Code Documentation for *SUGaR-PR: Software to Update Gamma-ray Reconstruction in the Pair Regime*

Isabella Sanford

University of Maryland, Center for Research and Exploration in Space Science and Technology II (CRESST II)/ Southeastern Universities Research Association (SURA)/ NASA Goddard Space Flight Center

February 4, 2026

Contents

1 Selecting pair events

For the purpose of this analysis, an algorithm to identify the type of event (eg. Compton, pair, etc.) is not developed. The focus of this research is to make advancements in reconstructing gamma-ray pair production events in order to ascertain the initial direction of an incident photon as well as recover polarization information about its source. Therefore, if desired, the user has the option to utilize Monte Carlo (MC) information about the type of event for a simulated gamma-ray interaction in order to select only the true pair events to perform analysis on.

1.1 Pair event filtering from SIM file

The structure of SIM files generated from Cosima can be found in the Cosima documentation [1]. The scripts that filter out the pair events work on both ".sim" and ".sim.gz" files and the type does not need to be specified in the command line. The only things that must be specified in the command line are the name of the script performing the filtering (eg. PairEventSelection.py or PairEventSelection_MultipleFiles.py) and the name of the SIM file from which the pair events are desired to be extracted. The structure of the command line input is as follows:

```
python3  <NAME OF FILTERING FILE>  <NAME OF INPUT SIM FILE>
```

To extract the pair events, the file is read line by line. Each of these lines is stored as a "header line". When the first line containing "SE" is read, all of the header lines are written into the output file such that the header of the original SIM file is exactly copied into the new file.

After reading the line containing "SE", the script continues through each line until it finds a line beginning with "IA PAIR" (the MC indication of a pair event). If this line is found before the next "SE", all of the lines beginning from "SE" and up to the next "SE" are written to the output file. Events that do not contain "IA PAIR" are discarded.

This process continues until a line containing "EN" is read. Once this happens, all lines containing and after "EN" are read out as "footer lines" to the output file. The naming convention of the output file is:

<ORIGINAL FILENAME>_PairEventsOnly.sim.gz

NOTE: If an output file of the above name already exists and the script is run again, the output file will simply index by one. Ex:

<ORIGINAL FILENAME>_PairEventsOnly2.sim.gz

1.1.1 Single SIM file

To extract pair events from a singular SIM file, the script titled "PairEventSelection.py" must be run. The file performs the filtering process outlined above.

1.1.2 Multiple SIM files (for parallel runs)

When using mcosima for parallel runs, output SIM files are created as well as a concatenation file. To handle this, a few extra steps are required. If mcosima was used, the script titled "PairEventSelection_MultipleFiles.py" should be run.

The first thing to note is that the concatenation file is to be used as the input filename in the command line in order to properly combine all outputs. Ex:

```
python3  <NAME OF FILTERING FILE>  <NAME OF CONCATENATION FILE>
```

For the filtering, the concatenation file is accessed, and the lines in the concatenation file containing the names of the individual SIM files are read. For each identified filename, the script accesses that SIM file. It starts with the first one identified, and then proceeds to process as it would a singular file. When it is done, it reads the next file, and so on.

Note that the header of the first file is written to the output file. Then, the pair events from that file are written without the footer. For the next file, only the events are written, and this continues until the last file. When the last file is read, all of the events are written out and then the footer of that file is written out. Therefore, the output file contains information from all of the individual SIM files and has the same structure as a SIM file output from cosima, containing a header, event information, and then a footer [1].

2 Vertex identification

One of the main challenges with polarimetry in the pair regime is accurately identifying the location of a vertex for a given pair production event. This difficulty can be exacerbated particularly at low energies, where particle tracks in the detector are more "curved" resembling Compton-like events, making it difficult to trace the electron and positron tracks back to where they intersect. Additionally, at these low energies, particles are more likely to scatter within the detector, meaning that looking for a traditional pattern of a singular hit in one layer with two hits in the layer below may not always be accurate. Extraneous hits are also an issue since there can be multiple hits in the detector at once, where some are associated with a pair event and others are not.

To deal with these complications, the algorithm outlined in the following subsections was developed. All corresponding code can be found in the *VertexIDAndPlotting.py* script file. The command line usage of this file is explained in 2.6.

