FilterAlg Description and User's Guide

David Wren

dnwren@milkyway.gsfc.nasa.gov

Introduction

FilterAlg is a Gaudi algorithm that is distributed as part of the OnboardFilter package. It consists of 3 files: FilterAlg.cxx, FilterAlg.h, and FilterAlg_acd.h, which duplicate the logic of the flight software contained in OnboardFilter. The original author and maintainer of the onboard filtering code is JJ Russell, and any changes to his version of the flight software will drive corresponding updates to OnboardFilter and FilterAlg.

One reason for writing FilterAlg was to discover bugs in OnboardFilter, as FilterAlg's output should agree completely with that of OnboardFilter. The second was to provide a version of the filtering logic that is more compact and easier to understand and manipulate than the copy contained in OnboardFilter. Instead of having to go through tens of files and thousands of lines of optimized code, the user may work with 3 files and a couple thousand lines of code. While still substantial, this is a significant reduction.

The purpose of document is to describe FilterAlg in enough detail that the user can understand the code well enough to investigate new logic by amending their local copy.

One important note: only 15 of 16 vetoes are coded. Veto number 16 is not included in FilterAlg at this point (the "skirt veto").

FilterAlg Files

FilterAlg consists of three files:

FilterAlg.cxx:	The implementation of the Gaudi algorithm, and all the filter logic.
FilterAlg.h:	The header file with the class declaration.
FilterAlg_acd.h:	A modified version of OnboardFilter's TFC_acd.C. This file contains code that extends projections (2 dimensional tracks) to the ACD in order to determine which (if any) ACD tiles were intersected. We will call this "track-tile" matching for the purposes of this document.

The first two files contain original implementations of OnboardFilter logic, while the third is almost an exact copy of a file in JJ Russell's C code. Initially, FilterAlg contained an implementation of the track-tile matching that was based on the standard Gleam geometry. The code would project a track vector from a coordinate in 3-D space to the surface of the ACD, and determine whether the tile was "hit." The method was precise, but too precise, as it did not completely agree with OnboardFilter. It turns out that OnboardFilter does not use exactly the same geometry contained in Gleam, though it is based on it. OnboardFilter must use a geometry description that takes the Gleam geometry as input, but then make small approximations regarding the boundaries of tiles and the positions of silicon strips. It is easiest for OnboardFilter to use integer quantities to describe the geometry of the LAT, and the fundamental unit of distance is the number of strips. ACD tiles, like everything else, are described in terms of X strips wide and Y strips across, rather than in mm or other decimal units. OnboardFilter also deals with the boundaries between tiles or the overlap of tiles by assigning its own fictitious tile boundaries to be somewhere between the physical tile boundaries. This is done out a desire and perhaps necessity for simplicity, but it does disagree with the boundaries contained in the Gleam geometry description.

The approximations in the OnboardFilter geometry description prohibit the use of the Gleam geometry when doing track-tile matching. If we desire 100% agreement between FilterAlg and OnboardFilter, the same geometry must be used for both. This means that although one could still describe a track as a Hep3DVector (which can be constructed with input in any units whatsoever), and try to project the track to the ACD, one would still encounter rounding errors when doing the projecting. To avoid these problems, OnboardFilter uses very simple, yet specific, methods to project the tracks to the tile planes. In order to get exact agreement between FilterAlg and OnboardFilter, it would be necessary to use these same methods. They are so simple (a handful of lines of code), that reproducing them would essentially mean copying them line by line. Instead of doing this we can just borrow the file where these methods are used, and alter it to work with FilterAlg.

Although this compromise ensures that the track-tile matching agrees, we are not creating an original implementation that can be used to evaluate the validity of OnboardFilter's methods. One way to mitigate this uncertainty with a fair degree of confidence is to run FilterAlg using the Gleam geometry description, and take a detailed look at events that disagree with OnboardFilter. When one does this, one sees that it is because tracks miss (or hit) tiles just a few mm from an edge. The reasons why can be traced back to OnboardFilter's approximated geometry description. Not all events can be examined in this way, however, so there remains the possibility that OnboardFilter has bugs. The ambitious user may decide to write an original implementation of the track-tile matching in order to investigate this question further.

Running FilterAlg

Running FilterAlg is as simple as adding a line to the Gleam basicOptions file where FilterTracks is called: `Triggered.Members += {"FilterAlg"};`

The user also has the choice of switching on and off portions of FilterAlg code with a few job options:

`FilterAlg.UseGleamAcdVetoes` is set to "0" by default. Setting it to "1" will force FilterAlg to use Gleam's AcdDigis to form lists of ACD tiles over veto threshold. This is instead of using OnboardFilter's lists of tiles "hit."

`FilterAlg.evaluateCal1` is set to "0" by default. Setting it to "1" will force FilterAlg to call the "`evaluateCal1()`" function, which includes some logic that is not currently implemented in OnboardFilter, but was at one point. The FilterAlg version of this function has been checked for accuracy against the OnboardFilter version, so the user can turn it back on with confidence if so desired.

`FilterAlg.evaluateAcdUncomment1` is set to "0" by default. Setting it to "1" "uncomments" a few lines of logic that has not been in use in OnboardFilter for some time. It is included in FilterAlg for completeness, and if the user is interested in looking at it, search FilterAlg.cxx for "`m_evaluateAcdUncomment1`." Note that "uncommenting" this code only turns it on in FilterAlg. For a comparison of this code with that in OnboardFilter, the user will have to uncomment the corresponding code in OnboardFilter's DFC_filter.c file.

`FilterAlg.UseFilterProjecting` is set to "1" by default. Setting it to "0" will cause FilterAlg to use a method of projecting a track to the ACD that is not used in OnboardFilter. We recommended that the user keep this option set to zero.

`FilterAlg.UseGleamTileGeometry` is set to "0" by default. Setting it to "1" will cause FilterAlg to use Gleam's geometry for the ACD. We recommend that the user keep this option set to zero.

`FilterAlg.DoPrjColumnMatchCheck` is set to "0" by default. Setting it to "1" adds some additional functionality that is not found in OnboardFilter. When OnboardFilter tries to determine whether a projection intersects a side face ACD tile, it only checks to see if a horizontal row of tiles has a veto tile. Setting this jobOption also checks the vertical column for a veto tile. If a set of projections determine the intersection of a row and a column that happens to coincide with a known veto tile, there is said to be a track-tile match. Because this functionality is not included in OnboardFilter, if the user decides to use this additional logic, FilterAlg will automatically use the Gleam tile geometry and will not use Onboard Filter's method of projecting a track. (UseGleamTileGeometry → 1, UseFilterProjecting → 0)

FilterAlg Organization

FilterAlg is modeled after the main file in OnboardFilter, DFC_filter.C, where logic is grouped into a few main sections. FilterAlg uses the same basic grouping, and the following functions are called in this order from FilterAlg::execute()

```
CheckCal();
evaluateAcd();
evaluateAtf();
evaluateZbottom();
evaluateCal1();
tkrFilter();
```

