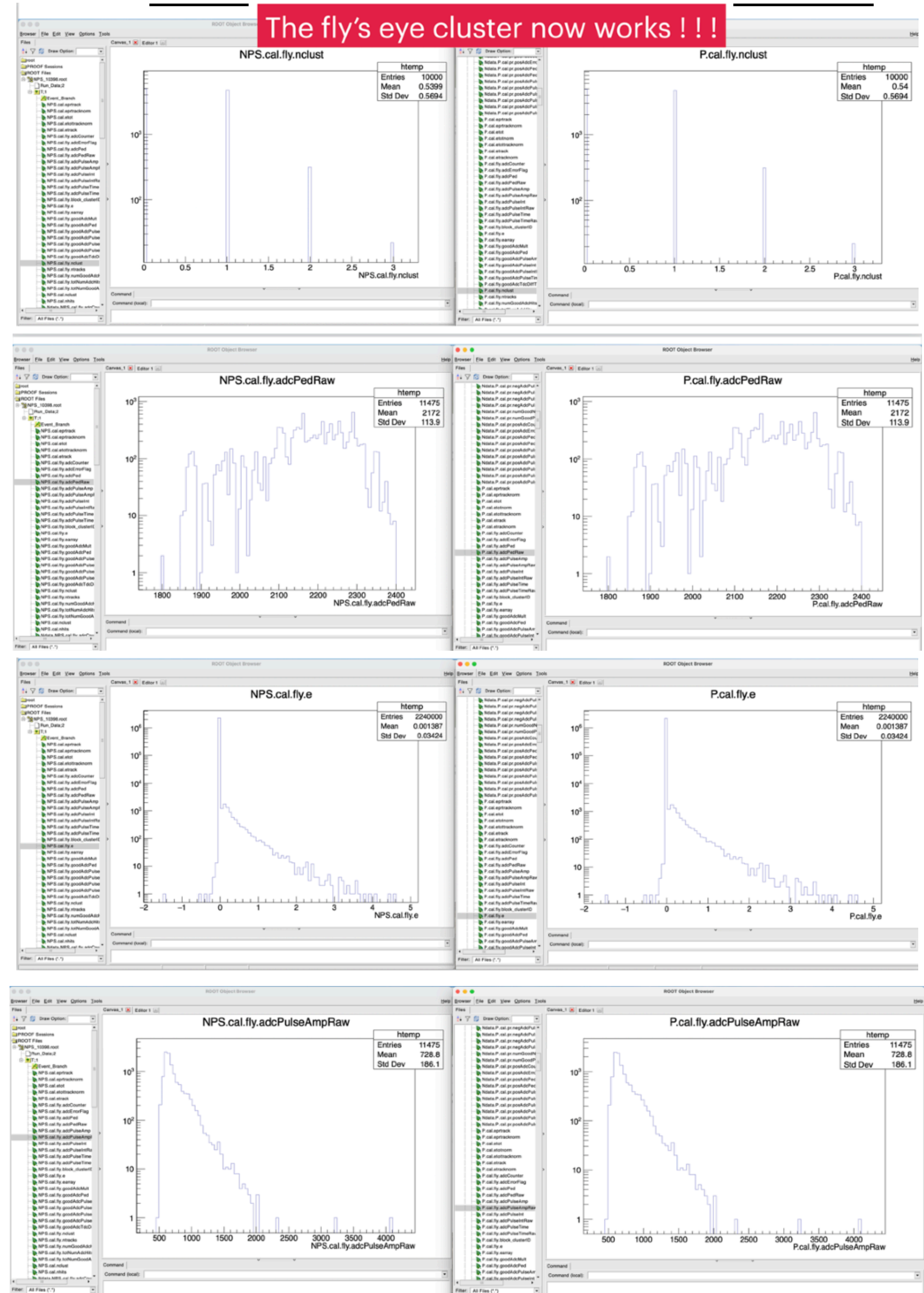
- The NPS C++ classes that have been adapted from `HCANA` to the `NPSApp` development repository are:

  e.g., `THcNPSAnalyzer`, `THcNPSApparatus`, `THcNPSCalorimeter`, `THcNPSArray`, `THcNPSShowerHit`

- Since NPS does not have tracking detector, tracking information has been removed and the NPS code has been tested under these conditions using actual SHMS data.

  e.g., these tests show agreement between NPS and SHMS software codes for variables which do NOT depend on tracking

- The NPS replay script has been adapted from that in the `hallc_replay` repository. The parameters, maps and database structure have also been adapted from `hallc_replay` to `NPSApp` and have been re-named accordingly.



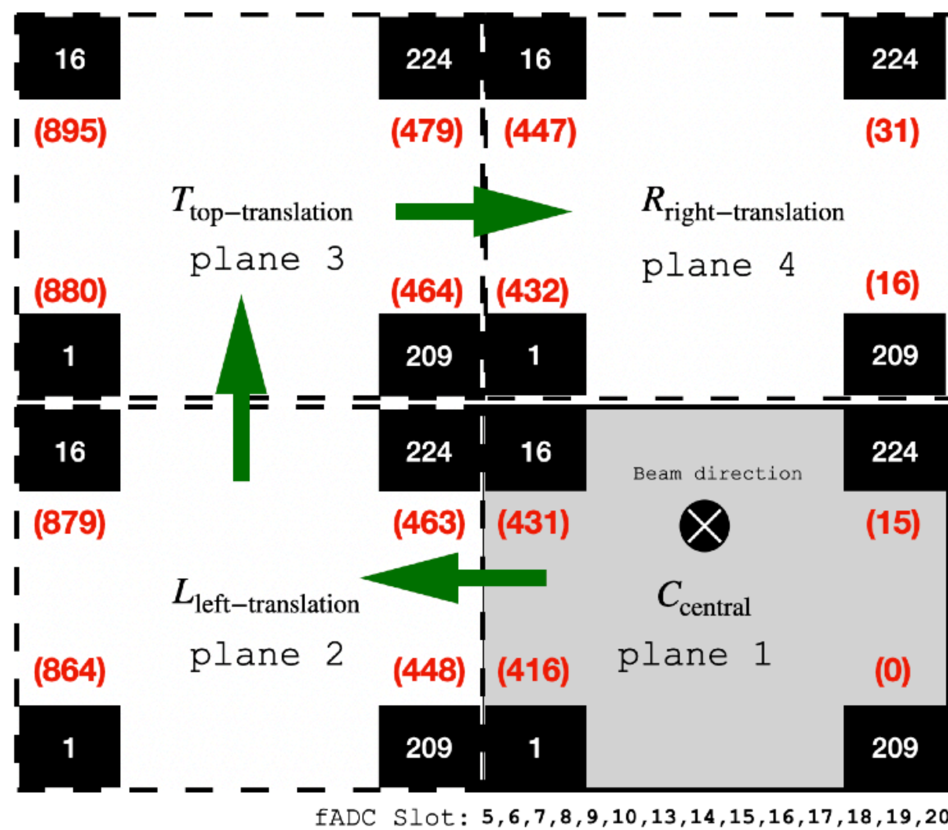
What has been done? (Part II: Testing the NPS Software)