2.1 Classes

2.1.1 Vertex

Defines a vertex with 3D position coordinates (x, y, z), an event's associated ID number from the SIM file, and a function to compute the azimuthal angle for a given event. Contains the functions:

- GetPosition()
- GetXPosition()
- GetYPosition()
- GetZPosition()
- GetID()
- ComputePhi() (see Section 2.4)

2.1.2 VertexFinder

2.1.3 EventPlotting

2.1.4 MCInteraction

2.2 Selecting a candidate vertex

First, only hits within the tracker are used. This avoids complications with clustered hits within the calorimeter, where hit positions have poor spatial resolution. With only tracker hits selected, a search for hits is started at the top-most layer of the detector. Iterating through each subsequent layer down the detector, general patterns are looked for.

2.2.1 Single hit first interaction

The first pattern is for a pair event whose first interaction in the tracker is a single hit. In this case the following is done:

1. A single hit is identified in a layer.
2. The next N layers are searched through. If two or more hits are identified in one of these layers, a candidate pair event is declared.

The value of N is set by the variable `SearchLayers` in the script and is currently set to be 5 but can be easily changed by the user in the line:

```
SearchLayers = 5
```

Allowing for N to be larger than 1 is particularly useful at high energies, where the opening angle of the electron-positron pair is likely to be small, resulting in the electron and positron hitting in the same pixel, registering only one hit. This may happen for multiple layers.

Layers within the detector are passed until the algorithm finds an event that meets these criteria. This is implemented with the **FindVertices** function in the **VertexFinder** class. When these criteria are met, the program declares that a "candidate vertex" exists, meaning that the basic requirements are met and that there is more analysis that must be done to determine if it is reasonably accurate to define said hit as a vertex. The final step for this vertex declaration is outlined in Section 2.3.1.

2.2.2 Double hit first interaction

If the event does not meet the previous criteria, it is passed to a different reconstruction algorithm. The requirements for this are as follows:

1. The first layer that contains hits has exactly two hits.

2. The layer immediately following that has exactly two hits.

These requirements exist due to the large amount of events that convert in passive material in the detector. At lower energies, when a gamma ray converts in dead material and has a large opening angle, the first layer in which the event has an interaction may contain two hits. The previous logic would throw out this event since it does not meet the one hit in the first interaction layer requirement.

When these two criteria are met, a candidate vertex is declared. The algorithm to reconstruct the vertex for this type of pair event interaction is outlined in Section 2.3.2.

2.3 Declaring a vertex

For each candidate vertex returned by the requirements outlined in Section 2.2.1, an additional requirement is set in order to exclude events where there is a single hit (not attributed to a pair event) and at least two hits in the layer below from some other interaction not related to the single hit. This requirement is dynamic, allowing for more accurate determination for each event, and is outlined in the rest of this subsection (2.3.1).

2.3.1 Dynamic vertex restrictions for single hit candidate vertex

Since actual observations of gamma-ray sources will be well-localized (we will have information regarding where the source is in reference to the instrument), we can say that we know the direction of the incident gamma-ray relative to the detector. In particular, a spherical coordinate system is used with the detector at the origin and its "detector-axis" (or on-axis reference) along the z-axis, where θ and ϕ are the relevant variables that must be known to proceed with this selection process (Fig. 1).

Given the θ and ϕ values, they can be converted into Cartesian coordinates via the following simple translations (where θ and ϕ are measured in radians):

$$x = \sin\theta\cos\phi \tag{1}$$

$$y = \sin\theta\sin\phi \tag{2}$$

$$z = \cos\theta \tag{3}$$

In the script, θ and ϕ are taken as command line inputs in degrees (see Section 2.6 for command line options and default parameters) and the code to translate the inputs into Cartesian coordinates appears as follows:

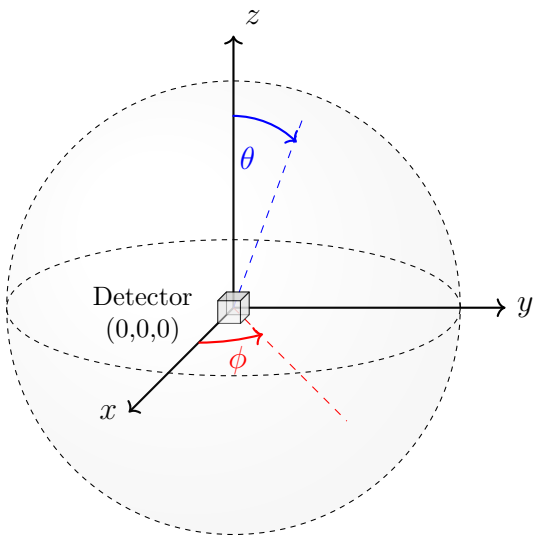


Figure 1: Schematic of the spherical coordinate system used with the detector location at the origin. The z-axis is defined to be the detector axis.

```
theta_rad = np.deg2rad(theta)
phi_rad = np.deg2rad(phi)

x = np.sin(theta_rad) * np.cos(phi_rad)
y = np.sin(theta_rad) * np.sin(phi_rad)
z = np.cos(theta_rad)
```

These are then used to declare the direction $\vec{d}$ of the incident gamma-ray to be:

$$\vec{d} = (x\hat{i}, y\hat{j}, z\hat{k}) \tag{4}$$

and then this is normalized:

$$\hat{d} = \frac{\vec{d}}{|\vec{d}|} \tag{5}$$

In the script this is implemented as follows:

```
init_dir = np.array([x,y,z])/np.linalg.norm(np.array([x, y, z]))
```

Given the normalized direction ray `init_dir` ($\hat{d}$), a virtual point can be projected onto the layer below for each candidate vertex as shown in Fig. 2 below.
To do this, a line was parameterized with the following form:

$$\vec{r}(t) = \vec{r}_0 + \hat{d}t \tag{6}$$

where $\vec{r}(t)$ is the projected line, $\vec{r}_0$ is the initial position, and t is some parameter. Since the interest is locating the virtual point in the layer below, this can be considered for strictly

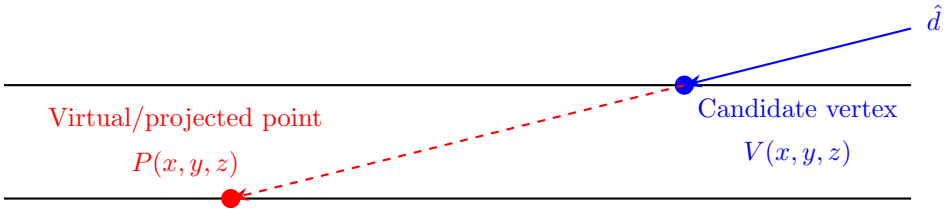


Figure 2: Schematic showing how the virtual/projected point is defined.

the z components:

$$z_{\text{target}} = z_0 + t d_z \tag{7}$$

where z_{target} is the z-coordinate of the layer below the candidate vertex, z_0 is the z-coordinate of the layer in which the candidate vertex is identified, d_z is the z component of the normalized gamma-ray direction vector $\hat{d}$, and t is some parameter. This allows for t to be solved for:

$$t = \frac{z_{\text{target}} - z_0}{d_z} \tag{8}$$

which can then be used to project a point P onto the z_{target} layer:

$$P(x, y, z) = V(x, y, z) + \frac{z_{\text{target}} - z_0}{d_z} \hat{d}(x, y, z) \tag{9}$$

where $V(x, y, z)$ is the position of the vertex (defined as `init_pos` in the code). The code that gets the vertex position and performs the projection is:

```
init_pos = np.array([
    candidate.GetPosition().X(),
    candidate.GetPosition().Y(),
    candidate.GetPosition().Z()
])

def project_to_layer(init_pos, init_dir, z_target):
    t = (z_target - init_pos[2]) / init_dir[2]
    projected_point = init_pos + t * init_dir
    return projected_point
```

This projected/virtual point is a reasonable estimate for a location where the pair should be. Considering this, a hard distance cutoff can be made, demanding that any hits at a distance greater than 5 cm from the virtual point in all directions are not considered related to the candidate vertex and are therefore excluded. The implementation of this looks like a sphere of radius 5 cm with the projected point located at the center.

