Compton Chain Algorithm

Kepler Domurat-Sousa

August 4, 2025

1 Overview of the Code

This code takes in a .phsp tuple file from TOPAS and attempts to determine the first scatter in each Compton chain. From the determined first scatters it then creates Lines-of-Response (LORs) that can be combined to form a complete PET image. The code has three major parts: input of tuple data, determination of first scatters, and output of LORs.

The code loops over the three parts, loading in and operating on one history at a time. There is also a small amount of code that runs at startup. This startup code opens the input and output files, records the allowed Figure-of-Merit (FOM) per scatter, and makes any run-time parameter changes.

2 Abbreviation Names in this Note

This note requires having a common name for certain aspects of the physics, geometry, major aspects of the code, and more. In cases where it applies just for a specific function the variable names will be defined in the description of the function. More general terms are defined here:

- LOR: a Line-of-Response (LOR) is the path from the first interaction of γ_1 to the first interaction of γ_2. The annihilation point lies upon this line, unless an IPS occured
- $T_1, T_2, ...T_i$: The true ordering of the scatters. This is always truth information based, and is not used in the creation of LORs.
- $R_1, R_2, ...R_i$: The reconstructed ordering of the scatters based on the search. This is not based on truth information and is the basis of the created LORs (from R_1 of one chain to R_1 of the other chain)
- FOM: The Figure-of-Merit (FOM) for the search tree. This tells how good the reconstruction finds the path it tested to be. Two versions of this are often discussed: the FOM which is the total value for a search, and the FOM/scatter which is used to set a FOM cutoff based on how large the searched chain is.

- IPS: An In-Patient-Scatter (IPS) is a history where one or more of the gammas scattered inside of the phantom. This prevents us from determining a good line of response.
- TOF: The Time-of-Flight (TOF) is the time from the start of a history. This is used for the calculation of the true center of the LOR (rather than the geometric center).

3 Startup Arguments

The code takes three setup arguments on all runs. These arguments can then be followed by flag arguments to give additional non-default behavior. Additional changes to code behavior can be done by changing the default values, as described in section 4.

3.1 Standard Arguments

The code has a specific first three arguments. The first argument is the .phsp file containing the tuple information to be worked on. It is expected that this file will be from the custom "MyNtupleEnergy" scorer in TOPAS. Files from any other source will lead to undefined behavior.

The second argument is the location and name for the output files. The name should not include a file extension. On running file extensions will be added to the given name, such as .lor, .debug, and .eng, referring respectivly to files contaning LOR data, diagnostic information, and scatter energies.

The third argument is a floating point number of how much FOM per scatter is allowed before hitting a cutoff. This has typically been set to 1.3, however this number could change with modifications to how the FOM is calculated.

3.2 Additional Flags

These are additional flags that can be applied to the running of the code to do non-standard runs. These do not cover all possible settings that the code has, just ones that are able to be changed at run-time rather than compile-time. For the compile time settings see section 4.

These flags are all a dash followed by a letter, such as -b. These flags may then require another argument directly after. This is used in cases where a value must be passed with the flag. For example the resolution changing flags (-E, -s, -t) must be followed by a new resolution.

- -b: This flag tells the code that the .phsp file that it will be loading is in binary format.
- -E: This flag tells the code to change from the default energy resolution to the value directly following the flag. For example: "-E 10" would change the energy resolution to $k_e = 10$ keV/switch.

- -s: This flag tells the code to change from the default time resolution to the value directly following the flag. For example: "-t 0.636" would change the time resolution to 0.636 cm (50 ps FWHM$_t$).
- -t: This flag tells the code to change from the default spatial resolution to the value directly following the flag. For example: "-s 0.032" would change the spatial resolution to $\sigma_x = 0.032$ cm.
- -h or -H: This flag tells the code to display the built in help information.
- -d: This flag tells the code to disable the time randomness applied to scatters. This greatly simplifies some debugging methods, allowing easy reading of the center of the predicted LORs.

4 Default Values

A group of default values for the simulation are set at the top of the .c file. These values define things from how close two floating point values must be to be considered equal ("#define ENG_RNG 0.001") to if output LORs should be grouped by truth information.