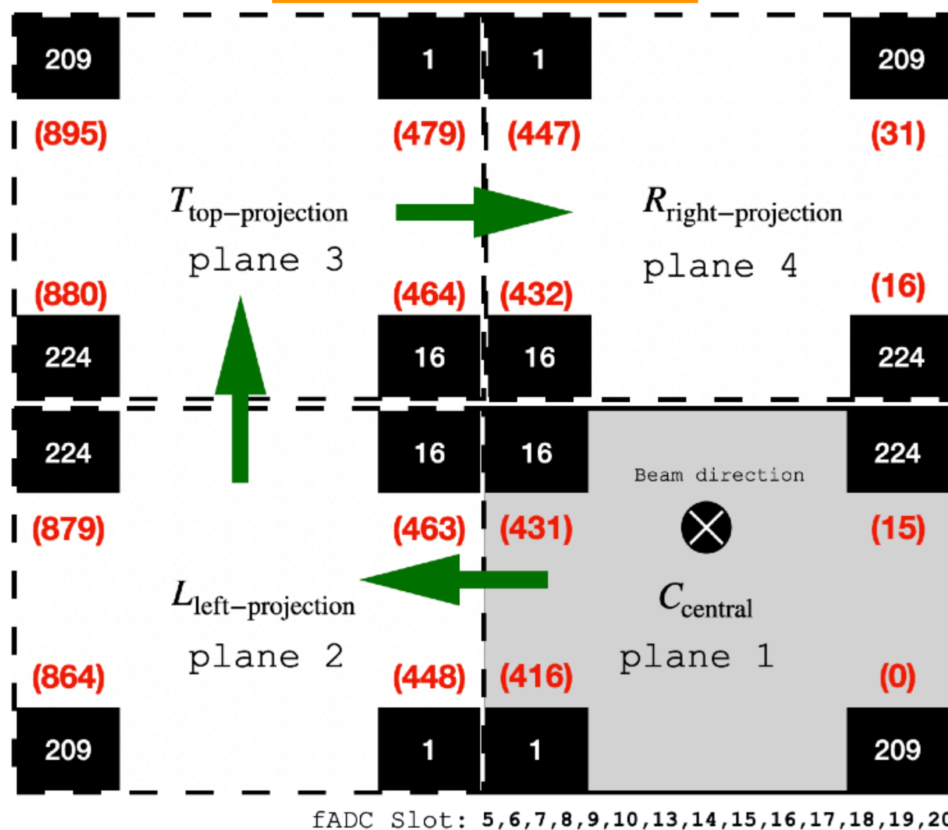
```
Merging events
--------------

Higher multiplicities can be simulated by merging several SHMS events.
To do this, use the THcNPSAnalyzer::SetNevMerge method to set the number
of events that should be merged.

```
THcNPSAnalyzer* analyzer = new THcNPSAnalyzer;
analyzer->SetNevMerge(4);
```
```

```
Simulating a larger calorimeter
-------------------------------

The SHMS fly's eye calorimter has 16 rows and 14 columns for a total of
224 calorimeter blocks.  The NPS calorimter as 36 rows and 30 columns
for a total of 1080 blocks.  In order to use SHMS data to simulate a
detector with a number of blocks similar to the the NPS, we can double
the number of rows and columns, making a fake detector with four
quadrants of 16x14 blocks.

To do this we set the row and column count parameters to 32 and 28.
We can then place data into these fake quadrants in two different
ways.

1.  One way is to use the map file pcal_nps_projection.map or
pcal_nps_translation.map.  These map files map each channel into 4
locations in the fake detector.  Then an event with a single cluster
will appear to have four shower clusters.

2.  The second way is to set the parameter nps_cal_replicate to one or
two.  Then for each event, one of the four quadrants will be chosen
randomly and all the hits will be moved to that quadrant.  If this is
used with the event merging described above, multipicites greater than
one can be simulated.
```

 1. Mimic NPS Channel Density Using External Map File

Legend: SHMS block (NPS block)





- *"Translate"* or *"Reflect"* SHMS "fly's eye" calorimeter on four planes to mimic NPS block density e.g., the corresponding map has already been developed

- The calorimeter mapping tests have served as additional checks/tests of the NPS software currently being developed

- **How does it work (What was done)?**

1. Modified the geometry and signal map files to include the 3 "fake" calorimeter planes in addition to the existing real plane.

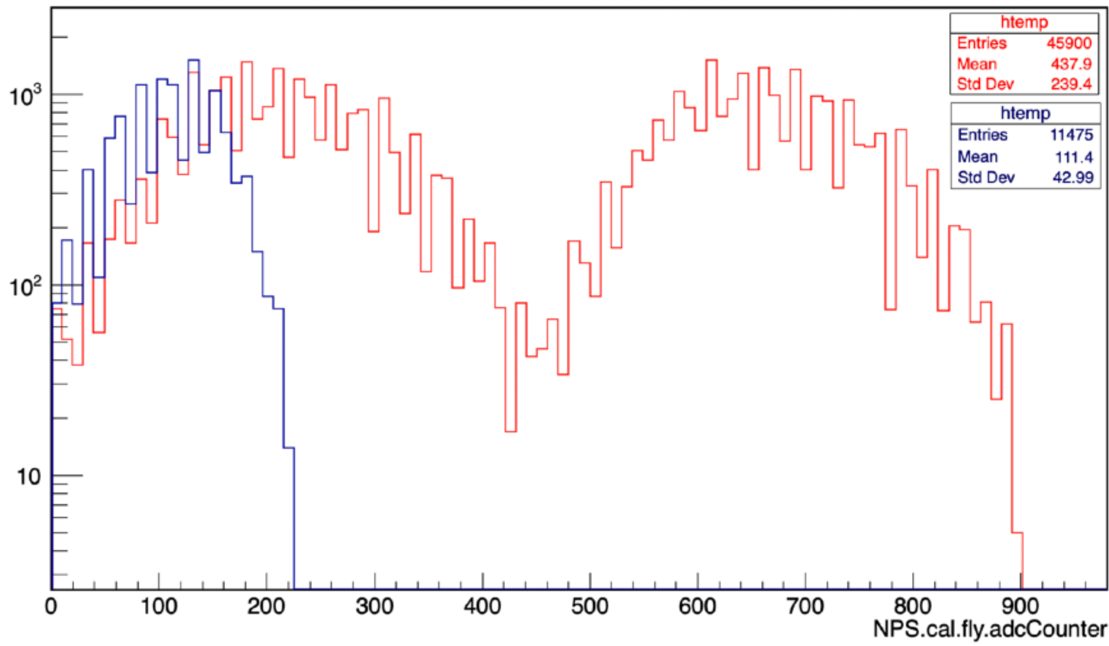
2. The calibration/cuts parameter files from existing SHMS calorimeter data were copied over (x3) times to create 896 parameters for the "fake" NPS