Within this sphere, a search is done. For all hits inside, the distance from the hit to the virtual point is computed:

$$D = \sqrt{\sum_i (h_i - P_i)^2} \tag{10}$$

where h_i is the i^{th} component of the hit position (i can be x, y, or z) and P_i is the i^{th} component of the projected point. Implemented in the program:

```
def distance_to_virtual(position, virtual_point):
    distance = np.linalg.norm(position - virtual_point)
    return distance
```

Then, all of the distances are sorted in increasing order and the two smallest values are selected. The hits attributed to these distances are chosen to be the electron and positron associated with the candidate vertex:

```
def select_two_closest_hits(hits, projected_point):
    distance_cutoff = 5  # cm
    hits_with_dist = []

    for hit in hits:
        pos = np.array([
            hit.GetPosition().X(),
            hit.GetPosition().Y(),
            hit.GetPosition().Z()
        ])
        dist = distance_to_virtual(pos, projected_point)
        if dist <= distance_cutoff:
            hits_with_dist.append((hit, dist))

    # Require at least two hits within cutoff
    if len(hits_with_dist) < 2:
        return None, None, []

    # Sort by distance and take the two closest
    sorted_hits = sorted(hits_with_dist, key=lambda x: x[1])
    hit1, hit2 = sorted_hits[0][0], sorted_hits[1][0]
    filtered_hits = [hit1, hit2]

    return hit1, hit2, filtered_hits
```

When this is done, the candidate vertex is designated as an "identified vertex" and the two closest hits are declared to be the associated electron and positron.

2.3.2 Tracing pair tracks to reconstruct two hit interaction events

The following is designed to reconstruct events that meet the criteria outlined in Section 2.2.2. It is arguably more difficult to handle these events since the photon conversion does not happen in active material and therefore, the vertex cannot be attributed to an identified tracker hit. This demands that the vertex be attributed to a "virtual hit" that is reconstructed using the electron and positron tracks.

The first step in this reconstruction is extracting the z-position of the topmost layer with hits:

```
shallowest_z = max(rese.GetPosition().Z() for rese in RESEs)
```

Then, the hits in the layer immediately following that must be correctly assigned to the hits in the first layer (eg. electron to the electron and positron to the positron). It is assumed that this is some real event where there is no access to Monte Carlo information. Therefore, distances are used to assign these hits as accurately as possible. To do this, the hits are defined as:

```
A1 = np.array([hit1lay1.GetPosition().X(),
    hit1lay1.GetPosition().Y(),
    hit1lay1.GetPosition().Z()])

A2 = np.array([hit2lay1.GetPosition().X(),
    hit2lay1.GetPosition().Y(),
    hit2lay1.GetPosition().Z()])

B1 = np.array([hit1lay2.GetPosition().X(),
    hit1lay2.GetPosition().Y(),
    hit1lay2.GetPosition().Z()])

B2 = np.array([hit2lay2.GetPosition().X(),
    hit2lay2.GetPosition().Y(),
    hit2lay2.GetPosition().Z()])
```

And the distances for all the possible pairings are defined in the function `best_hit_pairing` as:

```
d11 = np.linalg.norm(A1-B1)
d12 = np.linalg.norm(A1-B2)
d21 = np.linalg.norm(A2-B1)
d22 = np.linalg.norm(A2-B2)
```

such that the sums of the distances for the possible pairings are

```
pairing1 = d11+d22
pairing2 = d12+d21
```

In order to select the pairing that is most likely the correct one, the pairing with the smallest distance sum is selected:

```
if pairing1 <= pairing2:
    return (A1, B1),(A2, B2)
else:
    return (A1, B2),(A2, B1)
```

With the hits associated, the line that passes through each pair can be used to determine the location where the vertex (likely) is. Due to multiple scattering, the lines may not intersect

in 3D, making reconstruction more difficult. Therefore, the approach to determining the vertex location is to connect points between the lines, minimize that connection length, and then use the midpoint of that length to be the reconstructed vertex. The first step is defining the vectors for the two tracks:

$$\vec{v}_1 = q_1 - p_1$$
$$\vec{v}_2 = q_2 - p_2$$

where the points with "q" in the naming are in layer 2 and the points with p" in the naming are points in layer 1 and the numerical subscript indicates what track they are associated with (track 1 or track 2). This is implemented as:

```
# p = point in layer 1, q = point in layer 2
(p1, q1), (p2, q2) = track1, track2

v1 = q1-p1
v2 = q2-p2
```

The vectors are each normalized as:

```
v1 = v1/np.linalg.norm(v1)
v2 = v2/np.linalg.norm(v2)
```

Now, the distance between a point on track 1 and a point on track 2 in the same layer (same z-position value) is defined as $\vec{w}_0 = p_1 - p_2$ and is what must be minimized. To do this minimization, the track lines are parametrized as:

$$\text{TRACK 1: } r_1(t) = p_1 + t\, v_1$$
$$\text{TRACK 2: } r_2(s) = p_2 + s\, v_2$$