The default parameters include the chosen default resolution. These are "double time_uncert_cm = 6.36;" giving a default time resolution of 500 ps FWHM$_t$, or $\sigma_t = 212$ ps. Note that this time uncertainty is given in cm, not units of seconds. The second default parameter is "double spc_uncert = 0.1;" giving the one $\sigma_x = 0.1$ cm as the spatial resolution. The third and final default parameter sets the energy resolution: "double E_per_switch = 1.0;". This gives the energy uncertainty in units of keV/switch, giving the number of expected count of switched dye molecules by energy deposited as $k_e = 1$ keV/switch.

4.1 Physical Constants

Three physical constants are defined: the mass of an electron (in keV), the speed of light (in cm/ns), and the value of π. These allow easy access for calculations involving the constants, and localize any transcription errors.

4.2 Search Settings

Many search settings are set in this area and cannot be changed at run-time. The choice to make these only changeable at compile time was for simplicity and allowing the compiler make additional optimizations based on these constants. The following are search settings:

- "#define LARGEST 10" This tells the search function to only keep the brightest 10 scatters from a chain. Although this can usually be increased without the code slowing down majorly, occasionally there can be chains with many scatters that will cause a sudden slowdown. There may be other good solutions to speeding up those instances.

- "#define SKIP 0" This defines how many scatters from the end of a chain the code should skip. The code has generally been run with this at 0. Due to the increasing value of $\delta E_{incoming}$ as the search proceeds down a chain, the final few scatters do not typically cause good reconstructions to fail.
- "#define KEEP_SINGLES 1" This Boolean tells the reconstruction code to keep chains with just a single scatter. In such cases that individual scatter will be treated as R_1 for creating the LOR.
- "#define MAX_SINGLE_SIGMA 3.0" A limit on how much FOM a single step in the reconstruction algorithm can accumulate. If the increase exceeds this all branches from the step are pruned.
- "#define MIN_SCAT_ENG 10.0" The minimum energy (after noise-corruption) for a scatter to exist in the scatters list.
- "#define E_TRIGGER 20.0" The energy for the extremely simple trigger setup
- "#define PHI_MODULES 12" The number of modules in ϕ for the extremely simple trigger.
- "#define MODULE_SEPERATION 3" The minimum number of modules seperation needed for the simple triggering.
- "#define NEVER_CUT 0" A Boolean value to tell the reconstruction how to handle not finding a Compton reconstruction that passes the FOM. If set to 1 such a situation leads to the highest energy scattering being taken as R_1.

4.3 Diagnostic Controls

A number of settings for diagnostic information exist. It is not recommened to turn them all on at the same time. Some diagnostic data can be practically recorded for an entire large run of data, allowing for large statistics to understand detector behavior. Other information is highly data rich and produces files that are far to large to handle bigger runs. All of the diagnostic settings follow:

- "#define FIRST_N 5" Sets how many energies from each chain to output to the diagnostic .eng file. This allows for checking the energies of $T_1 ... T_N$ and $R_1 ... R_N$ where $N = 5$ by default.
- "#define READ_DEBUG 0" Was used for diagnostics of the read-in, no longer controls anything
- "#define GENERAL_DEBUG 0" Turns on basic diagnostic information in a large number of functions. This is designed to be used sending stdout to a file for later close reading. This is also not designed for large runs, producing detailed information on what various functions did with every history.

- "#define TREE_DEBUG 0" Proides written information to stdout on the paths taken by the tree search. This is typically run with the general debug on. This information output has largely been superseded by the graphviz debug.

- "#define SCATTER_LIST_DEBUG 0" Prints the two sets of scatters (as they are when passed to the search) to a file with the same name as the file of LORs but with the .scatters extension.

- "#define GRAPHVIZ_DEBUG 0" Prints formatted graphs for use with the graphviz package. These graphs are of each search starting from a guessed first scatter and looking for low FOM solution. The graphs contain the true scatter number T_i, the current FOM, and the best found FOM. Although they can be read from the file, it is recommended that the graphs should be visualized. To do this find the entry in the file for the history and gamma that you are looking for, then run graphviz (command line name "dot") and make the graph into a format that is easy to read. I have used the structure: "dot [filename] -tpdf > my_tree.pdf" to make a PDF of the tree.