This is the basic organization of function calls in FilterAlg:

```
initMasks()
CreateLocals()
      FFS()
useAcdDigi()
CheckCal()
      acdFilter()
      setVeto(m_NOCALLO_FILTER_TILE_VETO)       //VETO #30
      sumBits()
            cntBits()
      setVeto(m_SPLASH_0_VETO)                  //VETO #29
      AFC_splash()
            submits()
                  cntBits()
      setVeto(m_SPLASH_0_VETO)                  //VETO #29

evaluateAcd()
      setVeto(m_E0_TILE_VETO)                   //VETO #28
      acdFilter()
      setVeto(m_E350_FILTER_TILE_VETO)          //VETO #27
      sumBits()
            cntBits()
      setVeto(m_SPLASH_1_VETO)                  //VETO #26
      AFC_splash()
            submits()
                  cntBits()

evaluateAtf()
      compare()
            getPossible()
                  FFS()
            getTowerID()
            getMask()
            removeTower()
            coincidenceLevel()
                  reorderLayers()
                  cntBits()
            getStart()
                  FFS()
            setVeto(m_TOP_VETO)                 //VETO #25
            getRowMask()
            setVeto(m_SIDE_VETO)                //VETO #24

evaluateZbottom()
      getActiveTowers()
      getTowerID()
      checkPlanes()
            sumBits()
                  cntBits()
      setVeto(m_ZBOTTOM_VETO)                   //VETO #23

evaluateCal1()
      setVeto(m_EL0_ETOT_01_VETO)               //VETO #22
      setVeto(m_EL0_ETOT_90_VETO)               //VETO #21
```

```
tkrFilter()
      TFC_geosLocate()                        //External Function Call
      setVeto(m_TKR_EQ_0_VETO)                //VETO #17
      createTowerMask()
      FilterAlg_acdProjectTemplate()          //In FilterAlg_acd.h
      ACDProject()
            cntBits()
            convertId()
            acdId()                           //External Function Call
            getShapeByID                      //External Function Call
            getTransform3DbyID                //External Function Call
            projectionLoop()
                  convertPrj()
                        findStripPosition()   //External Function Call
                  getAcdCoord()
                        HepPoint3D()          //External Function Call
                  getTileBoundaries()
                        TFC_geosLocate()      //External Function Call
                        convertId()
                  reconstructTileNumber()
      evaluateTiles()
      if (m_useFilterProjecting)
      {
      FilterAlg_acdProject()                  //In FilterAlg_acd.h
            prjAcd_Top()
                  projectXYtoAcdTop()
                        findAcdTopMask()
            prjAcd_XM()
                  projectXYMtoAcd()
                        findAcdSideMask()
            prjAcd_XP()
                  projectXYMtoAcd()
                        findAcdSideMask()
            prjAcd_YM()
                  projectXYMtoAcd()
                        findAcdSideMask()
            prjAcd_YP()
                  projectXYMtoAcd()
                        findAcdSideMask()
            prjAcd_XMS()
                  projectXYStoAcd()
                        findAcdSideMask()
            prjAcd_XPS()
                  projectXYStoAcd()
                        findAcdSideMask()
            prjAcd_YMS()
                  projectXYStoAcd()
                        findAcdSideMask()
            prjAcd_YPS()
                  projectXYStoAcd()
                        findAcdSideMask()
      }
      setVeto(m_TKR_TOP_VETO)                 //VETO #20
      setVeto(m_TKR_ROW01_VETO)                //VETO #19
      setVeto(m_TKR_ROW23_VETO)                //VETO #18
      //SkirtProject()                         //not coded
      //setVeto(m_TKR_SKIRT_VETO)              //VETO #16 - not active
      setVeto(m_TKR_EQ_0_VETO)                 //VETO #17 (again)
      setVeto(m_TKR_LT_2_ELO_VETO)             //VETO #15
```

Detailed Description

<u>FilterAlg::initialize()</u>

When FilterAlg is first initialized, the `initMasks()` function is called. `initMasks()` sets up three bit masks for each ACD tile in the top face, and the top two rows of the side faces (rows 0 and 1), that are later used to determine whether a splash veto condition has been satisfied. In order to evaluate the splash veto, it is necessary to know which tiles surrounded the hit tile (a maximum of 8). To do this, each tile is assigned a number from 0-64 that is used to index the three mask arrays. Later, when FilterAlg needs to recall the necessary masks, it can select them with the correct array index. The three mask variables are called "m_MaskFront[], m_MaskY[], and m_MaskX[]." m_MaskFront[5], for example, holds a bit mask of tiles that surround tile number 5. It so happens that tile 5 sits on the edge of the top face of the ACD, and touches the X- side, but does not touch any Y faces. For this reason, m_MaskX[5] holds a value, but m_MaskY[5] is zero. This will make more sense after taking a look at the numbering conventions for the ACD.

In FilterAlg, we have two numbering conventions for the ACD. One system is used to access the correct masks, which are held in 65 element arrays, so we just number the tiles from 0-64. The top face is numbered as shown in this diagram (looking down). The two numbering schemes do not differ for the top face.

```
                     Y+ face
      -------------------------------------
      |  20  |  21  |  22  |  23  |  24  |
      |      |      |      |      |      |      ^
      -------------------------------------    /|\   Y+
      |  15  |  16  |  17  | 18   |  19  |      |
      |      |      |      |      |      |      |
      -------------------------------------     |
X-    |  10  |  11  |  12  |  13  |  14  |     -----------> X
Face  |      |      |      |      |      |
      -------------------------------------
      |   5  |   6  |   7  |   8  |   9  |     X+ Face
      |      |      |      |      |      |
      -------------------------------------
      |   0  |   1  |   2  |   3  |   4  |
      |      |      |      |      |      |
      -------------------------------------
                     Y- Face
```

And the side faces have two numbering schemes. The basic scheme that is used throughout FilterAlg is the top number on each of the tiles, and the number in parentheses below it is the corresponding mask index for that tile.

```
                    -------------------------------------
                    |  0   |  1   |  2   |  3   |  4   |
                    | (25) | (26) | (27) | (28) | (29) |
                    -------------------------------------
The Y- Face         |  5   |  6   |  7   | 8    |  9   |
                    | (30) | (31) | (32) | (33) | (34) |
                    -------------------------------------
                    |  10  |  11  |  12  |  13  |  14  |
                    |      |      |      |      |      |
                    -------------------------------------
                    |              15                  |
                    |                                  |
                    -------------------------------------



                    -------------------------------------
                    |  4   |  3   |  2   |  1   |  0   |
                    | (39) | (38) | (37) | (36) | (35) |
                    -------------------------------------
The Y+ Face         |  9   |  8   |  7   | 6    |  5   |
                    | (44) | (43) | (42) | (41) | (40) |
                    -------------------------------------
                    |  14  |  13  |  12  |  11  |  10  |
                    |      |      |      |      |      |
                    -------------------------------------
                    |              15                  |
                    |                                  |
                    -------------------------------------



                    -------------------------------------
                    |  4   |  3   |  2   |  1   |  0   |
                    | (49) | (48) | (47) | (46) | (45) |
                    -------------------------------------
The X- Face         |  9   |  8   |  7   | 6    |  5   |
                    | (54) | (53) | (52) | (51) | (50) |
                    -------------------------------------
                    |  14  |  13  |  12  |  11  |  10  |
                    |      |      |      |      |      |
                    -------------------------------------
                    |              15                  |
                    |                                  |
                    -------------------------------------
```

```
                  ------------------------------------
                  |   0  |   1  |   2  |   3  |   4  |
                  | (55) | (56) | (57) | (58) | (59) |
                  ------------------------------------
The X+ Face       |   5  |   6  |   7  |  8   |   9  |
                  | (60) | (61) | (62) | (63) | (64) |
                  ------------------------------------
                  |  10  |  11  |  12  |  13  |  14  |
                  |      |      |      |      |      |
                  ------------------------------------
                  |                15                |
                  |                                  |
                  ------------------------------------
```

