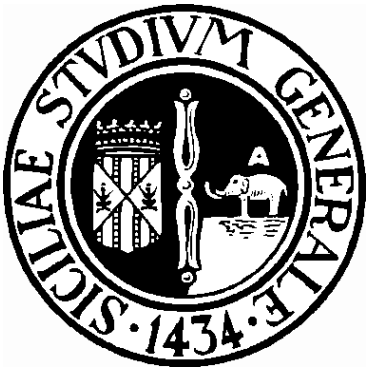
UNIVERSITA' DEGLI STUDI DI CATANIA



FACOLTA' DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

UN SIMULATORE IN C++ BASATO SU GEANT4 PER LO STUDIO DI SENSORI NELLA TOMOGRAFIA AD EMISSIONE DI POSITRONI

Fabio Vincenzo Salamone
Matricola: 616/000604

Relatori:
Chiar.mo Prof. Michele Malgeri
Chiar.mo Dott. Paolo Finocchiaro

Anno accademico 2006/07

Dedicata ai malati di cancro,

affinché sappiano trovare in Dio la forza per resistere

e nella scienza il coraggio per sperare.

Indice

Introduzione

Nel contesto informatico moderno è sempre maggiore l'attenzione che si pone verso l'ambito medico, con l'intento di simulare dispositivi diagnostici per ottimizzarne l'efficienza, oppure per tentare di individuare nuove soluzioni a problemi non ancora completamente risolti. Il ramo della medicina trattato in questo elaborato è quello oncologico che si occupa della diagnosi e della cura dei tumori.

L'obbiettivo di questo elaborato è quello di studiare e verificare, con l'ausilio di mezzi informatici, quali linguaggi di programmazione e toolkit di simulazione, il funzionamento dei sensori di raggi gamma utilizzati nella tomografia ad emissione di positroni (PET, Positron Emission Tomography), cioè una tecnica di medicina nucleare e diagnostica medica che produce immagini tridimensionali e mappe dei processi funzionali all'interno del corpo.

A tal proposito ho sviluppato un simulatore in C++ con l'ausilio del pacchetto Geant4 (GEometry ANd Tracking), ovvero un toolkit orientato agli oggetti per la simulazione del passaggio di particelle attraverso la materia. Geant4 è uno strumento concepito, prodotto e mantenuto secondo metodologie molto avanzate di ingegneria del software, che permettono di accedere e tracciare con trasparenza ogni passo della simulazione. Maggiori dettagli sui principi di funzionamento del Geant4 sono illustrati nel Capitolo 1.

Il lavoro si è suddiviso in diverse fasi. La prima di comprensione e studio del problema, che ha portato alla stesura del Capitolo 2 di questo elaborato, in cui è trattato puramente l'aspetto fisico di interazione delle particelle con la materia.

La seconda fase è consistita nello studio del toolkit Geant4 e nello sviluppo del codice utilizzato per effettuare alcune simulazioni di test, in modo tale da verificare la bontà e l'affidabilità del software prodotto. Questa operazione è stata possibile attraverso il confronto dei risultati ottenuti dalle simulazioni con i risultati sperimentali, disponibili grazie alla collaborazione con i LNS-INFN (Laboratori Nazionali del Sud – Istituto Nazionale di Fisica Nucleare). Maggiori dettagli su software prodotto sono illustrati nel Capitolo 3.

Successivamente sono entrato nel cuore del problema, creando e testando differenti geometrie dei sensori, con diverse combinazioni di materiali, in modo tale da estrarre ed elaborare i risultati delle simulazioni per ottenere un confronto tra ogni geometria e materiale.

Queste simulazioni hanno portato a risultati interessanti in quanto è stato possibile sfruttare le ottime proprietà di estensibilità, flessibilità e trasparenza di un linguaggio di programmazione come il C++, per studiare un problema di natura fisico.

L'ultimo passo è stato quello di simulare un nuovo modello di sensore per testarne il funzionamento e verificare se i risultati ottenuti siano migliori di quelli dei sensori attualmente disponibili. Maggiori informazioni su questo aspetto sono illustrate in dettaglio nel Capitolo 4.