- "#define LOR_GROUP 0" This sets how the LORs will be output. If set to 1 the output will be split into a .true and a .misID file. These files will contain all lors with $T_1 = R_1$ and $T_{i\neq 1} = R_1$ respectively. If this is set to true the .lor file will be empty

- "#define CUT_IPS 0" A flag to set the .misID file to not have any IPSs included. IPSs will not be recorded in the .true file if this flag is true or false as by definition an IPS does not have a first scatter in the detector.

5 Custom Structures

Three custom structures are defined in truth_assign.h: "event", "scatter", and "scatter_truth".

5.1 "event"

The "event" structure is setup to contain the full information from a single line of the .phsp input file. This information is a record of a single interaction in Geant4, containing the history number, particle energy (in keV), energy deposited in the step (in keV), location of the interaction (vector in cm), time of flight (ns), type of particle, and particle ID in the history. The "event" structure can also contain a short string of what sort of interaction created the particle. This is not currently used due to the setup of the scorer in TOPAS. The particle type is recorded using PDG codes. The event.id value is identifier for an individual particle within the history. It starts at 1 and counts up as new particles are generated.

It should be noted that event.id does not increase monotonically in the .phsp file as Geant4 follows a first-in-last-out stack pattern for running individual particles.

5.2 “scatter”

This structure contains all of the information needed for the search algorithm to run. Upon creation parameterized randomization is applied to determine what the detector recorded, giving a group of scatters. As such this should not contain true values. Truth information is carried by a pointer to a “scatter_truth” structure.

5.2.1 “scatter_truth”

This is the structure containing the truth information with scatters. It can safely be removed without breaking the search process. All diagnostic information about the search that requires the truth information uses the data contained in this structure.

6 Scatter Handling Functions

The following functions are used for basic tasks involving “scatter” structures. These tasks include creation, copying and freeing.

- scatter* new_scatter_old(vec3d* vector, double deposited, double time): Creates a scatter using just the location, deposited energy, and TOF. Generally should not be used, just exists to keep test_expected_energy() functioning as expected.
- scatter* new_scatter(vec3d* vector, vec3d* dir, double deposit, double time, double eng_uncert, double space_uncert, double time_uncert): Creates a new scatter with the given location, electron direction, deposited energy, TOF, uncertainty in energy, uncertainty in space, and uncertainty in time.
- scatter* copy_scatter(scatter* a): Copies a given scatter, including making copies of the included location and direction vectors. It does not copy truth information.
- void* delete_scatter(void* in): Frees all contained structures (including any truth information) and then frees the scatter itself. Returns NULL to make this useful with fmap() to clean up a list of scatters.
- int scatter_dep_compare(void* va, void* vb): Takes pointers to two scatters and compares their energies. If the first scatter (a) has a lower energy, 1 is returned. If the two scatters have the same energy 0 is returned. If a has a higher energy -1 is returned. This is used for sorting an array of scatters by energy.
- int partition(void** array, int low, int high, int (*f)(void*, void*)): The quicksort partition function that is used to sort scatters by energy.
- void scatter_quicksort(scatter** arr, int low, int high): The main function for doing a quicksort on an array of scatters.

7 Event Handling Functions

The following functions are used for basic tasks involving "event" structures.

- event* duplicate_event(event* source): Duplicates an event and all of its contents. Returns a pointer to the new copy.
- void* delete_event(void* in): frees an event and all of its components. Returns NULL
- void* print_event(void* in): Prints the contents of an event

8 Reading in Data

Data from the .phsp file is read in line by line. Each line of the file defines an event structure. Additional lines are read until the history number changes. A list of all events in the history can then be passed out to other functions.

The reading of an entire history is done by the llist* load_history(FILE* source, event* (*f)(FILE*)) function. In the inverse_kinematics.c version in the Squires Lab GitHub there is an error with the first history being read by "load_historyb" and all subsequent histories being loaded by "load_history". All histories should be loaded by the latter, and this leads to a bug mangling the second history that is loaded. All subsequent histories behave fine.

8.1 load_history

This function has the form llist* load_history(FILE* source, event* (*f)(FILE*)), taking in a .phsp file (FILE* source) and the function to be used for reading a line (event* (*f)(FILE*)). If the file to be read is binary (the -b flag was used) then read_line_binary is passed, otherwise read_line is used. The two line reading functions are simillar, just using a different reading method.

load_history has a static variable containing a pointer to the previous event (event* previous_event) that was recorded. At the beginning this will be a NULL pointer. In later runs this pointer contains the first entry in the new history. As long as the history number in the previous_event when the function was called matches the read-in history number, read in events are added to the list of events. Once an event has a new history number it is left in previous_event for the next call of the function, and a pointer to the list of events of the loaded history are returned.