The entire ACD can be described with three words, one for the top/front face, one for the X faces, and one for the Y faces. When a bit is set to "1," the ACD tile is either masked off or over veto threshold (depending on the context).

```
Top face (also called face "0")

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
f e d c b a 9 8 7 6 5 4 3 2 1 0 f e d c b a 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| | | | | | | |2|2|2|2|2|1|1|1|1|1|1|1|1|1|1| | | | | | | | | | |
| | | | | | | |4|3|2|1|0|9|8|7|6|5|4|3|2|1|0|9|8|7|6|5|4|3|2|1|0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             |                                                 |
|             |<--------------------- TOP --------------------->|
|             |                                                 |

X faces (also known as faces "1" and "3")

3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
f e d c b a 9 8 7 6 5 4 3 2 1 0 f e d c b a 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|1|1|1|1|1| | | | | | | | | | |1|1|1|1|1|1| | | | | | | | | | |
|5|4|3|2|1|0|9|8|7|6|5|4|3|2|1|0|5|4|3|2|1|0|9|8|7|6|5|4|3|2|1|0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               |                               |
|<------------- X+ ------------>|<------------ X- ------------->|
|             face 3            |           face 1              |

Y faces (also known as faces "2" and "4")

3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
f e d c b a 9 8 7 6 5 4 3 2 1 0 f e d c b a 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|1|1|1|1|1| | | | | | | | | | |1|1|1|1|1|1| | | | | | | | | | |
|5|4|3|2|1|0|9|8|7|6|5|4|3|2|1|0|5|4|3|2|1|0|9|8|7|6|5|4|3|2|1|0|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               |                               |
|<------------- Y+ ------------>|<------------ Y- ------------->|
|             face 4            |            face 2             |
```

To see how one assembles the masks for a given tile, look at tile 5 again as an example. Referring to the tile diagrams, we see that tile 5 is on the top face, surrounded by 0,1,6,10, and 11 on the top, and by 0, 1, and 2 on the X- face. We want to set all of these bits to "1" in the appropriate word:

```
m_MaskFront[5] = 0000 0000 0000 0000  0000 1100 0100 0011 (in binary)
               = 0x00000C43 (in hex)
m_MaskX[5]     = 0000 0000 0000 0000  0000 0000 0000 0111 (in binary)
               = 0x00000007 (in hex)
m_MaskY[5]     = 0
```

<u>FilterAlg::execute()</u>

The first function called in FilterAlg::execute() is `CreateLocals()`, which assembles a few other variables that are used later in FilterAlg. The first of these is `m_triggered_towers`, which is a word that holds information regarding which of the sixteen towers had a 3-in-a-row tracker trigger. This information is simply lifted from the sixteen most significant bits of a word found in `DFC_filter.C, m_tcids. CreateLocals()` reorders the bits such that tower 0 corresponds to bit 0, tower 1 to bit 1, and so on. If a bit is set, OnboardFilter says the tower has a tracker trigger.

Next, `CreateLocals()` forms two 16 element arrays, of which each element holds a word that describes where in a tower there exist coincidences of layers. This allows FilterAlg to look up a tower, and determine whether there were any 2-in-a-rows or 3-in-a-rows, and if so, which layer they started in. There are 18 bi-layers in each tower (consisting of the overlap of an X layer and a Y layer), numbered from 0 to 17, with 0 closest to the calorimeter, and 17 closest to the front of the ACD. `m_coincidences_3[tower]` holds 16 words, one for each tower, with a bit set at the position of the start of a 3-in-a-row (3 XY layer pairs hit in a row). Bit 0 corresponds to layer 0, bit 1 to layer 1, etc. `m_coincidences_2` holds something similar, but for 2-in-a-rows instead. For example, if there are XY layer pairs hit at 3, 4, 5, and 6, in tower 4, a list of layers hit could look like this in binary:

```
               654  3
00 0000 0000 0111 1000
```

Then the starting layers of 3-in-a-rows would look like this:

```
00 0000 0000 0110 0000
```

Therefore, m_coincidences_3[4] = 0110 0000 = 0x60

The starting layers of 2-in-a-rows would look like this:

```
00 0000 0000 0111 0000
```

So, m_coincidences_2[4] = 0111 0000 = 0x70

The coincidence arrays are formed from a few peculiar variables taken directly from OnboardFilter: `xy00[], xy11[], xy22[], and xy33[]`. There are 16 of each of these, one for each tower. `xy00` is actually the only fundamental of the four, as the rest of them are derived from it with via bit manipulations. Each of these variables contains raw information on which layers were hit, but the ordering scheme is unusual.

`xy00` has the layers ordered like this:

```
         1  3  5  7  9  11  13  15  17  0  2  4  6  8  10  12  14  16
Then xy11:                              1  3  5  7  9  11  13  15  17
     xy22:                            0  2  4  6  8  1012  14  16  --
     xy33:                                 1  3  5  7  9   11  13  15
```

It is also important to note that `xy00 = xcapture & ycapture`, which are ordered the same as `xy00` (obviously), but only contain the layers hit in the x and y directions respectively. So here we have in `xy00` the layers (bi-layers) hit for each tower, and some interesting things can be done with the derived quantities `xy11`, `xy22`, and `xy33`. If we want to know where a 3-in-a-row began, we just do this: `xy00 & xy11 & xy22`, and `xy00 & xy11 & xy33`! This is exactly what is done in `CreateLocals()`:

```
xy1[tower] |= m_xy00[tower] & m_xy11[tower] & m_xy22[tower];
xy2[tower] |= m_xy00[tower] & m_xy11[tower] & m_xy33[tower];
```

And if we just want to know where 2-in-a-rows start, we do this:

```
xy1[tower] |= m_xy00[tower] & m_xy11[tower];
xy2[tower] |= m_xy11[tower] & m_xy22[tower];
```

`CreateLocals()` also takes care of the remaining task, which is to reorder the bits into a more user friendly format. Instead of this:

```
1  3  5  7  9  11  13  15  17  0  2  4  6  8  10  12  14  16
```

we end up with `m_coincidences_3[]` and `m_coincidences_2[]` in this order:

```
17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
```

The only function remaining before the event filtering begins is the optional call to `useAcdDigi()`, which will force FilterAlg to look up which tiles Gleam says are veto tiles, rather then get that information from OnboardFilter.

Event Filtering: FilterAlg::CheckCal()

The first function that contains event filtering logic is `CheckCal()`, which checks whether there were cal-lo or cal-hi triggers, and whether there were any splash vetoes. The logic of this section is identical to that in `DFC_filter.C`, which is described in detail in another document (LAT-TD-02979), so we will not duplicate the description here. Instead, we want to describe how FilterAlg implements the logical equivalent, especially with regard to the `AFC_splash()` function.

The OnboardFilter concept of a splash veto is not complicated. If there is a cal-hi trigger and four or more ACD tiles are hit anywhere in the ACD, there is a splash veto. Alternatively, if there is a cal-hi trigger and only three ACD tiles hit, if those hit tiles follow certain rules regarding their respective placement, OnboardFilter may execute a splash veto. `AFC_splash()` examines the pattern of hits in the ACD and makes this determination.