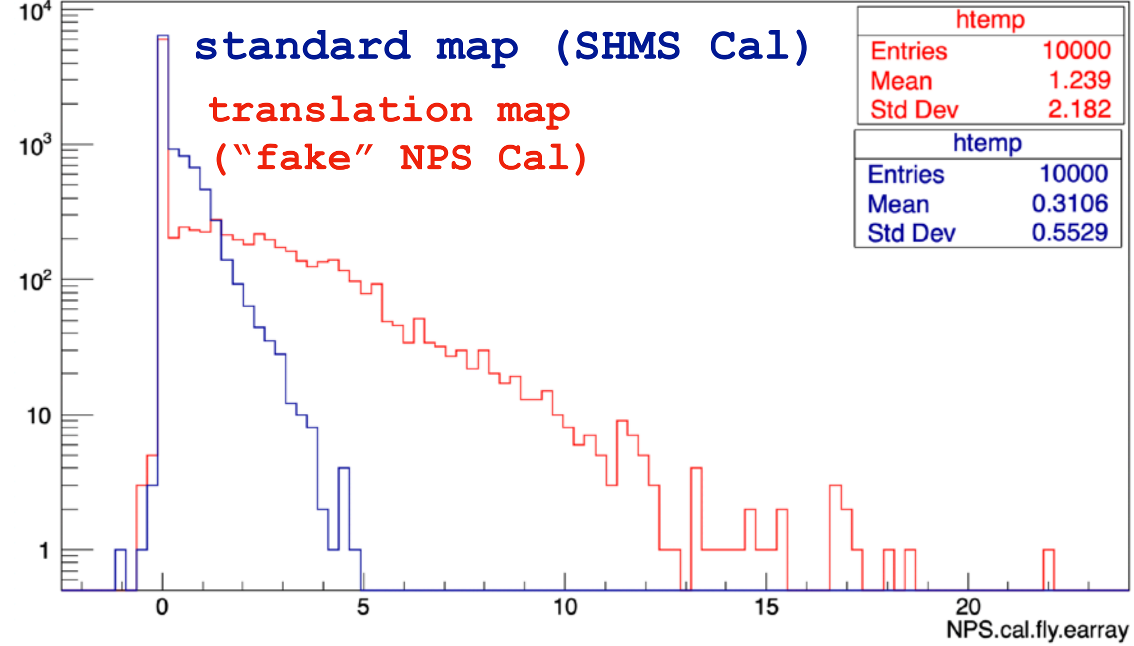
e.g., This was done only for the translation case, as it is the one we have looked at so far using the map files

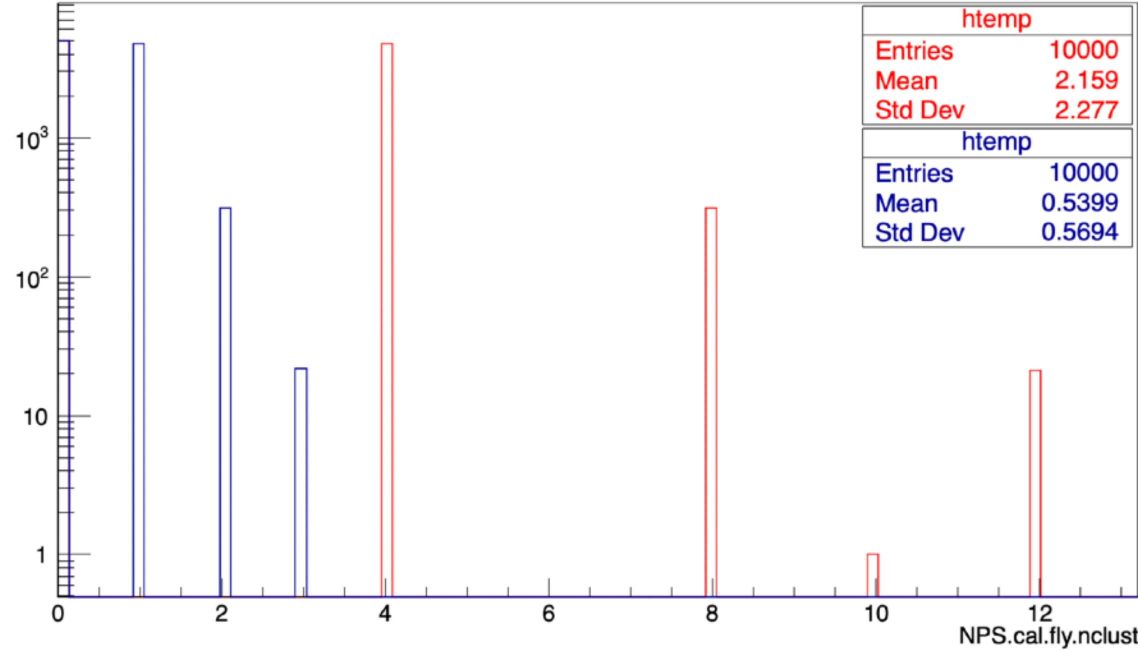
3. During the NPS replay (using SHMS data) the signals from the real calorimeter plane are either *translated* or *reflected* onto the 3 remaining "fake" planes, effectively generating 3 additional shower clusters per each cluster formed in the real plane.