9 Trigger Functions

These functions are used to check if a given scatter would have triggered a module, and if triggered, what module.

9.1 double vec_to_phi(vec3d* a)

This function calculates the angle ϕ of the vector a. This is calculated as $arctan(\frac{a_y}{a_x}) = \phi$

9.2 int test_vec_to_phi()

This is a test function for the vec_to_phi() function. This returns the number of passed tests. If a test fails it prints the expected result and calculated result to stderr.

9.3 int phi_trigger(scatter* a, uint modules)

This function first checks to see if the scatter has enough energy to trigger a module. If not 0 is returned. Otherwise it determines the module that the interaction occured in and returns that value.

10 Determining from Truth if an IPS Occured

This code next determines if an IPS occured. This information is needed to correctly provide the diagnostic information on what the T_i value is for each scatter. This information is also used to take actions such as cutting based on IPS if enabled.

10.1 int in_patient(llist* list)

This function sets the list pointer to the start of the history list. It then calls int rec_in_paitent(llist* list, int reject_id) with the list and a given reject_id of -1. The recursive function then looks for the first instance of each gamma in the history. If the first instance has a energy more than 0.1 keV from the mass of an electron then the history is considered to have a IPS.

If the IPS was associated with the first gamma in the list a 1 is returned. If the IPS happened to the second gamma a 2 is returned. If both a 3 is returned. If no IPS occurred then a 0 is returned. These values allow the return value to act as both a Boolean and a bitmask. Return & 0b1 gives if the first gamma had an IPS, return & 0b10 gives if the second gamma had an IPS.

11 double add_quadrature(double a, double b)

Calculates quadrature addition, i.e. return $= \sqrt{a^2 + b^2}$

12 Calculating Compton Kinematics

This group of functions calculates predicted gamma energies based on the geometry and deposited energies of events. All of these functions follow the convention that the gamma went $S_a \rightarrow S_b \rightarrow S_c$ This involves the following functions:

- double expected_uncert_b(double b, double theta, double uncert_b, double uncert_theta): This function calculates the uncertainty in the predicted energy $\delta E_{a\to b}$ of the gamma from $S_a \to S_b$. The involved calculation is more completely described in section 12.1.
- double expected_energy_b(scatter* a, scatter* b, scatter* c, double* uncert): The calculated energy of the gamma from $S_a \to S_b$. This function can also give the uncertainty $\delta E_{a\to b}$ by writing to the double pointed to by "uncert". The equation giving the returned energy is 3, where $E_{deposit}$ is the energy deposited at S_b and $\theta = \angle S_a S_b S_c$
- double expected_energy_a(scatter* a, scatter* b, scatter* c): A variation on expected_energy_b() giving a prediction for the energy of the gamma coming into S_a. This is done by taking the result from expected_energy_b() and adding the deposited energy at S_a. This function is no longer used.
- int test_expected_energy(): This function tests if the expected energy functions are behaving as expected.

12.1 Expected energy for $\gamma_{S_a \to S_b}$

This function determines expected energy of the gamma ray going into scatter location b with the path $S_a \to S_b \to S_c$. The function has the form:

$$g(S_a, S_b, S_c) = E_{a\to b} \pm \delta E_{a\to b} \tag{1}$$

where S_a, S_b, and S_c are scatters with location and energy information, along with their respective uncertainties. $g(S_a, S_b, S_c)$ will use the position information of all three scatters, and the energy deposited information of S_b to find the energy of the gamma ray going $S_a \to S_b$ as determined by the Compton scattering formula. The uncertainty in this predicted energy is also calculated. The locations of S_a, S_b, and S_c will be refered to as $\vec{a}$, $\vec{b}$, and $\vec{c}$ respectively

This calculation is based on the energy deposited in the scattering event S_b and the angle θ between the paths $\vec{ab}$ and $\vec{bc}$. These locations have uncertainties in position of δa, δb, and δc. This gives an uncertainty in θ of approximately

$$\delta\theta = \sqrt{\frac{\delta a^2 + \delta b^2}{\left\|\vec{ab}\right\|^2} + \frac{\delta b^2 + \delta c^2}{\left\|\vec{bc}\right\|^2}} \tag{2}$$