`FilterAlg::AFC_splash()` first loops over the tiles in the word that contains the top face hits, extracting the tile bit number of the first hit tile using the `FFS()` function. Given a 32 bit word, the `FFS()` function returns the number of bits from the left, or from the most significant position. Subtracting this number from 31 gives the bit position. The idea is to get the tile that is hit so that we can look up the appropriate masks and do the splash veto calculation. Because the bit number for the top face is also the mask number (refer back to the discussion on `initMasks()`), we can simply retrieve the splash veto masks like this with this bit number as the array index.

OnboardFilter defines this splash veto condition as met if two conditions are satisfied. The first is that there are three hit tiles, including the tile we are looping over, in the top or upper two rows of the ACD. This is simply a matter of masking off the top and upper two rows of the ACD veto words and counting the remaining bits with a call to `sumBits()`. The second condition is that no more than one hit tile may be adjacent to the tile we are looping over, and this is where the masks come into play. So we AND the masks for each face with the ACD words containing tile hits, and demand that the sum of the remaining bits be no more than one.

If no splash is found when looping over the top/front face of the ACD, FilterAlg moves on to the Y faces. The procedure is the same, except that there is a function that maps the bit number in the Y faces ACD veto word to the mask array index that corresponds to the same tile. The two functions that do this mapping are `mapBitY()` and `mapBitX()`, and their operation is elementary. Finally, if no splash vetoes were found on the Y faces, the X faces are examined.

Event Filtering: FilterAlg::evaluateAcd()

As a general rule, veto logic gets more complicated the farther along one progresses. The next section of code, called "`evaluateAcd()`," is perhaps the only violation of this rule. Vetoes are set if any tiles are hit and there is less than 10 MeV in the cal, or if any of the "filter" tiles are hit (meaning the front or top two rows of the ACD) and the energy is less than 350 MeV. Also in this section is some splash veto checking that has long been commented out in OnboardFilter, and can be turned on with a jobOption in FilterAlg. It looks for a splash veto when the energy is less than 40 GeV, unlike the case described above which only looks when there is a cal-hi.

Event Filtering: FilterAlg::evaluateAtf()

OnboardFilter, and thus FilterAlg, begin to get more complicated with the "`evaluateAtf()`" section that follows. This logic is only evaluated when the energy is 5 GeV or less, and it does something very similar to one of the proposed trigger throttling methods. It checks to see if a triggered tower is shadowed by an ACD veto tile. The bulk of the `evaluateAtf()` logic is found in the `FilterAlg::compare()` function, which begins by assembling a list of towers to check ("`possible_towers`"). The towers examined are those that have 2-in-a-rows or 3-in-a-rows. Looping over these towers, FilterAlg retrieves the tower id and the masks of tiles that shadow that tower. If there are any veto tiles in coincidence with the masks, the checking continues. The tower is considered "valid" if `coincidenceLevel(tower)` says that there were either 6/6 or 7/8 layers struck. This means we either have a 3-in-a-row, or *almost* a 4-in-a-row.

It is worth pausing here to describe the details of the `coincidenceLevel()` function, because the format of some of the variables can be confusing. Recall from the description of `CreateLocals()` that `xcapture[]` and `ycapture[]` were in an unusual format:

```
1 3 5 7 9 11 13 15 17 0 2 4 6 8 10 12 14 16
```