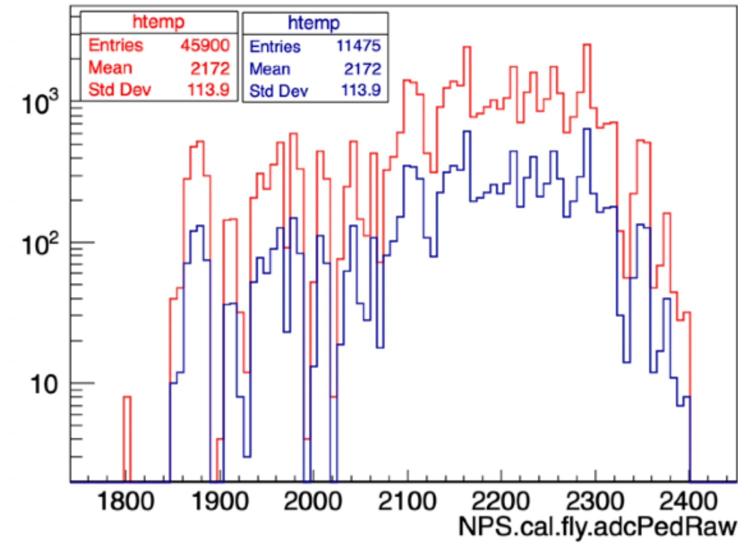
e.g., additional relevant variables, such as the total shower energy deposited or fADC variables will appear to be made effectively larger as compared to the original SHMS data, since the hits have been quadrupled

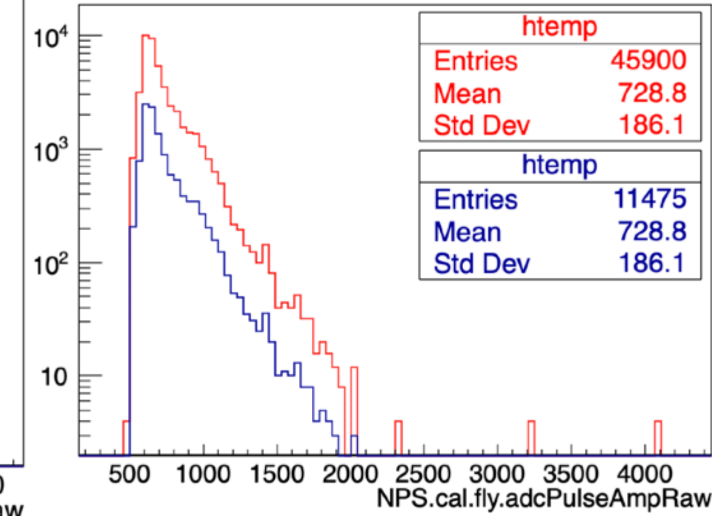
Part II: Testing the NPS Software| 1. Mimic NPS Channel Density Using a Map File