Taking the energy deposited at S_b to be E$_{deposit}$ and the uncertainty in deposited energy as δE_b the following equations can be derived from the Compton Scattering Formula:

$$E_{a\to b} = \frac{E_{deposit} + \sqrt{E_{deposit}^2 + \frac{4E_{deposit} m_e}{1-cos\theta}}}{2} \tag{3}$$

$$\delta E_{a\to b}(\delta\theta) = \frac{m_e E_{deposit} sin\theta}{(1-cos\theta)^2\sqrt{E_{deposit} + \frac{4m_e E_{deposit}}{1-cos\theta}}}\delta\theta \tag{4}$$

$$\delta E_{a\to b}(\delta E_{deposit}) = \frac{1}{2}(\frac{2E_{deposit} + \frac{4m_e}{1-cos\theta}}{2\sqrt{E^2_{deposit} + \frac{4m_e E_{deposit}}{1-cos\theta}}} + 1)\delta E_{deposit} \tag{5}$$

$$\delta E_{a\to b} = \sqrt{\delta E_{a\to b}(\delta\theta)^2 + \delta E_{a\to b}(\delta E_b)^2} \tag{6}$$

13 Recursive search algorithm

The main method of searching the tree of all possible scattering paths is a recursive search of the tree. This is done by the function double recursive_search(double best, double current, double inc_eng, double inc_uncert, scatter* origin, scatter* loc, scatter** remaining, uint remain_count, llist** path). This function matches the following equation:

$$f(\mathcal{F}_{best}, \mathcal{F}_{current}, E_{incoming}, \delta E_{incoming}, S_{i-1}, S_i, \{S_{i+1}...S_{N-1}\}) \tag{7}$$

where $\mathcal{F}_{best}$ is the best figure of merit (FOM) found so far, and $\mathcal{F}_{current}$ is the FOM total for all previous scatters in the searched chain. $E_{incoming} \pm \delta E_{incoming}$ is the expected energy of the gamma from S_{i-1} to S_i. S_i is the location we are checking, having an energy deposited and a location in space. The index i is the current location in the searched chain of scatters of length N.

Equation 7 does not match the function definition perfectly as a handful of additional parameters must be passed for the code. A count of how many remaining scatters is needed to determine where the end of the array of scatters is, along with when to stop if SKIP has been set to a non-zero value. A list is also passed to provide a location to put the list of chosen scatters to provide diagnostic information.

The energy of the outgoing gamma is also calculated based on the deposited energy.

$$E_{outgoing} = E_{incoming} - E_i \tag{8}$$

$$\delta E_{outgoing} = \sqrt{\delta E^2_{incoming} + \delta E^2_i} \tag{9}$$

If the list of further scatters to be searched ($\{S_{i+1}...S_{N-1}\}$) is empty then we cannot continue the search as there are no outbound paths from S_i. In this case we will simply return $\mathcal{F}_{current}$ as the total figure-of-merit of the chain that lead to this situation.

13.1 For $i' = i + 1$, while $i' < N$

We now iterate over the remaining scatter locations that the chain could continue down. In each iteration we will check the quality of the path $S_{i-1} \to S_i \to S_{i'}$, giving a figure of merit value for the step.

We calculate the expected energy of the gamma ray going $\vec{a} \rightarrow \vec{b}$ using equation 1:

$$E_{Compton} \pm \delta E_{Compton} = g(S_{i-1}, S_i, S_{i'}) \tag{10}$$

The difference (E_{error}) between the incoming energy and the energy predicted by the Compton formula is then calculated.

$$E_{error} = |E_{incoming} - E_{Compton}| \tag{11}$$

$$\delta E_{error} = \sqrt{(\delta E_{incoming})^2 + (\delta E_{Compton})^2} \tag{12}$$

$$\mathcal{F}_{step} = \frac{E_{error}}{\delta E_{error}} \tag{13}$$

At this point it is also possible to add additional weights to the figure of merit. A weight for the degree to which the electron's path was out of the plane of the scatter was previously used, however it did not substantially improve the performance of the algorithm and so its weight has been set to zero.