1. Simulazione discreta: Geant4 e programmazione ad oggetti

Le simulazioni giocano un ruolo fondamentale in tutti quei campi in cui è interessante conoscere il funzionamento e la risposta di un dispositivo, ancor prima che esso venga fisicamente realizzato. La simulazione è un processo quasi indispensabile nel caso in cui il dispositivo da simulare è un oggetto di difficile realizzazione o molto costoso, in cui è quindi preferibile conoscerne il comportamento a monte. Attraverso la simulazione è quindi possibile prevederne il funzionamento, modificarne le caratteristiche e continuare a simulare finché non si raggiunge l'obiettivo cercato. Solo dopo una buona simulazione si saprà con certezza che il dispositivo costruito secondo le specifiche imposte sarà in grado di funzionare come desiderato.

A seconda del campo in cui la simulazione deve essere applicata, si utilizzeranno metodi differenti, che permetteranno di ottenere i risultati desiderati. Nel nostro caso si è deciso di utilizzare il toolkit Geant4 per la realizzazione di un software che sia in grado di gestire le interazioni di particelle e raggi gamma con materiali scintillanti, in modo da poter simulare il funzionamento di rivelatori di radiazioni. Esistono altri toolkit di simulazione Monte Carlo che gestiscono le interazioni fisiche, ovvero:

- *Fluka*: impiegato maggiormente per simulazioni di dosimetria.
- *EGS*: è scritto in un linguaggio strutturato chiamato Mortran3 che è stato sviluppato al SLAC da A. James Cook and L. J. Shustek. Il precompilatore

Mortran3 è stato scritto nel 1966 in FORTRAN IV e nel 1977 in FORTRAN 77. Tutte le caratteristiche di EGS erano già state incluse nel Geant3.

- *Detect*: è estramemente semplificato e tutte le sue caratteristiche sono state incluse nel Geant4.

1.1 Il Geant4

Il Geant4 (acronimo di GEometry ANd Tracking, geometria e tracciamento) è un toolkit orientato agli oggetti per la simulazione del passaggio di particelle attraverso la materia. La sua area di applicazione include la fisica delle alte energie e gli esperimenti nucleari, il campo medico, gli acceleratori e gli studi fisici, la biologia e la fisica astro-particellare.

Tutti gli aspetti della simulazione sono inclusi all'interno del toolkit e permettono la gestione:

- della geometria del sistema
- dei materiali implicati
- delle particelle fondamentali di interesse
- della generazione di eventi
- del tracciamento delle particelle attraverso la materia ed i campi magnetici
- dei processi fisici che governano l'interazione con la materia
- delle risposte dei componenti sensibili
- del salvataggio di eventi e tracciati
- della visualizzazione di eventi e traiettorie delle particelle

Il fulcro di Geant4 gira intorno ad un ampio set di modelli fisici per gestire le interazioni delle particelle attraverso la materia con la capacità di gestire un *range* molto vasto di energie.

Geant4 è uno strumento che si differenzia nettamente da strumenti ad esso analoghi per la simulazione Monte Carlo, in quanto è stato concepito, prodotto ed è mantenuto secondo metodologie molto avanzate di ingegneria del software, che permettono di accedere e tracciare con trasparenza ogni step della simulazione.

E' uno strumento sviluppato formalmente presso il CERN, ma di fatto sviluppato da una collaborazione di oltre 140 persone. La release più recente è la 8.3 rilasciata nel Maggio 2007. Si noti che le versioni precedenti fino a Geant3 erano scritte in Fortran.

Geant4 è un insieme di librerie C++ ed è quindi nativamente orientato agli oggetti (OO, Object Oriented). Le versioni più datate, risalenti al 1994, non erano scalabili e quindi non permettevano il semplice inserimento di nuove porzioni di codice o modifiche di quelle già presenti, rendendo molto difficoltose le operazioni di mantenimento.

Il Geant4, invece, include i più svariati strumenti per gestire ogni aspetto della simulazione, dalla geometria dei materiali al tracciamento delle particelle, dalla gestione della risposta dei rivelatori alla visualizzazione grafica ed interfaccia con l'utente. Il disporre di tutti questi strumenti permette di non preoccuparsi del funzionamento a basso livello e quindi riuscire a realizzare applicazioni ad alto livello in cui si riesce ad includere ogni dettaglio fisico.