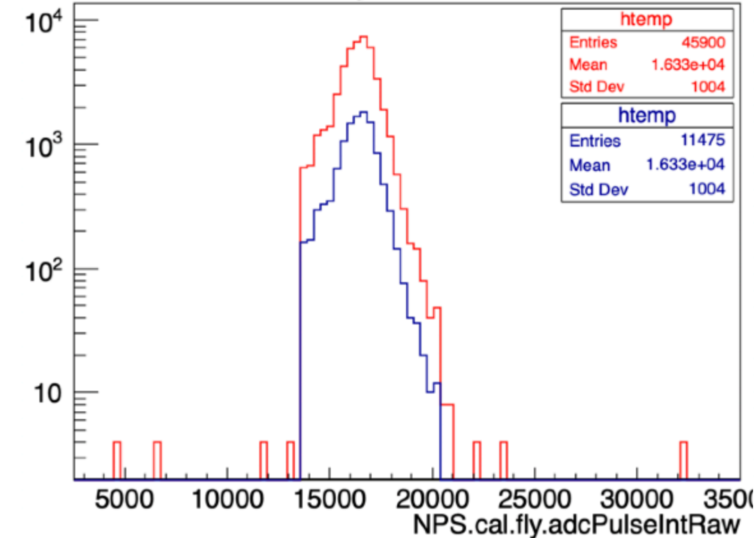


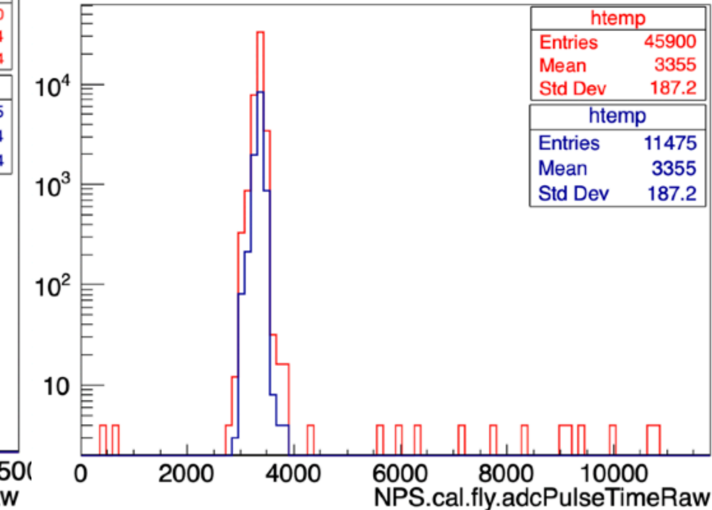


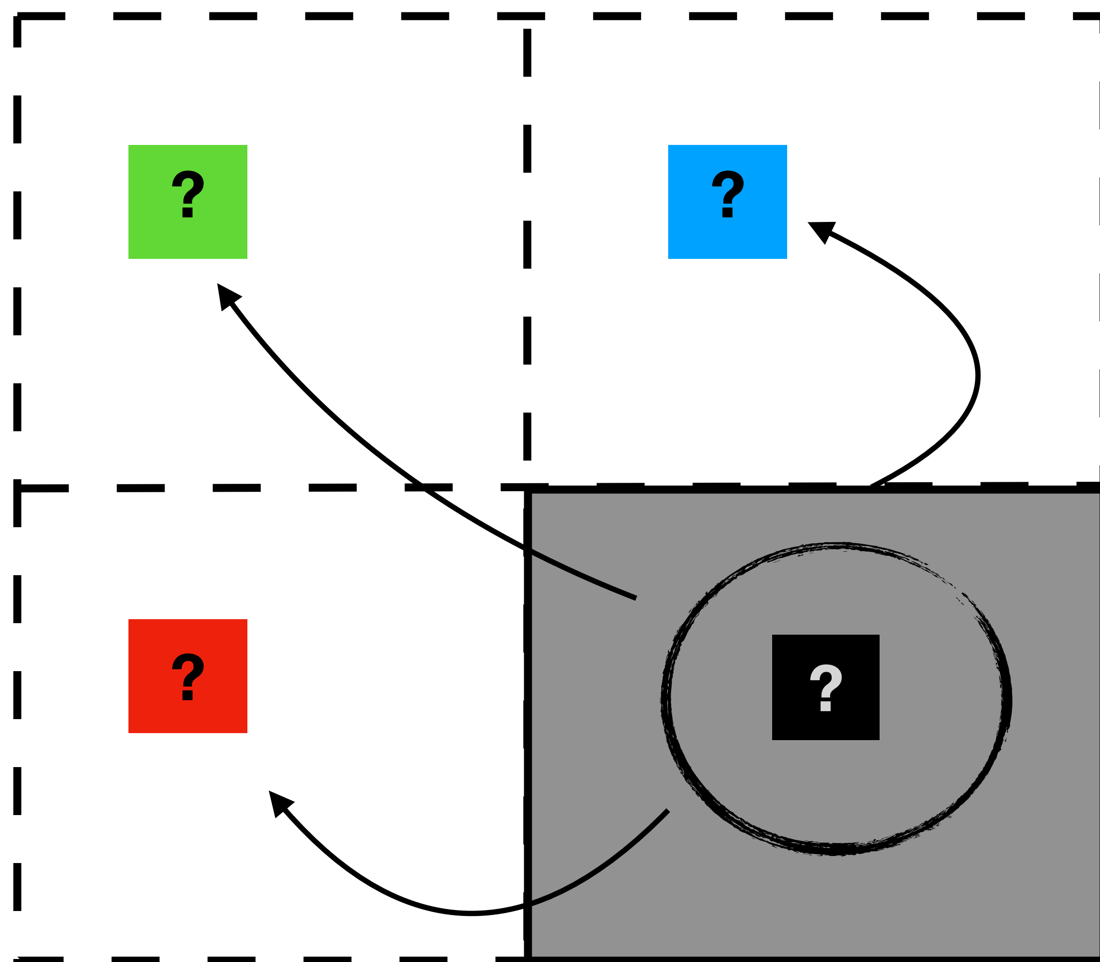












- S. Wood developed an alternative method to populate our "fake" NPS calorimeter more realistically, using the newly added `nps_cal_replictae` parameter

- **How does it work ?**

  1. For every trigger (physics event) detected on the real calorimeter plane, select one of the four quadrants at **random** and move the event to that quadrant either by *"translation"* (`nps_cal_replicate=1`) or *"reflection"* (`nps_cal_replicate=2`) .

  2. Since the signal hits are moved to one of these quadrants, then the map file requires ONLY the signal map from the standard SHMS calorimeter, since the mapping to other quadrants is done internally (in `NPSApp` source code)

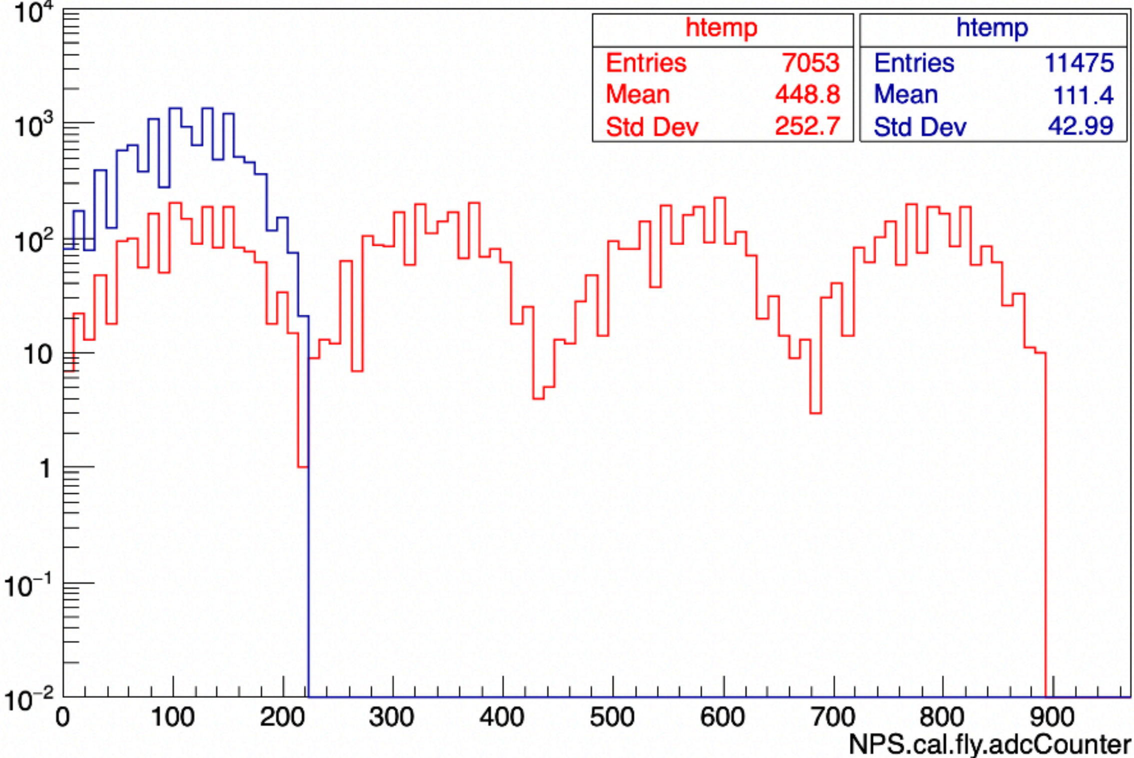
  3. In terms of the calibration/cuts param files, we still need the 896 elements, as these will be used depending on which quadrant the event lands in.