13.1.1 If (($\mathcal{F}_{step} + \mathcal{F}_{current} < \mathcal{F}_{best}$) and ($\mathcal{F}_{step} <$ MAX_SINGLE_STEP))

We check that with the current step to j we have not gotten a worse total $\mathcal{F}$ then the best found, and that the step did not get so much worse in this step that we should not bother continuing along the chain. If either is the case, then we will skip the steps in this subsection.

Otherwise, we calculate $\mathcal{F}_{below}$ to find the figure-of-merit of the best continuing path. The calculation is done using the same recursive function but with S_j removed from the list of remaining scatters.

$$\mathcal{F}_{below} = f(\mathcal{F}_{best}, \mathcal{F}_{step} + \mathcal{F}_{current}, E_{outgoing}, \delta E_{outgoing}, S_i, S_{i'}, \{S\}_{remaining}) \tag{14}$$

If $\mathcal{F}_{below}$ is the best value figure-of-merit we have found so far we will keep that value as $\mathcal{F}_{best}$, otherwise it can be discarded.

13.2 End of the recursive function

After iterating through every scatter in the list of remaining scatter locations the function returns the value $\mathcal{F}_{best}$ that it has found.

14 Search of a Single Sided Tree

Searches of the tree of possible scatters of γ_{near} are constrained by the list of scatters of γ_{far}. Each tried first scatter for γ_{near} has the gamma come in from the location of one of γ_{far}'s scatters, giving a two sided tree of scatters. This prevents the algorithm from choosing a starting path other than a possible LOR. For typical searches where there are multiple gammas in each list the each of the chosen first scatters must have the other as the “origin” location of the initial gamma. This type of restriction is not always wanted, either to allow greater freedom for the search algorithm or to reduce the number of tested branches (due to better restriction of the FOM).

To run this single sided tree the function scatter* multi_gamma_stat_iteration(llist* history_near, llist* history_far, double sigma_per_scatter, int* best_find_array, float* alpha_eng_distro) is used. This function takes in a list of scatters that is to search for the first scatter (history_near), a list of scatters as origin constraints (history_far), a given FOM per scatter, and two arrays for truth information.

If there is only a single entry in the near list then that is returned if KEEP_SINGLES is true.

The two lists of scatters are converted to an array. Then these arrays are sorted by energy, putting interactions with the highest energies at the start. This allows the search to start with the highest energy scatters that have the highest probability of producing good results. The energy ordering also makes removing low energy scatters easy.

The code next limits to just the LARGEST + SKIP highest energy scatters. By default this will be 10.

The limit on FOM must now be calculated, given as $\mathcal{F}_{best}$ = sigma_per_scatter $*$ LARGEST + SKIP

Taking the length of the array of near scatters S_j^{near} to be M and the length of the array of far scatters S_i^{far} to be N we iterate as follows:

14.1 For $j = 0$, while $j < M$

A new array of $S_{\neq j}^{near}$ is made so that it contains all scatters in the list except the currently chosen origin.

14.2 For $i = 0$, while $i < N$

We now call recursive_search (equation 7): $\mathcal{F}_{try} = f(\mathcal{F}_{best}, 0, 511., 0., S_i^{far}, S_j^{near}, S_{\neq j}^{near})$. The returned FOM $\mathcal{F}_{try}$ is the quality of the LOR starting from S_i^{far} and going to S_j^{near}. If $\mathcal{F}_{try} < \mathcal{F}_{best}$ then the current j is kept as the current best solution and $\mathcal{F}_{best}$ is set equal to $\mathcal{F}_{try}$. This makes later searches faster as the reducing $\mathcal{F}_{best}$ limits the depth of search of many paths.

When the iteration is completed the best found scatter is returned. If no paths pass the FOM cutoff then NULL is returned.

15 Search of a Double Sided Tree

To produce a more constrained search a double-sided-tree is used. In these cases the far side scatter that acts as the initial gamma direction for the search of a scatter tree must be the first scatter for the search of the other list of scatters. This means that any found solution goes from the determined R_1 of the far side to the R_1 of the near side.

To choose a pair of search trees that produce the best behavior a test of their combined FOM is used. This is simply the addition of the two FOMs. This best combination FOM is stored in a variable with initial condition $\mathcal{F}_{best} = (\text{sigma\_per\_scatter} * N) + (\text{sigma\_per\_scatter} * M)$, where the length of the array of one set of scatters S^b_j is M and the length of the array of other scatters S^a_i is N.