The first thing that `coincidenceLevel()` does is reorder `xcapture[]` and `ycapture[]` to the more familiar format also used in `CreateLocals()`:

```
17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

It then proceeds to look for 6 out of 6 layers hit, and then 7 out of 8, by masking off either 3 or 4 layers of `xcapture` and `ycapture` and counting the bits, then repeating until all the layers are checked. This is a simple procedure, and the highest numbered layer of the topmost coincidence is returned. In other words, if there is a 3-in-a-row (6/6) starting at layer 15 and at layer 10, layer 15 will be noted. However, if there is a 7/8 starting at layer 16, it will be returned instead of the 3-in-a-row at layer 15. The level of coincidence is irrelevant (7/8 does not trump 6/6 or vice versa), it is the position that matters.

Once the coincidence level has been determined and the `valid_tower` flag set, there is some additional checking to make sure that FilterAlg has found the starting layer for coincidences. Once the starting layer has been determined, we move on to looking for the tile-tower shadowing. If there is a valid tower, the code checks to see if there is an ACD veto tile among the possible front face shadowing tiles for that tower. For each tower in the center (towers 5, 6, 9, and 10), there are four possible tiles to check in the ACD front face. For side towers, there are four front face tiles, plus four side tiles (only the top two rows count). For corner towers, there are four front face tiles, and eight side tiles to check. If there is a front face tile match and the starting layer of the coincidence was in the top three layers (layer 15, 16, or 17), veto number 25 is set. If these conditions are not met, the code checks to see if there are side face tile matches. If so, it then looks at the starting layer of the coincidence. The rule in OnboardFilter is that the starting layer of the coincidence must be shadowed by a tile above it. In other words, if the starting layer is 13 or higher, the match is only valid if the veto tile is in row 0 of the side face. If the starting layer is 6 to 12, the veto is valid if the tile is in row 0 or 1. And if the starting layer is down near the cal (layer 5 or less), the veto tile is valid if it is in row 0, 1, or 2. To check this, FilterAlg uses `getRowMask()` to mask off the appropriate rows for the start layer. If a shadowing side tile is found that is in an acceptable row given the starting layer of the coincidence, veto number 24 is set.

Event Filtering: FilterAlg::evaluateZbottom()

The next section in FilterAlg reproduces the "z bottom" veto logic, which requires that four out of six planes closes to the calorimeter be hit if there is greater than 100 MeV in the cal. The method is simple: call `getActiveTowers()` to get a list of any towers that have any number of X or Y layers hit. If there is just one out of 36 layers hit, that tower is considered active. Then for each tower, a call to `checkPlanes`() counts the number of layers hit in the six layers closest to the cal. This is done by masking off three bits in `xcapture` and three bits in `ycapture` (for layers 0, 1, and 2), summing the number of bits remaining, and seeing whether the total is 4 or more. If not, a veto is set.

Event Filtering: FilterAlg::evaluateCal1()

`evaluateCal1()` is a section of logic that evaluates two possible vetoes, but has been commented out in OnboardFilter for some time. It is included here in case the user wants to see what its effects are. If the energy is greater than zero but less than 300 MeV, the ratio of the energy in layer zero of the cal to the total energy must be greater than 1 percent, but less than ninety percent. If the ratio is less than or equal to 0.01, the event is vetoed, and if the ratio is greater or equal to 0.90, the event is vetoed.

Event Filtering: FilterAlg::tkrFilter()

Each section of veto logic until this point has been relatively straightforward, but here things get more complicated as we begin to describe the track related vetoes. The concepts behind theses six vetoes are actually quite simple. Three vetoes look for intersections of tracks with veto tiles, one looks for tracks that escape in the gap between the cal and the bottom of the ACD (known as the skirt region), and two vetoes are active when there are not any, or not enough tracks found. The implementation of these vetoes is not horribly difficult, but it is more involved than those found above.

The first thing to check is whether there are any towers with 2-in-a-rows or better. If not, veto number 17 is set and the tkrFilter() routine exits. If there are, however, a call to `createTowerMask()` makes a mask of towers that have 2-in-a-rows or better. Then a call to `FilterAlg_acdProjectTemplate()` sets up another mask called "`dispatch`," which will later tell a track projecting routine which ACD faces to check for track-tile matches. If the ACD has no hits, for example, `dispatch` will equal zero.

With the preliminaries taken care of, the main loop over towers begins to execute. All towers with a 2-in-a-row or better are examined, starting with the lowest numbered tower. The reader will note that the tower id is obtained with a call to `FFS(tmsk)` and the subtraction of "16." This is because `tmsk` is based on `m_tcids` (taken directly from OnboardFilter), and `m_tcids` is numbered from left to right. `tmsk` is also numbered from left to right, with tower 15 in the least significant bit position, and tower 0 in the 16th bit position from the right. `FFS()` returns the number of spaces from the *most significant* end of a 32 bit word, so `FFS(tmsk)-16` gives us the tower number. After reading in the tower number, it is removed from the `tmsk` list until no towers are left.

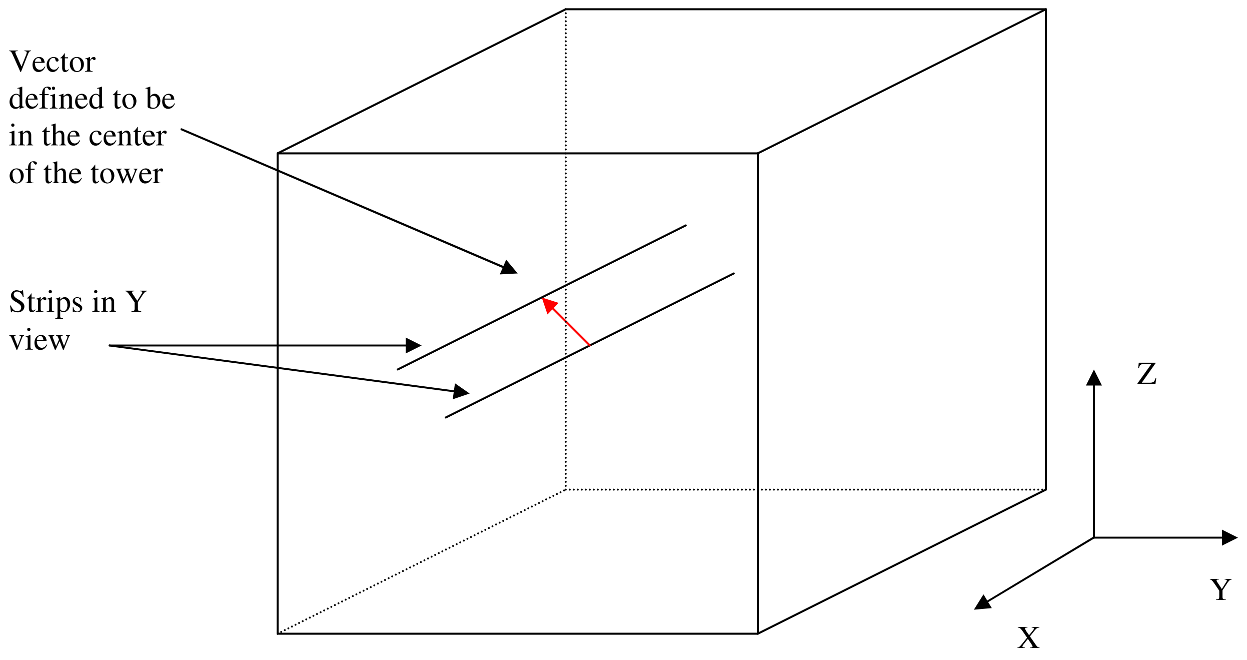
Moving along, the user will read a comment in the code that states that the projections (tracks), are copied directly from OnboardFilter at this point. If the user desires to form tracks differently, they will have to write a function to do this, and make the formatting the same as the current projections.

At this point, the track finding presents the user with some options. As discussed in the first sections of this document, there are different geometries and different track projecting methods available to the user. Currently, the code is written such that OnboardFilter's geometry and track projecting methods are used. But because other methods are available to the user (one might say because of historical reasons), the description of FilterAlg's inner workings must branch a bit here.

We will go in order of what the user encounters when reading through the code. After grabbing the projections from the TDS, where they are stored after OnboardFilter executes, `tkrFilter()` makes a call to `ACDProject()`. It is important to note that the calculations in `ACDProject()` are only relevant when the user has instructed FilterAlg to not use the OnboardFilter method of track-tile matching. If the user leaves the jobOptions at their default values, the calculations in `ACDProject()` will be overwritten later. `ACDProject()` begins by looping over each ACD veto tile, starting with the front face. The tile and face information is passed to `convertId()`, which returns a row and column number for that tile. Passing the row and column number to `acdId()` creates an `acdId` object, which is then used to get the Gleam `volumeId`, tile shape, transform, and ultimately the position and dimensions of the tile. These few lines of code are borrowed directly from `AcdRecon()`.