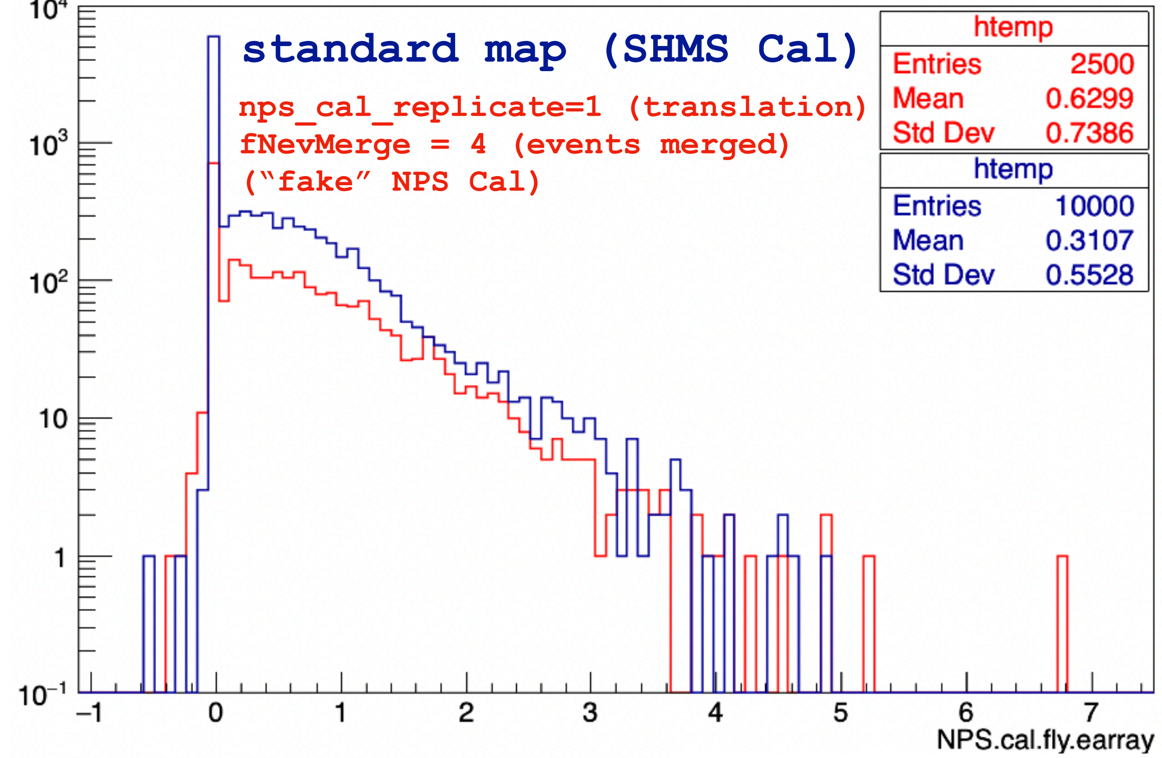
- **Event Merging Option**

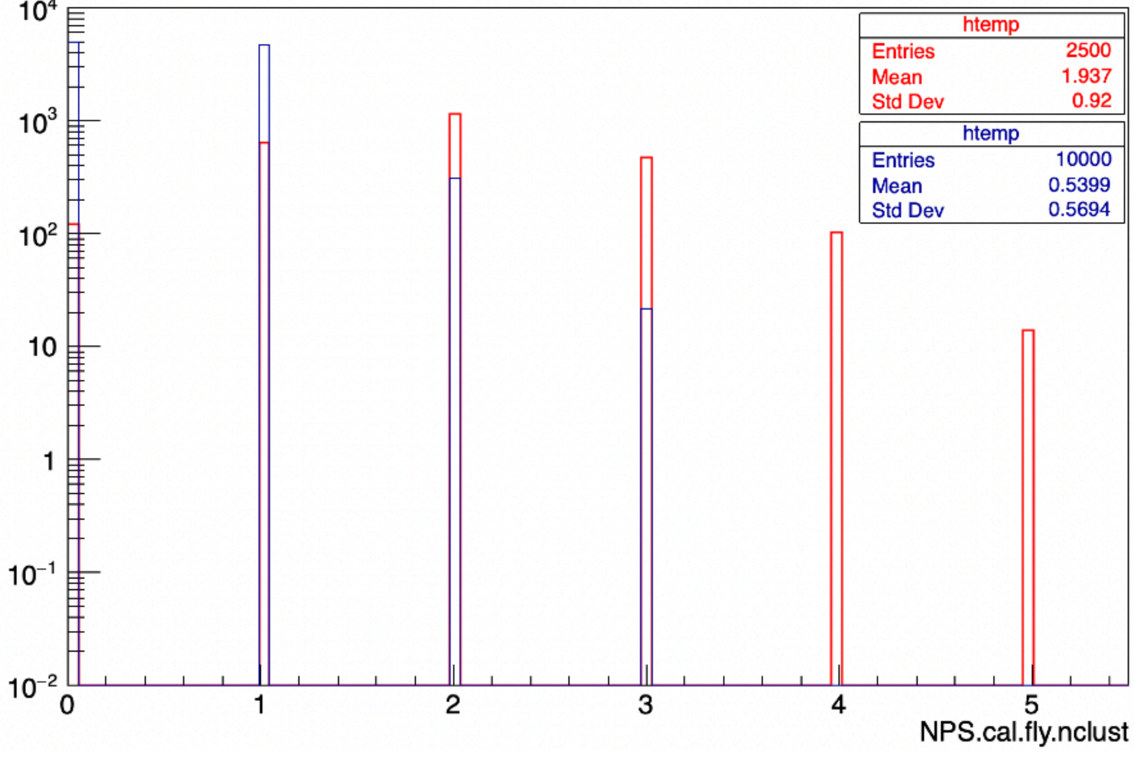
In addition to randomly moving an event to a quadrant, one can also "merge" hits every *N* events into the hit list. When the *Nth* event is reached, then all of the hits accumulated from the previous events (including *N*) are analyzed as if they were from the *Nth* event, the hit list is then cleared before analysis of the next *N* events.

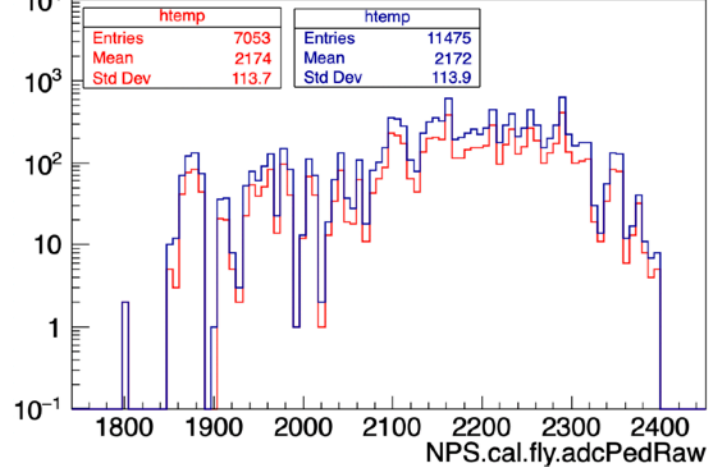
This option is used for simulating higher rates (i.e., multiplicities), and can be used concurrently with the `nps_cal_replicate` option described above to generate higher multiplicities on the "fake" NPS calorimeter.