We then iterate as follows:

15.1 For $j = 0$, while $j < M$

A new array of $S^b_{\neq j}$ is made so that it contains all scatters in the list except the currently chosen origin for list b.

15.2 For $i = 0$, while $i < N$

To have correct search behavior, the starting limit FOM must be reset to $\mathcal{F}^b_{best} = \text{sigma\_per\_scatter} * M$ on each new tree.

The search for the quality of a tree coming from S^a_i and first scattering at S^b_j using recursive search (equation 7): $\mathcal{F}^b_{try} = f(\mathcal{F}^b_{best}, 0, 511., 0., S^a_i, S^b_j, S^b_{\neq j})$

If $\mathcal{F}^a_{best} \geq \mathcal{F}_{best}$ then the search of the other tree can be skipped as the second search cannot produce a result that will be better than the previous best found solution, the first tree did not pass the cutoff.

If from the above it is reasonable to search the second tree we create a new array of scatters $S^a_{\neq i}$ for the search. We then set the limit FOM $\mathcal{F}_{limit}$ to the lower of $\mathcal{F}^a_{best}$ and $\mathcal{F}_{best} - \mathcal{F}^a_{try}$ and then run the search: $\mathcal{F}^b_{try} = f(\mathcal{F}^a_{limit}, 0, 511., 0., S^b_j, S^a_i, S^a_{\neq i})$

If $\mathcal{F}_{best} > \mathcal{F}^a_{try} + \mathcal{F}^b_{try}$ then the S^a_i and S^b_j are kept as the current best two first scatters. If that is the case $\mathcal{F}_{best}$ is set to $\mathcal{F}^a_{try} + \mathcal{F}^b_{try}$ to provide a lower limit on all further searches.

15.3 After iteration

After iteration the function returns the pair of scatters that it found to be best. If it was unable to find a pair then the returned array not allocated and returns NULL instead.

If a list was unable to be searched with the recursive search function (it had a length of 1) then if KEEP_ SINGLES is true the single scatter is returned as one of the returned scatters and a single sided tree search is run to find the other.

16 Creating the Chains of Scatters

The list of scatters for the tree search are made by the build_scatters function. This function takes in a detector history and the ID of the gamma that the scatters are from. It returns the pointer to a list of the scatters.

The function works by finding the first instance of each electron in the list of interactions. It then checks which gamma this electron was associated with by finding the gamma interaction that is closest to this electron. If the ID of this gamma matches that passed to the function then this electron must be added to the list of scatters.

16.1 Creation of the scatter object

After determining that the electron should be added to the list of scatters the location of the electron is copied to be the location of the scatter. An attempt is then made to determine the direction of the electron by checking for the location of the next interaction of the electron. If no such location exists or is in the same location as the original location the direction is not given a vector.

A new scatter is created using the scatter location, scatter direction, deposited energy $E_{deposit}$ as the energy of the electron in this step, TOF as the time of this electron, energy uncertainty as $E_{switch} * \sqrt{E_{deposit}/E_{switch}}$, spatial uncertainty as spc_uncert, and time uncertainty as time_uncert_cm. It is important to note that at this point in the code the data contained in the scatter object is all truth information, without any added noise.

16.2 Additional noise based on detector resolutions

To create a Gaussian distributed noise value repeated addition of random numbers is done. Each random number is uniformly distributed from 0 to 1. By adding these values UNCERT_REP times (default 12) and then subtracting half of UNCERT_REP we get a close approximation of a Gaussian σ deviation. The noise determined as above will be referred to as δ_x, δ_E, and δ_t for space, energy, and time noise respectively.

For spatial resolution the distance variation of δ_x is multiplied by the spatial resolution to give the total miss distance. A random ϕ and θ are then determined, with care taken to make sure that they have correct distribution to give equal density to the entire surface of the unit sphere. A vector is then made in the direction given by ϕ and θ with length 1. This is then multiplied by δ_x times the spatial resolution. This gives the full noise to be applied to the location of the scatter. The scatter location is updated to this addition.