Next, we decide to pick a "view" to look at. Recall that strips are parallel to either the X or Y axis depending on which layer they are in. When dealing with strips, we have to specify which orientation by specifying a view. When the view is zero, we are talking about strips that are parallel to the Y axis, which means that the horizontal distance between these strips is measured along the X axis. This is known as the "X view." Likewise, view one is the Y view. `ACDProject()` gets the count of projections in one view, meaning they are described by strips in that view, and passes the count to the `projectionLoop()`, which loops over all the projections in that tower in order to determine whether they intersect the tile that `ACDProject()` is currently focused on.

The basic concept behind the `projectionLoop()` function is that it loops over each projection in one view, converting the projection to a vector, and then determines where the extended vector intersects the ACD. OnboardFilter only considers projections that start in layer eight or higher, so we do the same here. The process of converting the projection into a vector is simple. A call to `convertPrj()` returns a `HepVector3D` that is created from two points. One point is the top-most hit in the projection's list of hits, and the other is the second hit in that list, just one layer down. These are the same hits that are used by OnboardFilter to form its projections. The Cartesian coordinates of these hits are fetched by a call to `findStripPosition`(), a function that we borrow from `FilterTracks.cxx`, the code that calculates and draws tracks in the Gleam Event Display. Because `findStripPosition()` only deals with one view at a time, it only knows two coordinates of any hit, which are the two coordinates that define the position of the strip within a tower. The strip is effectively a line that extends the length of a tower, not a point within the tower, so if one wants to specify a three-dimensional coordinate for a hit on a strip, one must arbitrarily decide where to place the hit along the strip. `findStripPosition()` uses the middle of the strip as its default third coordinate. So what we have for any given projection is a well defined vector in two dimensions that is arbitrarily placed in the center of a tower. Because the placement in the unknown dimension is arbitrary, and the vector is really only defined by two parallel strips at different heights and displacements within the tower, there are an infinite number of vectors that can be drawn between these strips, which effectively defines a plane. This is why these "tracks" are referred to as "projections." They are projections of a plane onto a two dimensional surface. Because these projections are only 2-dimensional "tracks," we need two of them, one in each view, to define a complete 3-dimensional track.



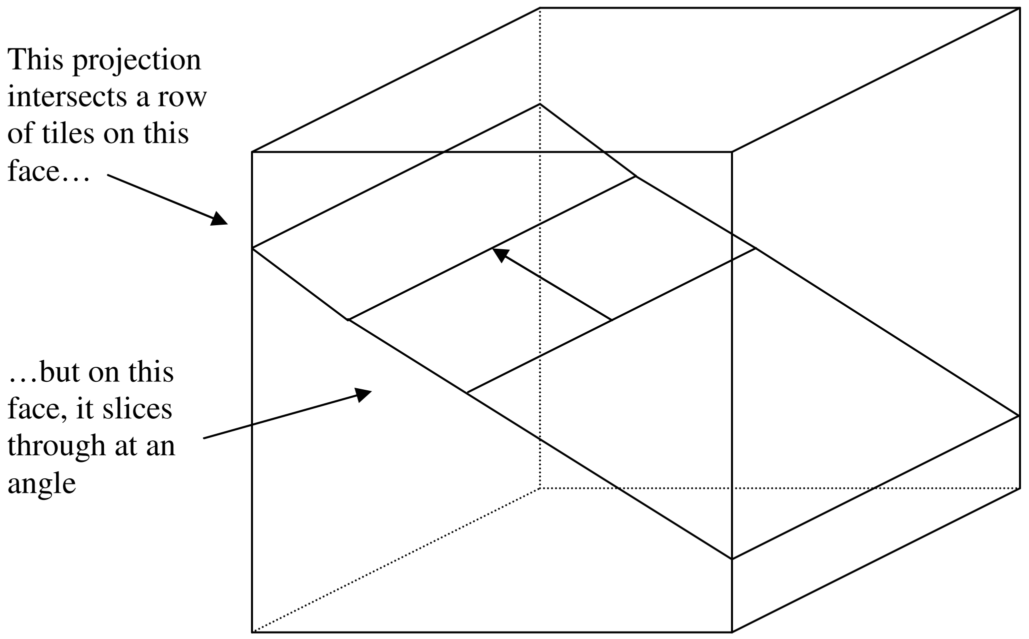
Furthermore, when we extend a projection to an ACD face, it draws a line on the face, not a point. If the tile in question is on the front face, for example, a Y view projection will define a row of possible tiles, and an X view projection will define a column. If the overlap of a row and column coincides with an ACD veto tile, we can say that the 3-D track that defines the 2-D projections intersected the hit ACD tile.

So we can see how having a projection defined, one can extend it to the ACD face to see where it intersects. The `getAcdCoord()` function does the projecting, first noting which face to project to (information we have because we are focused on one specific ACD tile at a time), and then calculates the distance to that face. The calculations for projecting to the top face are actually very simple, while projecting to the side faces is a bit more complicated because one must take care to make sure that the sign of each number is correct (the top face is always "up," but the side faces can be "in front of" or "behind" the vector). The other reason for added complexity when projecting to the side faces is some added functionality that FilterAlg contains. This added functionality, which we will describe soon, requires knowing the boundaries of the ACD tile currently under examination. These boundaries can be extrapolated from two points contained in `boundCoord1` and `boundCoord2`. Furthermore, we can tell if a projection is contained within these boundaries by checking four points, one on each of the four boundary lines of a tile. These four points are contained in `checkPoint1`, `checkPoint2`, `checkPoint3`, and `checkPoint4`.

Once these quantities are computed, the program returns to `projectionLoop()`, where it uses its knowledge of the ACD tile coordinates and the coordinates of the projection-ACD intersection to determine whether the intersection was within tile boundaries. One possible source of error here comes from the fact that the ACD tiles do have a finite thickness (most are 1 cm thick), and we are treating them as planes. It is possible for a projection to "miss" a tile, but actually graze the corner. At this point, the code branches slightly depending on whether the user has selected the Gleam tile geometry or the OnboardFilter tile geometry. If using the Gleam geometry, it checks to see if the distance from the center of the tile to the point of intersection was less than or equal to the distance from the center to the edge. If so, the row or column of the tile was hit, depending on which view we are looking at. We do not yet know if the tile itself was hit – for that we need the intersection of a perpendicular column/row of tiles. Now if we have not instructed FilterAlg to use the Gleam tile geometry, it will use that of OnboardFilter (the default). The boundaries of the tile are selected from the `getTileBoundaries()` function, and the code checks to see if the intersection fell within these OnboardFilter boundaries.

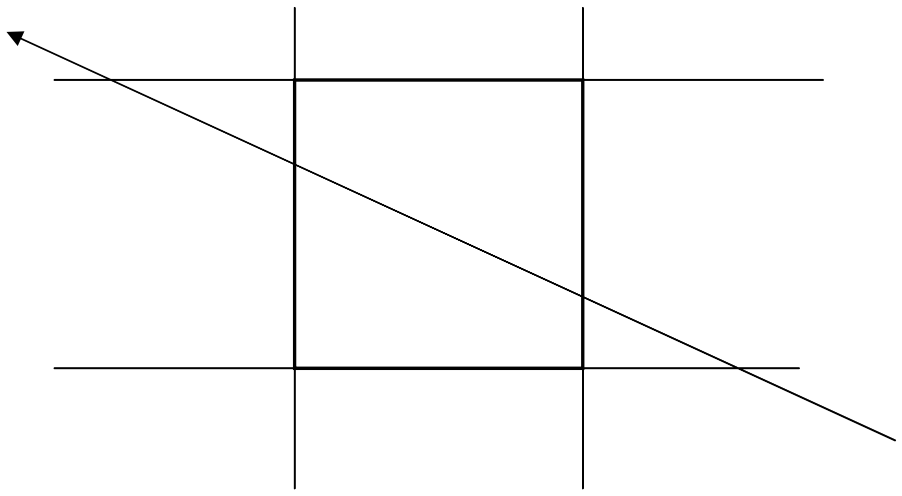
So far, this description has focused on projecting to the front face of the ACD. Most of this is still valid when projecting to the side, but there are some additional complications that one must deal with. First we will explain what happens for projections that are defined by strips that are parallel to the ACD face that we are projecting to. When the projection is extended to that face, it will always intersect a horizontal row of ACD tiles. Now, when looping over a side face tile in one of these faces, there is a corresponding set of towers that touch the face belonging to that tile. For towers that are *not* in the set of four, OnboardFilter looks at the X or Y distance between the first and third strips that belong to a projection in the same view as the face. It then requires that the distance between them be greater than two thirds the width of the tower. In other words, if the tile is in the X+ face, OnboardFilter looks at projections in the X view to see how long they are in the direction perpendicular to the X axis. If the distance condition is not satisfied, the projection is said not to have intersected the tile's row after all. Now if the tower containing the projection is in the set of four that touches the ACD face, we look at the slope of the projection, and require that it point towards the face, and that the projection not exit the tower too quickly.

Once `projectionLoop()` has finished looping over the projections in one view, it switches to the other view. Projecting to the top face is simple, as it was before, but projecting to the side faces gets much more difficult. The process of creating a vector and getting the intersection points on the ACD is the same as it was above, but now we have the option of looking at projections that are composed of strips that are perpendicular to the active ACD face. This means that the projections themselves are parallel to the face, so when we extend the projections, they slice through the ACD face at an angle, not as a horizontal line.