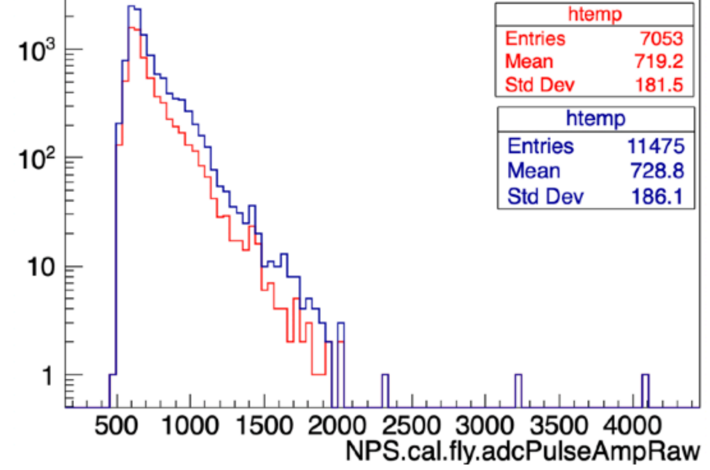
Part II: Testing the NPS Software| 2. Mimic NPS Channel Density Using "nps_cal_replicate" Option

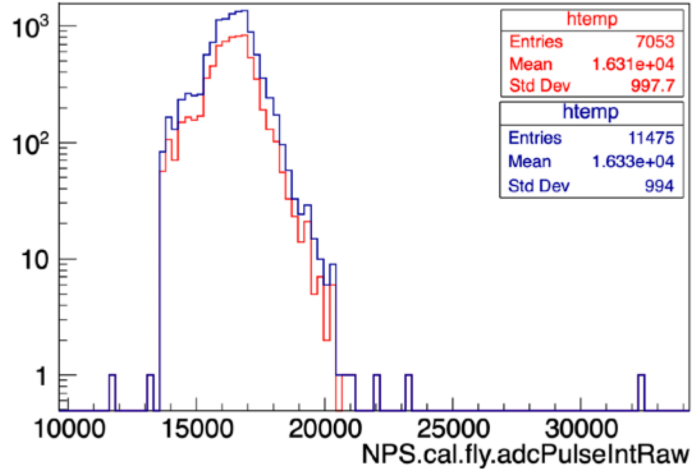


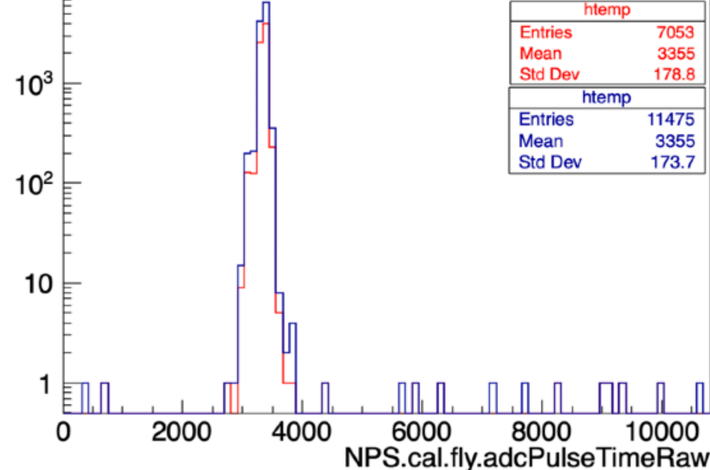












NPS Calorimeter Clustering Algorithm: Cellular Automata Method

Definition: A cellular automaton is an array of simple individual processing cells which evolve iteratively according to a fixed set of instructions or rules.

A cellular automaton evolves iteratively : at each step, each cell examines its inputs, decides on the basis of a transition rule whether or not to change its state, and sends its new output value to the inputs of its neighbours. At the next step, these new inputs are examined and the cells evolve simultaneously.

- The principle of the calorimeter clustering algorithm using the cellular automata approach is described in Ref. https://doi.org/10.1016/0168-9002(95)00217-0

- The cellular automata approach was implemented in the Hall A DVCS Calorimeter some time ago (2006)

- Hall C NPS calorimeter will follow the Hall A DVCS calorimeter clustering algorithm, as there is already experience using the cellular automata approach.

- We plan to use time information as well as x-y coordinates of blocks) in cluster identification. (3D clustering)

For each trigger, the energy deposit in each block will be used as the cell initial state. Only hit cells can evolve. The main rule of the cellular automaton evolution is that a cell at a given step will take the value of its highest energy neighbour. This can easily be understood by drawing a parallel to biology. For a given trigger, several particles, or at least the one that fired the trigger, have developed a shower. The block with the highest energy in these showers is oing to act like a virus contaminating the other blocks. Iteratively the contamination will spread over the calorimeter from cell to cell. When the process stabilizes the clusters in the calorimeter will be tagged by their respective virus.

NPS Calorimeter Clustering Algorithm: Cellular Automata Method

Rule–I A given cell is only sensitive to its eight neighbours. A cell is a virus if its value is higher than the value of each of its neighbours.

Rule–II At a given step, a cell will take the value of its highest energy neighbour.