Energy resolution is somewhat more complicated due to modeling of counting fluorescent dye molecules. It is assumed that the count can be determined to a high degree of accuracy, and so the question becomes how many molecules switched. The switching efficiency is set by E_per_switch, here referred to as k_E. This gives the expected number of molecules switched as $N = E_{dep}/k_E$. The uncertainty in this value is $\sqrt{N}$, giving a noise added energy of $E_{noise} = E_{dep} + (\delta_E k_E \sqrt{N})$. This noise adjusted energy is then converted back into a number of switched molecules, rounded to the nearest integer number of molecules, and converted back into an energy. The energy of the scatter is then set to this quantized noise adjusted value. Finally the uncertainty in the energy is set based on $\sqrt{N}$ of this new energy value.

For time resolution the scatter time simply has $\delta_t \cdot \frac{\text{time\_uncert\_cm}}{c}$ added to the true time.

At no point in the Compton chain algorithm is the timing information used, making its behavior independent of whatever timing system is eventually used (apart from however timing information would change the quality of the lists of scatters).

The list of produced scatters is not assumed to be in any particular ordering, however it is likely to be in reverse time order due to the stack-like behavior of Geant4 secondary particle handling.

16.3 find_double_endpoints_stat

This function takes in a detector history and creates the scatter lists, provides a large amount of diagnostic information, and returns the found LOR endpoints. The function has several distinct steps, each handling a different part of this process.

The function first moves to the start of the history and checks to see if there were two gammas in the detector, and if so records the particle ID of each of the two.

Next, the function creates a list of scatters for the first and second gammas. This is done using two calls to the build_scatters function (section 16) with the two different gamma IDs. If either list was unable to be made then the function returns a failure to find the endpoints. If the scatter lists were made, then truth information about them is recorded.

Next, the double_tree_stat_iteration function (section 15) is run to find the first scatter locations of each list of scatters. After some final truth information gathering these two scatters are returned to be used as the endpoints of the LOR.

17 create_lor

This code takes in two scatters and returns the LOR connecting the two. Taking $\vec{a}$ and $\vec{b}$ to be the locations of the two scatters, the following math is done: The geometric center of the LOR is found $\vec{g} = 0.5 * (\vec{a} - \vec{b})$. Next the unit vector for $\vec{b} \rightarrow \vec{a} = \widehat{ba}$ is found. The difference in time of the two scatters is then found and when combined with the unit vector $\widehat{ba}$ gives the offset between the geometric center and the true center of the LOR. This then gets applied to give the true center of the LOR.

Finally, the transverse and longitudinal uncertainty are calculated from the spatial and timing uncertainty of the scatters.

18 Depreciated Code

- llist* load_historyb(FILE* source, event* (*f)(FILE*)): this function is identical to load_history but has a different static variable. This was used for separate loading of the full truth .phsp file and detector information .phsp files.

- uint inside_radius(double distance, double height, event* event): Was used to determine occurance of IPS with full truth information

- vec3d* find_annihilation_point(llist *history): Was used with the full truth information to give a measurement of LOR miss distance.
- double line_to_dot_dist(vec3d* start, vec3d* end, vec3d* point): Used to determine miss distance of LORs using truth information
- scatter* scattering_iterator(llist* scatter_list, double energy_percent): An early version of the Compton search
- scatter* multi_gamma_iterator(llist* history1, llist* history2, double energy_percent): An early version of the Compton search
- scatter* multi_gamma_ele_iterator(llist* history_near, llist* history_far, double energy_percent): An early version of the Compton search
- scatter** find_endpoints(llist* detector_history, double energy_percent): Early function for determining first scatter pair from history list.
- scatter** find_endpoints_2hist(llist* detector_history, double energy_percent): Early function for determining first scatter pair from history list.
- scatter** find_endpoints_ele_dir(llist* detector_history, double energy_percent): Early function for determining first scatter pair from history list.
- scatter** find_endpoints_stat(llist* detector_history, double sigma_per_scatter): Finds the two first scatters independently rather than the current standard of requiring both trees to connect.
- double first_scat_miss_transverse(lor* lor, vec3d* annh_loc): Determines the transverse distance between a LOR and the given annihilation location.
- double first_scat_miss_longitudinal(lor* lor, vec3d* annh_loc): Determines the longitudinal distance between a LOR and the given annihilation location.
- int find_annih_gamma(event* item): No longer does anything for so many reasons!