Geant4, sin dall'inizio del suo progetto, è stato caratterizzato da un corposo lavoro di ingegneria del software che ha permesso la modellizzazione di un sistema orientato agli oggetti in grado di permettere:

- lo sviluppo ed il mantenimento indipendente di ogni porzione di codice

- la decomposizione del problema in un insieme di classi legate unidirezionalmente l'una con l'altra

Inoltre la programmazione OO ha permesso una rapida evoluzione del codice di Geant4 grazie alle proprietà che un buon software orientato agli oggetti possiede:

- estensibilità: è facile aggiungere, modificare ed eliminare porzioni di codice, cosicché anche l'utente può personalizzare a proprio piacimento ogni sezione di Geant4
- flessibilità: è possibile scegliere tra un insieme di possibili soluzioni per risolvere ogni problema della simulazione, spetterà all'utente scegliere quello di maggior interesse per il proprio caso
- trasparenza: è possibile guardare all'interno di ogni singola porzione di codice per comprenderne il funzionamento

Abbiamo definito Geant4 un toolkit perché è uno strumento composto da un insieme di componenti, legati l'uno con l'altro, in cui:

- ogni componente è specializzato ad una specifica funzionalità
- è possibile ridefinire ogni componente, attraverso l'ereditarietà, il polimorfismo e l'incapsulamento, fino ad entrare nei dettagli
- ogni componente lavora in collaborazione con tutti gli altri, al fine di riuscire a gestire anche i casi più complessi di simulazione

Un toolkit come Geant4 prevede anche la capacità di evoluzione e di mantenimento, in modo da rimanere costantemente aggiornato con le nuove scoperte della fisica, che vengono immediatamente riportate all'interno del software dal team di supporto attraverso il rilascio, di solito semestrale, di nuove release del software.

Il pacchetto Geant4 è il risultato di numerose collaborazioni internazionali di oltre un centinaio di scienziati provenienti da ogni parte del mondo e la suddivisione interna delle strutture di collaborazione prevede:

- il *Collaboration Board*, che gestisce le responsabilità e le risorse
- il *Technical Steering Board*, che si occupa delle questioni tecniche e scientifiche
- i *Working Groups*, che si occupano del mantenimento, dello sviluppo, dei controlli di qualità, ecc., nei domini del codice di loro competenza

Attualmente è anche possibile trovare sul sito ufficiale di Geant4 (all'indirizzo cern.ch/geant4/) una corposa community appassionata che fornisce preziosi consigli utili a risolvere i più svariati casi.

Una delle caratteristiche fondamentali di Geant4 è quella di riuscire a garantire un livello di dettaglio (verbosità) a scelta dell'utente. Sarà possibile quindi seguire la più piccola interazione, a discapito della velocità di esecuzione, oppure raccogliere esclusivamente i risultati finali della simulazione, riducendo ovviamente i tempi di esecuzione. Spetterà dunque all'utente scegliere il livello di verbosità desiderato a seconda delle proprie necessità ed esigenze. E' proprio questo il senso di simulazione discreta, in quanto si segue step dopo step l'avanzare della simulazione.

1.1.1 Design ed architettura

In Figura 1.3 è mostrato il diagramma delle categorie. Le categorie nella parte inferiore sono utilizzate da quelle nella parte superiore e forniscono quindi le fondamenta del toolkit.

Cominciamo con l'analizzare il diagramma ed il funzionamento delle categorie principali:

- *Run* ed *Event*: permettono di generare l'evento primario con la conseguente produzione di particelle secondarie causata dall'interazione con il materiale attraversato. La simulazione comincia sempre con il metodo `BeamOn` appartenente alla classe `G4RunManager`. La classe che rappresenta *Run* è `G4Run` mentre quella che rappresenta *Event* è `G4Event`. La gestione degli eventi avviene attraverso uno stack all'interno del quale vengono inserite le particelle da processare. Ogni particella viene quindi estratta e quando lo stack è vuoto l'evento termina.
- *Tracking*, *Track* e *Step*: sono legati alla propagazione della particella e permettono di limitare la lunghezza dello *step*. Uno *step* è rappresentato dalla classe `G4Step` e descrive l'intervallo tra due punti spaziali. E' evidente come le performance di una simulazione dipendano principalmente dalla capacità di calcolo della CPU; in Geant4 ogni particella è mossa *step by step* con una tolleranza che permette ottimizzazioni molto significative dei tempi di esecuzione, preservando comunque la precisione richiesta.