Rule–III A cell already contamined in an earlier stage by a virus is immunized against any other virus (restriction to Rule II)

- "Cellular Automata" rules to follow when forming clusters of hits (See Ref. https://doi.org/10.1016/0168-9002(95)00217-0
- These rules may be flexible depending on the circumstances. For example, depending on the energy deposited in a cell of a highly segmented calorimeter, or depending on the location of a cell, one may need to re-define what constitutes neighboring cells
- * Need to determine how to treat cells with shared clusters. Do we apply "Rule-III" or determine how to share the energy between the two clusters later during the calorimeter reconstruction process?

cell (hit value ∝ ADC pulse amplitude)

cell(no hit)

0.3	0.6	0.2		0.6	
1.3	4.8	1.2	0.7	0.2	
	0.9		3.1	0.7	

step_0: Per physics event—identify all blocks with ADC amplitude > ADC_threshold. Only the hit blocks will form part of the clustering algorithm later on.

cell neighborhood boundary

cell (virus)

0.3	0.6	0.2		0.6	
1.3	4.8	1.2	0.7	0.2	
	0.9		3.1	0.7	

step_1: Identify the virus, i.e. cells with a local maxima or larger energy deposit than their neighbors.

cell (contaminated or "tagged")

etched cell* (shared by two clusters)

0.3	0.6	0.2		0.6	
1.3	4.8	1.2	0.7	1.2	
	0.9		3.1	0.7	

step_2: Virus cell contaminates nearest neighboring cells, which are "tagged" with the same color as the virus to form clusters. Cells with multiple tags (etched) are shared between their respective clusters, and a determination of how to handle these cases must be made. The contamination will spread iteratively over secondary, tertiary, etc. neighbors provided these have not been "tagged"

Cellular Automata Evolution

NPS Calorimeter Clustering Algorithm: Cellular Automata Method in DVCS

- Currently, we are studying the DVCS clustering algorithm code and plan to adapt it to the `NPSApp` source code
- I had some questions regarding the DVCS clustering algorithm code:

```
//________________________________________________________________________________
Int_t TCaloEvent::DoClustering(Double_t timemin, Double_t timemax, Float_t BlockThreshold)
{
```

```
//If a time window is used, blocks with no pulse inside it are excluded
if(timemin>-1000||timemax>-1000){
  if(block->GetEnergy(0)>0.&&(block->GetTime(0)<timemin||block->GetTime(0)>timemax)
     &&block->GetEnergy(1)>0.&&(block->GetTime(1)<timemin||block->GetTime(1)>timemax)) {
    blocksenergy[block->GetBlockNumber()]=-10;
  }
  if(!(block->GetEnergy(1)>0.) &&(block->GetTime(0)<timemin||block->GetTime(0)>timemax))
     blocksenergy[block->GetBlockNumber()]=-10;

  if(block->GetEnergy(0)>0.&&block->GetTime(0)>=timemin&&block->GetTime(0)<=timemax)
    blocksenergy[block->GetBlockNumber()]=block->GetEnergy(0);

  if(block->GetEnergy(1)>0.&&block->GetTime(1)>=timemin&&block->GetTime(1)<=timemax
     && (TMath::Abs( (block->GetTime(1) )-( (timemax+timemin)/2.) ) <
         TMath::Abs( (block->GetTime(0) )-( (timemax+timemin)/2.) ) )   )
    blocksenergy[block->GetBlockNumber()]=block->GetEnergy(1);

  if(block->GetEnergy(0)>0.&&block->GetTime(0)>=timemin&&block->GetTime(0)<=timemax
     &&block->GetEnergy(1)>0.&&block->GetTime(1)>=timemin&&block->GetTime(1)<=timemax) {
    returnval++;//If two pulses the energy is set to the closest in time to the middle of the window
  }
}
```

- What is the difference between the two *::DoClustering()* methods? (It just seems as if they both do not use the 'time')

```
//________________________________________________________________________________
Int_t TCaloEvent::DoClustering(Double_t timemin1, Double_t timemax1, Double_t timemin2, Double_t timemax2, Float_t BlockThreshold)
{
```

```
Double_t timemin=timemin1;
Double_t timemax=timemax1;
for(Int_t nwindow=0; nwindow<2; nwindow++){//Loop over clustering windows
  if(nwindow==1){
    timemin=timemin2;
    timemax=timemax2;
  }
  Int_t NbClusters=0;
  for(Int_t i=0;i<fNbBlocks;i++) {
    TCaloBlock* block = (TCaloBlock*)fCaloBlocks->UncheckedAt(i);
    bl[i]=block->GetBlockNumber();
    blinv[bl[i]]=i;
    blocksenergy[block->GetBlockNumber()]=block->GetBlockEnergy();

    //If a time window is used, blocks with no pulse inside it are excluded
    if(timemin>-1000||timemax>-1000){
      if(block->GetEnergy(0)>0.&&(block->GetTime(0)<timemin||block->GetTime(0)>timemax)
         &&block->GetEnergy(1)>0.&&(block->GetTime(1)<timemin||block->GetTime(1)>timemax)) {
        blocksenergy[block->GetBlockNumber()]=-10;
      }
      if(!(block->GetEnergy(1)>0.) &&(block->GetTime(0)<timemin||block->GetTime(0)>timemax))
        blocksenergy[block->GetBlockNumber()]=-10;

      if(block->GetEnergy(0)>0.&&block->GetTime(0)>=timemin&&block->GetTime(0)<=timemax)
        blocksenergy[block->GetBlockNumber()]=block->GetEnergy(0);

      if(block->GetEnergy(1)>0.&&block->GetTime(1)>=timemin&&block->GetTime(1)<=timemax
         && (TMath::Abs( (block->GetTime(1) )-( (timemax+timemin)/2.) ) <
             TMath::Abs( (block->GetTime(0) )-( (timemax+timemin)/2.) ) )   )
        blocksenergy[block->GetBlockNumber()]=block->GetEnergy(1);

      if(block->GetEnergy(0)>0.&&block->GetTime(0)>=timemin&&block->GetTime(0)<=timemax
         &&block->GetEnergy(1)>0.&&block->GetTime(1)>=timemin&&block->GetTime(1)<=timemax) {
        returnval++;//If two pulses the energy is set to the closest in time to the middle of the window
      }
    }
  }
```

- Im not clear about what is the functionality of 'nwindow' here, and how it plays a role in determining whether to assign a timemin/max 1 or 2 to the generic timemin/max variable. It also seems as if the adc pulse time was not really used here as well, since the min/max are set to >-1000

- Maybe I can have offline discussion with Carlos M. Camacho to learn more about this method.

Open Discussion On these Topics?

From Brad's talk yesterday:
https://wiki.jlab.org/cuawiki/images/a/ac/Sawatzky-DAQ_update_01Feb2021.pdf

- Analyzer (Hall C) mods (Steve?)
  - → *Integrate existing DVCS software into hcana*
  - → *Decoder updates for VTP, F250 mods*
  - → *Solve and Implement "Data Unblocking" issue*
  - → *Merge multi-threaded podd with hcana?*

We are currently working on this (The plan is to actually integrate DVCS cluster algorithm into hcana)

We can work on these if we get some information and example data files (S. Wood)

We hope someone else does this. (This might be done in hardware)

We don't think multi-threaded podd is ready yet, but we can work on it when it is ready.

Thanks !