And the function to be minimized is:

$$D^2 = |r_1(t) - r_2(s)|^2$$

Minimizing gives the following:

$$t = \frac{be - cd}{ac - b^2}$$
$$s = \frac{ae - bd}{ac - b^2}$$

where

$$a = \hat{v}_1 \cdot \hat{v}_1$$
$$b = \hat{v}_1 \cdot \hat{v}_2$$
$$c = \hat{v}_2 \cdot \hat{v}_2$$
$$d = \hat{v}_1 \cdot \vec{w}_0$$
$$e = \hat{v}_2 \cdot \vec{w}_0$$

They share a common denominator of

$$ac - b^2 = (\hat{v}_1 \cdot \hat{v}_1)(\hat{v}_2 \cdot \hat{v}_2) - (\hat{v}_1 \cdot \hat{v}_2)^2$$

Which must be zero if the tracks are parallel and will therefore not return a vertex location. Finally, the vertex can be assigned as the midpoint m that arises from this minimization of t and s:

$$\begin{aligned} c_1 &= p_1 + t\hat{v}_1 \\ c_2 &= p_2 + s\hat{v}_2 \\ m &= \frac{c_1 + c_2}{2} \end{aligned}$$

This is implemented in the script as:

```
def calculating_vertex_position(self, p1, v1, p2, v2):
    v1 = v1/np.linalg.norm(v1)
    v2 = v2/np.linalg.norm(v2)

    w0 = p1 - p2
    a = np.dot(v1, v1)
    b = np.dot(v1, v2)
    c = np.dot(v2, v2)
    d = np.dot(v1, w0)
    e = np.dot(v2, w0)

    denom = a*c - b*b
    if abs(denom) < 1e-6:
        return None

    t = (b*e - c*d)/denom
    s = (a*e - b*d)/denom

    pca1 = p1 + t*v1
    pca2 = p2 + s*v2

    return 0.5 * (pca1 + pca2)
```

with the vertex assigned as

```
vtx = Vertex(rese=None, Geometry=self.Geometry, AllRESEs=[
   hit1lay1, hit2lay1, hit1lay2, hit2lay2], position=
   vertex_point)
Vertices.append(vtx)
```

2.3.3 Multiple vertices for one event

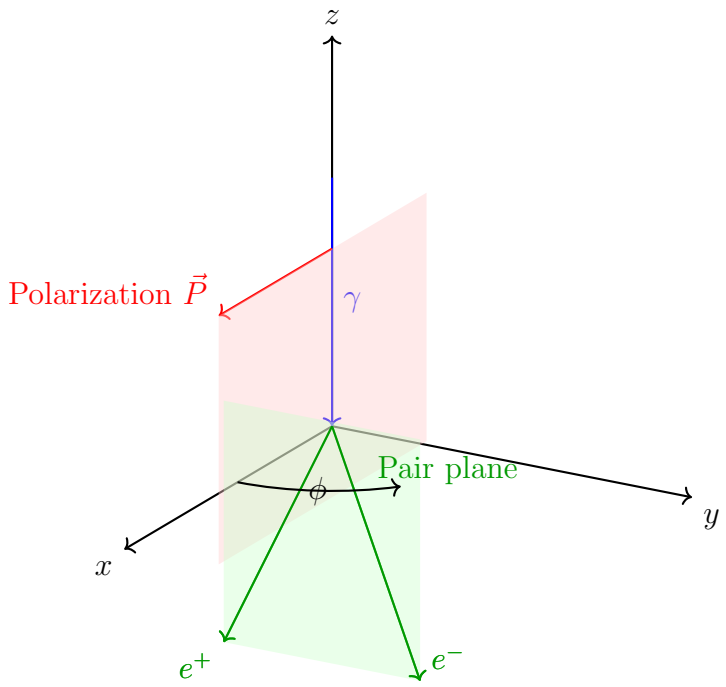
It is possible that an event may have multiple identified vertices. For this reason, the top-most vertex is used as the vertex for the pair event. This is a reasonable selection to make

since the events of interest are pair events where the gamma ray converts into an electron and positron, not pair production resulting from subsequent tracker interactions. Therefore, the topmost identified vertex is a reasonable requirement for this analysis. The function to do this is implemented in the VertexFinder class. It looks at all vertices, sorts by z-position in the tracker, and uses only the vertex with the highest value (the top of the tracker is set at $z = 0$):

```
def TopVertex(self, vertex_list):
    # Select the single vertex for any given event to be that at
        the "top" (largest Z position value)
    return max(vertex_list, key=lambda v: v.GetZPosition())
```