This projection intersects a row of tiles on this face…
…but on this face, it slices through at an angle

This is where the optional added functionality of FilterAlg comes into play. OnboardFilter only deals with the intersection of a projection with a row of tiles on the face that is parallel to the strips that define the projection. However, checking to see if a specific tile is intersected by a projection that is defined by strips perpendicular to the face is much more complicated. The next diagram shows how FilterAlg approaches the problem.

A side face tile with a projection intersecting it at an angle:



The reader will note that in the diagram above, the boundaries of the tile have been extended outward for illustrative purposes. Also note that the projection vector intersects each of the two horizontal boundaries and each of the vertical boundaries. FilterAlg calculates the intersection points for each of these four boundaries, and the position of the boundaries in `getAcdCoord()`, and the intersection points do exist for each projection that is not absolutely vertical. FilterAlg then checks to see if each of the four points lies within either the horizontal or vertical boundaries. If two intersection points lie within the appropriate boundaries, the projection intersected the tile. All that remains is to see whether a projection in the other view, from the same tower, intersected the row that the tile is in. If both of these criteria are met, there is a definite match for the tile we are looping over.

However, OnboardFilter is not this strict. It considers a track-tile match for a side face to be valid if the projection just intersected the horizontal row of tiles that the veto tile is belongs to. One thing that is very important to note for users who want to experiment with this extra functionality in FilterAlg, is that it is only possible to do it with the Gleam tile geometry. In FilterAlg's current incarnation, OnboardFilter's tile geometry is not programmed in for this kind of projection calculation. Regardless of which geometry is used, and whether the additional functionality is turned on, the projection loop ends at this point, either with a match or without. If it ends without a match, having looped over all projections in this tower, it exits, returning control to `ACDProject()`, which moves on to the next tile and repeats the process (still on the same tower).

Once `ACDProject()` finds a track-tile match, the routine calls `reconstructTileNumber()`, exits and returns control to `tkrFilter()`. Or if it does not find a track-tile match, it exits to `tkrFilter()`, which will move on to the next tower and repeat the entire process. Either way, control returns to `tkrFilter()`, which now knows which tile was intersected by a track, and this information is coded by the call to `reconstructTileNumber`(), which put numbers the tiles from 0 to 25 on the front, 25 to 40 on face 4 (Y+), 41 to 56 on face 2 (Y-), 57 to 72 on face 3 (X+), and 73 to 88 on face 1 (X-). This information is contained in "`firstmatch`," which is quickly passed on to `evaluateTiles()`. `evaluateTiles()` simply decodes firstmatch to determine whether the intersected tile was from the front face or a side row of the ACD, and if a side row, it says which one.

Moving on, FilterAlg counts the number of projections found in the current tower, and sums up the total number of projections found in all towers examined thus far. This information will be needed later. It then goes through more logic if the energy of the event was less than 5 GeV. If the FilterAlg default jobOptions are set, and the user is using OnboardFilter's method of track-tile matching, and there are projections in this tower, a call to `FilterAlg_acdProject()` does OnboardFilter's projection method.

OnboardFilter's method of projecting to the ACD is not very different than the alternate method coded into FilterAlg, but it is very specific. First it determines which ACD faces to look at depending on which had tiles that were hit. If no projections were found in a given view, some faces are eliminated from the list for efficiency's sake. `FilterAlg_acdProject()` proceeds to examine nine possible variations of the same theme, starting with a look at the top of the ACD.

If the dispatch mask says that the top face was active, `FilterAlg_acdProject()` calls `prjAcd_Top()`, which projects to that face. Keep in mind that this entire scheme of projecting to the faces is within a larger loop over individual towers, so it only focuses on one tower at a time. First it projects the X track projections to the top face, so it forms columns of Y tiles to check for intersections. Looping over the X projections, `prjAcd_Top()` calls `projectXYtoAcdTop()`, sending it information on the tower id, the distance from the tower to the Y column, the dimensions of the tower, and the details of the projection itself. `prjXYtoAcdTop()` takes this information and does four things:

(1) It gets the distance from the start of the projection to column of top tiles it is projecting to.
(2) It extends the projection to the tiles.
(3) It translates the coordinate of the intersection into a column of tiles.
(4) It keeps track of which columns of tiles have been intersected.

Control returns to `prjAcd_Top()`, which now has a mask of tiles that have been intersected by the projection. This process repeats for each X projection, and then for each Y projection. At the end of the `prjAcd_Top()` function, the masks of rows and columns are ANDed together to form a group of candidate tiles. These candidates are the intersections of the overlapping rows and columns, which means they are the tiles that may have been intersected by three dimensional tracks. `prjAcd_Top()` returns control to

`FilterAlg_acdProject()`, which compares this mask of candidate tiles with the ACD tiles known to be hit. If there is a match, a veto is set, because there is a track-tile match.

A similar set of routines are executed for the side faces, but they only calculate which horizontal rows of tiles are intersected, because determining the intersection of a column is computationally difficult. However, there are some safety checks included that make projecting to the side somewhat different than projecting to the top. The first case that is encountered is when projecting from a side tower to the adjacent face. `prjAcd_XM()` is used to project to the X- face from an adjacent tower, and deals with X projections exclusively. It calculates the distance from the tower to the ACD face, and then loops over the X projections. For each projection, it calls `projectXYMtoAcd()`, which computes slope of the projection and verifies that it is negative (it must be negative in order to exit the left/lower edge of the tower). The function then demands that the particle leave the tower within two layers, so the distance between the first and third hits are computed. If this condition is met, the projection is extended to the plane of the ACD face, where the Z position of the intersection is noted. This Z coordinate is then passed to `findAcdSideMask()`, which returns the row of tiles that correspond to that Z. All projections are looped over, and the masks are ANDed together until control returns to `FilterAlg_acdProject()`. The mask of tiles is compared with tiles that are known to be hit, and if there is a match, a veto will be set.