- *Hits* e *Digitization*: permettono la gestione della vera e propria parte sensibile del rivelatore creando la risposta alla particella che lo attraversa e generando un segnale logico. Un *hit*, rappresentato dalla classe `G4Hit`, è una istantanea dell'interazione di una traccia con una zona sensibile del rivelatore, chiamato anche *Sensitive Detector*. Alla fine di ogni evento, tutti gli oggetti *hit* vengono raccolti in collezioni di oggetti di `G4Event`.
- *Particles* e *Material*: queste due categorie implementano i metodi necessari per descrivere le proprietà fisiche di particelle e materiali. Le particelle sono basate sulla classe `G4ParticleDefinition` che descrive le proprietà base, come massa, carica, etc., e permette anche di gestire la lista dei processi a cui la particella deve essere sensibile. E' già definito un set di classi virtuali per la definizione dei principali tipi di particelle: leptoni, bosoni, mesoni, barioni, etc. che permettono l'implementazione di classi concrete come `G4Electron`, `G4PionMinus`, etc. Il design dei materiali rispecchia ciò che realmente esiste in natura: i materiali sono composti da un singolo elemento o da una mistura di elementi; gli elementi sono, a loro volta, composti da un singolo isotopo o da una mistura di isotopi. La categoria dei materiali si occupa anche dell'implementazione di metodi per la descrizione delle proprietà delle superfici.
- *Geometry*: permette la creazione di svariati tipi di strutture geometriche, come parallelepipedi (`G4Box`), sfere (`G4Sphere`), etc. In Geant4 la modellizzazione dei solidi è compatibile con lo standard ISO STEP in modo da permettere scambi di informazioni geometriche con i sistemi CAD. Due concetti fondamentali sono quelli di volume logico e volume fisico.

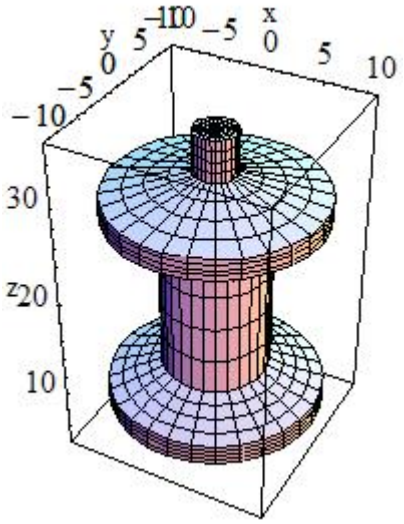


Figura 1.1: Esempio di Polycon implementato in Geant4 con la classe G4Polycon

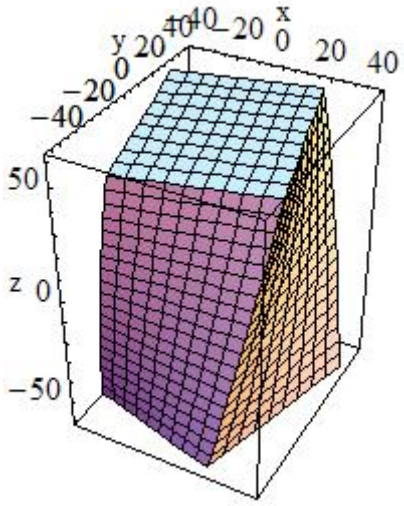


Figura 1.2: Esempio di Twisted Box implementato in Geant4 con la classe G4TwistedBox

Il *volume logico* rappresenta un elemento con un una certa forma, che può contenere altri volumi al proprio interno; esso inoltre permette l'accesso a tutte quelle informazioni che non dipendono dalla posizione fisica, come ad esempio al materiale o alle proprietà di *Sensitive Detector*. Il *volume fisico* rappresenta il posizionamento spaziale del volume logico. E' importante mantenere una struttura ad albero gerarchica di ogni volume, in cui ogni elemento ne contiene uno più piccolo, facendo attenzione al non sovrapporre od intersecare più volumi.

- *Physics*: si occupa della gestione di tutti i processi fisici che partecipano all'interazione tra la particella e la materia. I campi principali di cui si occupa questa categoria sono tre. Il *decadimento delle particelle*: Geant4 possiede una tabella di default con i valori di decadimento delle principali particelle. Benché la lunghezza dello *step* è calcolata basandosi sulla vita media delle particelle, è

molto importante considerare anche il loro processo di decadimento. La *fisica elettromagnetica*: Geant4 gestisce le interazioni elettromagnetiche di leptoni, fotoni, adroni e ioni. Sono incluse nel package i principali processi di ionizzazione, di bremsstrahlung, di multiple scattering, di Compton e di Rayleigh, effetti fotoelettrici, di annichilazione, di scintillazione, di rifrazione, di riflessione, di assorbimento e di Cherenkov. La *fisica adronica*: Geant4 possiede tutte le caratteristiche necessarie alla gestione della fisica adronica attraverso l'utilizzo di classi tra cui `G4HadronicProcess` e `G4HadronicInteraction`.