2.4 Computing the azimuthal angle

After a vertex is identified, all of the required information is available to compute the azimuthal angle φ. The first step to do so is to use the two returned adjacent hits (`hit1` and `hit2`) from the `select_two_closest_hits` function. Since these hits represent the electron and positron (the distinction between the two is not relevant for this analysis), their information can be used to compute φ. This is done within the "Vertex" class (see Section 2.1.1) in *VertexIDAndPlotting.py*.



First, a reference direction must be specified. This is defined to be the reference direction that was input for the cosima SIM file generation. Currently the script supports inputs of `RelativeX`, `RelativeY`, and `RelativeZ` only. The code then maps this input string to the relevant unit vector as follows:

```
ref_map = {
    'RelativeX': np.array([1., 0., 0.]),
    'RelativeY': np.array([0., 1., 0.]),
    'RelativeZ': np.array([0., 0., 1.]),
}
```

This information is stored for later use. Then, the positions of `hit1`, `hit2`, and the identified vertex position are extracted:

```
pos1 = np.array([
    hit1.GetPosition().X(),
    hit1.GetPosition().Y(),
    hit1.GetPosition().Z()
])
pos2 = np.array([
    hit2.GetPosition().X(),
    hit2.GetPosition().Y(),
    hit2.GetPosition().Z()
])

vertex_position = self.GetPosition()
```

The directions of the electron and positron from the vertex/conversion point are determined by simply using the positions $h_{j,i}$ where j is either 1 or 2 (corresponding to `hit1` or `hit2`) and i is the component of the position (x, y, or z) and V_i:

$$\vec{r}_j = (h_{j,x} - V_x, h_{j,y} - V_y, h_{j,z} - V_z) \tag{11}$$

These two vectors could then be normalized:

$$\hat{r}_j = \frac{\vec{r}_j}{|\vec{r}_j|} \tag{12}$$

In the script this is done with the following:

```
# Electron and positron direction
d1Dir = pos1 - vertex_position
d2Dir = pos2 - vertex_position

# Normalizing the direction vectors
d1Dir /= np.linalg.norm(d1Dir)
d2Dir /= np.linalg.norm(d2Dir)
```

After this, the initial direction vector `init_dir` $\hat{d}$ (as previously defined in Section 2.3.1) is used to remove the component of the reference direction (`ref_dir` $\hat{f}$) along the initial direction. The mathematical construction for this is Gram-Schmidt orthogonalization:

$$\vec{f}_{lab} = \hat{f} - (\hat{f} \cdot \hat{d})\hat{d} \tag{13}$$

and this vector must (of course) be normalized:

$$\hat{f}_{lab} = \frac{\vec{f}_{lab}}{|\vec{f}_{lab}|} \tag{14}$$

This final vector is a unit vector in the plane perpendicular to `init_dir` $\hat{d}$. In the script:

```
refDir_lab = refDir - np.dot(refDir, init_dir) * init_dir
refDir_lab /= np.linalg.norm(refDir_lab)
```

Then, a second reference vector orthogonal to both $\hat{d}$ and $\hat{f}_{lab}$ can be created:

$$\vec{f}_2 = \hat{d} \times \hat{f}_{lab} \tag{15}$$

and normalizing:

$$\hat{f}_2 = \frac{\vec{f}_2}{|\vec{f}_2|} \tag{16}$$

The code implementation follows logically:

```
ref2Dir = np.cross(init_dir, refDir_lab)
ref2Dir /= np.linalg.norm(ref2Dir)
```

Finally, to compute φ, we acknowledge that the

$$\varphi_j = tan^{-1}(\frac{\hat{f}_2 \cdot \hat{r}_j}{\hat{f}_{lab} \cdot \hat{r}_j}) + 2\pi \tag{17}$$

2.5 Vertex analysis

2.5.1 Event visualization

2.5.2 Plotting residuals

2.6 Running the vertex identification script

The script containing all of the analysis described in the previous parts of this section is titled *VertexIDAndPlotting.py*. The use in the command line (with no extra options set) is:

```
python3  <NAME OF FILTERING FILE>  <NAME OF INPUT SIM FILE>
```

This script is very involved and has the option to set things in the command line:

`inputfile` : The input file to compute azimuthal angles for. Should only contain pair events.