After projecting to the X- face, `FilterAlg_acdProject()` moves on to the X+ face with a call to `prjAcd_XP()`. This works almost the same way as `prjAcd_XM()`, except that the slope must be positive. Likewise, projecting to the Y- and Y+ faces are identical except for the change of view. The situation is slightly different when considering a tower that is not adjacent to an ACD face. `prjAcd_XMS()` is the first example of a function used to project to the X- face from a non-adjacent tower, and it deals with X projections exclusively. It calculates the distance from the tower to the ACD face, and then loops over the X projections. For each projection, it computes the X distance between the first and third hits in the projection (each projection must have at least 3 hits by definition), and it sends this information to `projectXYStoAcd()`. `projectXYStoAcd()` is a very simple function, which makes sure that the slope of the projection is greater than the width of the tracker divided by two layers. JJ says in his comments that if this condition is met, there is a reasonable chance that the track may hit only two layers in the adjacent tower. If this condition is met, the projection is extended to the plane of the ACD face, where the Z position of the intersection is noted. This Z coordinate is then passed to `findAcdSideMask()`, which returns the row of tiles that correspond to that Z. All projections are looped over, and the masks are ANDed together until control returns to `FilterAlg_acdProject()`. The mask of tiles is compared with tiles that are known to be hit, and if there is a match, a veto will be set. The same procedures are used when projecting from a non-adjacent tower to each of the three remaining side faces.

After `FilterAlg_acdProject()` has run through any or all of these nine possibilities, it has either found a track-tile match, or it has not. Regardless, control returns again to `tkrFilter()`, where the last few vetoes must be computed.

If the user has allowed FilterAlg to use the default track-tile matching method that we just described above, a variable named "`acd_firstmatch`" will have been set by the return value of `FilterAlg_acdProject()`. The contents of this variable tell us whether there were any track-tile matches (`acd_firstmatch` is non-zero), and if there were, in which face the first one was found. If the last four bits (the most significant bits) of `acd_firstmatch` equal "4," the track intersected a top face tile. tilematch is then set to "`TOPFACE`," and firstmatch is set to "88." The reader may recall the `tilematch` and `firstmatch` variables from the discussion of "`ACDProject()`" and "`evaluateTiles()`" a few paragraphs above. These are the same two variables that were set by `ACDProject()` and `evaluateTiles()`, but we overwrite them here because FilterAlg is using OnboardFilter's track-tile matching method instead of the proprietary FilterAlg method that is provided as an alternate (and was already calculated). If tilematch is not set to "`TOPFACE`," one of the side faces must be hit. Determining which row is just a matter of masking off the tiles in the top two rows and seeing if any hits were found there. The lower 28 bits of `acd_firstmatch` contain the list of tiles hit, so this is elementary. As for setting firstmatch to "88," if a track-tile match was found, it is simply an arbitrary value that we use to indicate such a match. "89" is the value used to indicate that there is no match. Likewise, if `tilematch` is not set to 0, 1, or 2 (which correspond to `TOPFACE`, `ROW01`, and `ROW23`), that also indicates that there is no match.

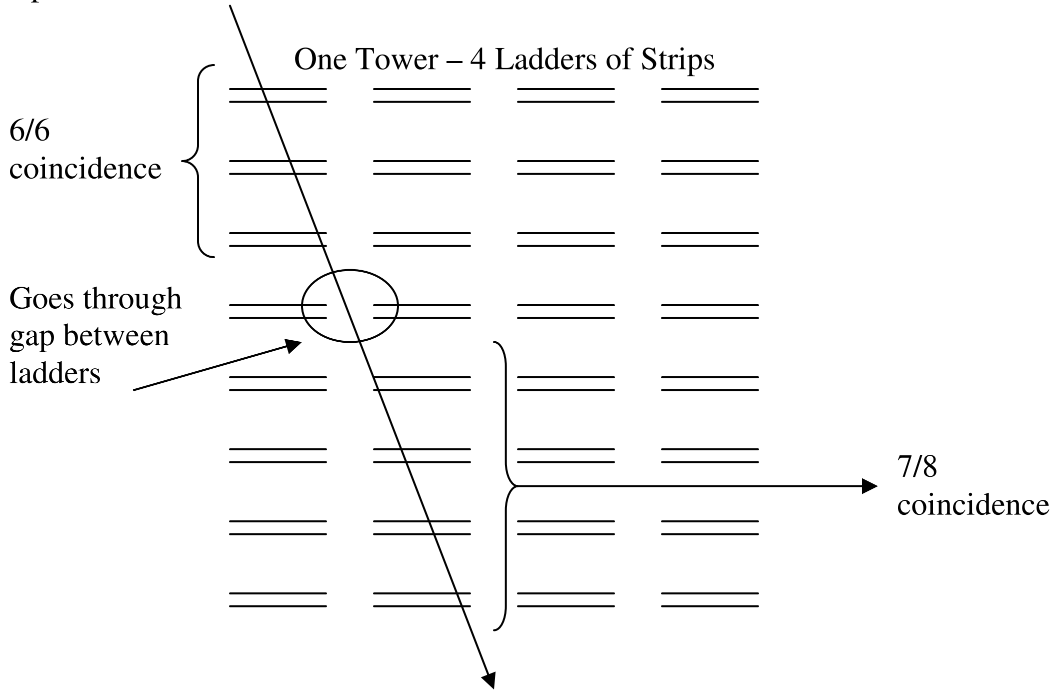
With the `firstmatch` and `tilematch` values set, `tkrFilter()` continues to a few lines of code that set up to three remaining vetoes. First, if there were any track-tile matches (`firstmatch` is less than 89), if the tile match was in the top face, and the energy was less than 30 GeV, a top face veto is set. If the match was in the upper two rows of a side face instead, and there was less than 10 GeV in the calorimeter, a different veto is set. Finally, if the match was in the bottom two rows of a side face, and the energy was less than 30 GeV, another veto is set. Note that only one track-tile match veto will be set, and the three that are possible just indicate which part of the ACD was intersected.

At this point, assuming all active towers have been examined, there are only three remaining vetoes to evaluate. One of these is not currently computed in FilterAlg: the skirt veto. The veto should be written such that it is active if a track passes through the skirt region between the ACD and the cal. Obviously, this can only be active if the energy of the event is zero. The last two vetoes are very simple. `tkrFilter()` checks to see if the total number of projections in the entire LAT is less than three. If so, and the energy is less than 350 MeV, a veto used to be set that said there was no evidence of two tracks. The thinking was that there should be two tracks detectable at low energies. However, this veto is currently commented out because it killed too many low energy gammas. Continuing, the code looks to see if the total number of projections is actually less than two, and if the energy is greater than 250 MeV. If so, a veto is set which says that no tracks were found. This is a reflection of the fact that at least two projections are necessary to form a track, and the energy cut limits the damage this cut does to low energy gammas. These were the last of the two vetoes, and FilterAlg is finished for this event.

Unfinished Business

FilterAlg currently computes 15 of the 16 vetoes, with the skirt veto as the exception. When the skirt veto is written, it will be necessary to do one of two things. Either the TFC_skirt.C file will have to be copied and adapted to work with FilterAlg, done in the same spirit as what was done with TFC_acd.C, or an original implementation of the skirt veto will have to be written into FilterAlg. If the latter path is taken, the OnboardFilter geometry must be used, or the results will not match. As explained early in this document, the necessity of using the OnboardFilter geometry was the driver for adapting TFC_acd.C instead of relying on an original implementation of track-tile matching. The same concerns may require the adaptation of TFC_skirt.C.

The only other difference between FilterAlg and OnboardFilter is found in the evaluateAtf() function. In this case, the result to trust, when they differ, is that of FilterAlg. The calculation in OnboardFilter has a small error in logic that cause it to miss a veto in a somewhat rare situation. If a track is formed that creates a 6/6 coincidence, then a gap of one or more XY layer pairs, and then farther down in the same tower there is a 7/8 coincidence, the starting layer is determined to be that of the 7/8, not the 6/6. This only happens when the track comes in at such an angle that it creates a 6/6, then passes through the gap between ladders of strips, thus missing an entire layer, and then goes on to form a 7/8 (see the diagram below). Because OnboardFilter looks for a 7/8 first, and does not bother looking for a 6/6 if a 7/8 is found, it misses the 6/6 that is higher in the tower. This is a problem, because veto 25 is only enacted if the starting layer is within the top three. JJ Russell is working on a fix to this problem, and OnboardFilter will be updated shortly. In the meantime, vetoes 25 and 24 (a related veto) may not agree between FilterAlg and OnboardFilter. In that case, the user should trust FilterAlg's output.


One Tower – 4 Ladders of Strips
6/6 coincidence
Goes through gap between ladders
7/8 coincidence