`--event-number` : The number of the event to be plotted. This is NOT the event ID.

`--eventID` : The ID of the event to be plotted. This is NOT the event number.

`--layers` : The desired number of layers to classify a vertex. The default is 2.

`--plot-histogram` : Plot a histogram of the number of vertices per event.

`--plot-residuals` : Plot the residuals between the MC vertex and the reconstructed vertex.

`--plot-events` : Plot individual event visualizations for all events.

`--ref-dir` : Reference direction used for azimuthal angle calculation - can choose RelativeX, RelativeY, or RelativeZ (default is RelativeX).

`--theta` : Polar angle θ (in degrees) of the incoming gamma-ray (default is 0 degrees).

`--phi` : Polar angle ϕ (in degrees) of the incoming gamma-ray (default is 0 degrees).

`--plot-distance-dist` : Plot distance distribution of hits relative to virtual hit in the layer below vertex.

2.7 The main function

3 Modulation analysis

For pair production, the electron-positron pair is preferentially emitted in the plane containing the polarization vector. This means that, in reference to the polarization vector, a majority of azimuthal angles will occur when φ is 0 or π with minima at $\pi/2$ and $3\pi/2$. This distribution pattern follows a sinusoidal modulation, given by the following:

$$\frac{N}{2\pi}\Big[1 - A\cos(2(\varphi - \varphi_0)\Big] \tag{18}$$

In order to remove any instrumental effects, Eqn.18 is fit to the ratio of polarized to unpolarized calculated azimuthal angles. This is implemented in the *ModulationFit.py* script which is used in the command line as:

```
python3 ModulationFit.py --polarized=<TEXT FILE CONTAINING POLARIZED
AZIMUTHAL ANGLES> --unpolarized=<TEXT FILE CONTAINING UNPOLARIZED
AZIMUTHAL ANGLES>
```

3.1 Implementation

To perform this analysis, first a function `GetPhiValues` is defined to extract the azimuthal angles from the input text file. This function is then used on both the polarized and unpo-

larized azimuthal angle files to store the azimuthal angles. These are then binned and put into a histogram with the following parameters where Poisson errors are considered:

```
bins = np.linspace(-np.pi, np.pi, 17)
x = 0.5 * (bins[:-1] + bins[1:])

h_pol, _ = np.histogram(phi_polarized, bins=bins)
h_unpol, _ = np.histogram(phi_unpolarized, bins=bins)

# Adding poisson errors and propagating into the ratio
ratio = h_pol/ h_unpol
error = ratio * np.sqrt(1/h_pol + 1/h_unpol)
```

After this, the histogrammed values are fit with a sine curve modeled by Eqn.18 (defined as `polarfit` in the code):

```
def polarfit(x, A, phi0, N):
return N / (2 * np.pi) * (1 - A * np.cos(2 * (x - phi0)))
```

This plot is displayed and the fit parameters are printed out in the terminal.

4 Material Analysis

When a pair conversion occurs, it is not restricted to happening in the silicon detectors. For this reason, it may be of interest to know how many pair conversions happen in different materials within a given instrument.

4.1 Identifying the material for a given pair conversion

The *HitsInMaterial.py* script is able to take information about the MC location of a vertex (pair conversion) and map that location to the corresponding location in the associated geometry file. When it finds that point, it extracts information about the material and associates that material with the event ID for that vertex. To run this script the command line usage is:

```
python3 HitsInMaterial.py <NAME OF SIM FILE CONTAINING ONLY PAIR EVENTS>
```

This will output a list in the terminal with all the materials in which conversions occurred and the number of conversions in each material. Additionally, it will output a text file with the naming convention

<NAME OF INPUT SIM FILE>_MaterialInfo.txt

which contains a column of event IDs and a column of materials.

4.1.1 File structure

First, the input file and output file are opened. For each pair event, the position of the IA (interaction) point is extracted and then that position in the geometry is found and the material information is extracted. This mapping from IA position to material is done via the following code:

```
pos = IA.GetPosition()
volume = Geometry.GetVolume(pos)

if volume:
    mat = volume.GetMaterial()
```

For each IA, the event ID and material associated with the location of pair conversion are printed into an output text file.

4.2 Computing azimuthal angles in a material

After identifying the material associated with each pair conversion, the script *MaterialVertexAnalysis.py* can compute the azimuthal angles for the vertices identified in a chosen material. The desired material can be entered in the command line as follows:

```
python3 MaterialVertexAnalysis.py input_file <MATERIAL INFO TEXT FILE>
sim_file <ORIGINAL SIM FILE> material <CHOSEN MATERIAL>
```

Running this will write an output text file containing all of the azimuthal angles computed. The naming convention of this file is:

<NAME OF MATERIAL INFO FILE>_<NAME OF CHOSEN MATERIAL>.txt

4.2.1 Referencing previous azimuthal angle calculation

The file *VertexIDAndPlotting.py* discussed in Section 2 already establishes a process for calculating the azimuthal angle for a given event. Given the structuring of that file into classes, it is easy to call the necessary classes and associated functions into a new file to do vertex identification and azimuthal angle calculation. In *MaterialVertexAnalysis.py*, the following is called:

```
from VertexIDAndPlotting import VertexFinder
```

Refer to Section 2.1.2 for detailed information about the `VertexFinder` class. After the input file and original SIM file are read in, the event IDs are grouped by material. This is implemented by the following:

```
event_ids_by_material = defaultdict(set)
with open(input_file, 'r') as f:
    for line in f:
        line = line.strip()
```

```
            if not line or line.startswith('#'):
                continue
            parts = line.split()
            if len(parts) < 2:
                continue
            try:
                eid = int(parts[0])
            except ValueError:
                continue
            material = parts[1]
            event_ids_by_material[material].add(eid)
```

Here, `event_ids_by_material` is a dictionary where each key corresponds to a material identified in the geometry, and the associated value is a set of event IDs belonging to that material. Then, the chosen material (from the command line) is assigned the title `target_material` and the dictionary is searched through to find the event IDs (designated to be `target_eids`) corresponding to that material:

```
target_eids = event_ids_by_material.get(target_material, set())
```

Then, the `VertexFinder` class is initialized:

```
vertex_finder = VertexFinder(Geometry, NumberOfLayers=2)
```

Events are then read through, with the event ID of each begin extracted and compared to the list of `target_eids`. If a given event ID is not in this list, it is skipped:

```
eid = RE.GetEventID()

# Only process events that belong to the target material
if eid not in target_eids:
    continue
```

For the remaining events, detector noise and clustering algorithms are applied. The `vertex_finder` is used on these clustered events

```
# Cluster the event
REI = M.MRawEventIncarnations()
M.SetOwnership(REI, True)
REI.SetInitialRawEvent(RE)
Clusterizer.Analyze(REI)
ClusteredRE = REI.GetInitialRawEvent()
M.SetOwnership(ClusteredRE, True)

# Find vertices
vertices = vertex_finder.FindVertices(ClusteredRE, theta=0.0,
   phi=0.0)
```

where, currently, the theta and phi values are hard-coded to correspond to only on-axis simulations. For each vertex found, the azimuthal angle is computed from the function in the original `VertexFinder` class:

```
for vtx in vertices:
    hit1, hit2 = vtx.AllRESEs
    phi_val = vtx.ComputePhi(theta=0.0, phi=0.0, hit1=hit1, hit2
       =hit2, ref_direction="RelativeX")
    results[target_material]["event_ids"].append(eid)
    results[target_material]["phi"].append(phi_val)
```

These azimuthal angles are then saved to the output text file.

5 Pipeline review

To run a full analysis from selecting pair events to extracting azimuthal angles and fitting the modulation function to their distribution, the scripts are run in the following order:

1. Pair event filtering: *PairEventSelection.py* or *PairEventSelection_MultipleFiles.py*
2. Vertex identification: *VertexIDAndPlotting.py*
3. Modulation curve fitting: *ModulationFit.py*

If material analysis is also desired, step 1 is done and then:

2. Identifying material for conversion: *HitsInMaterial.py*
3. Computing azimuthal angles: *MaterialVertexAnalysis.py*
4. Modulation curve fitting: *ModulationFit.py*

References

[1] Zoglauer, A. *Cosima Documentation*. `https://github.com/zoglauer/megalib/blob/main/doc/Cosima.pdf`