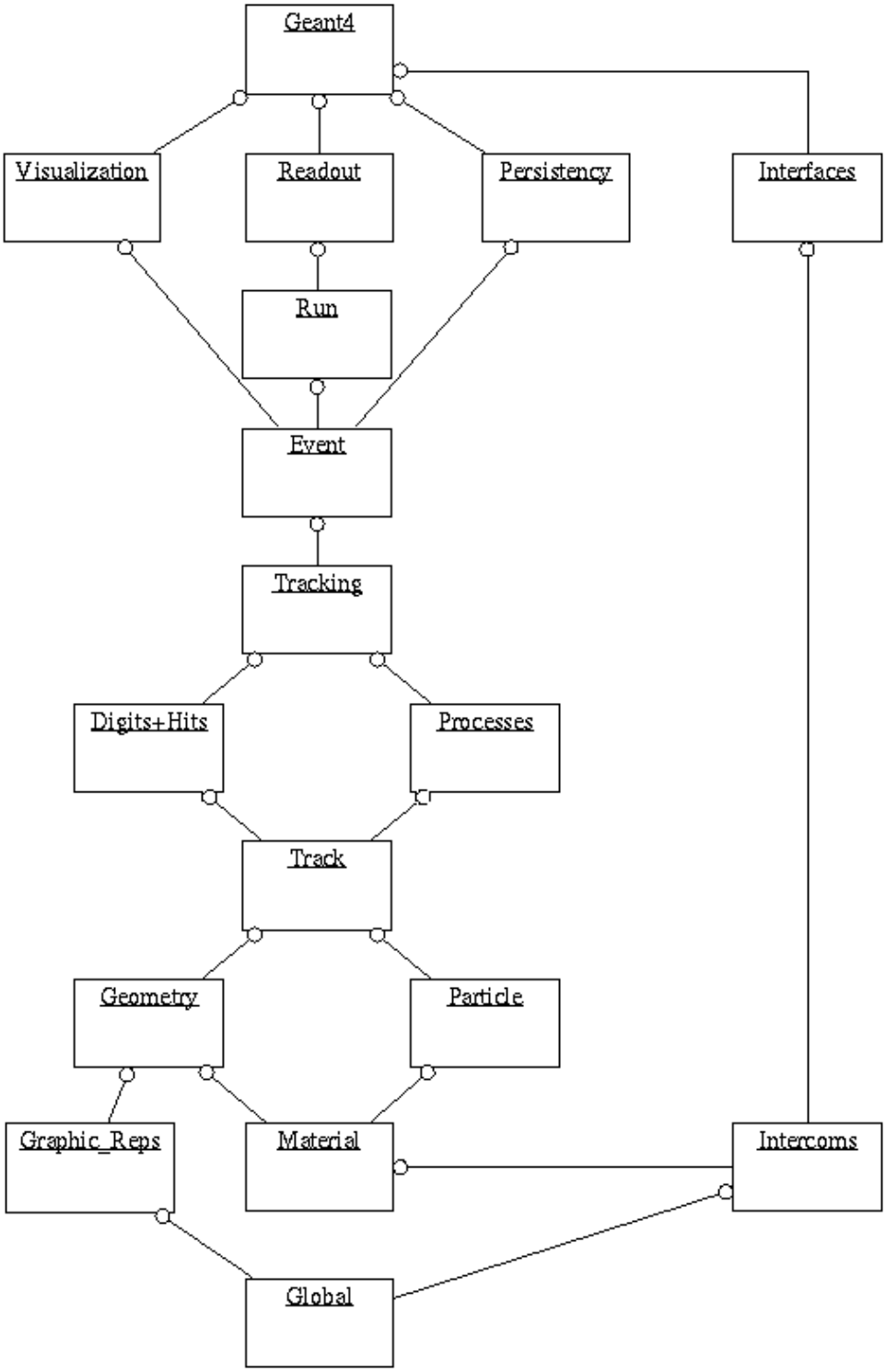


Figura 1.3: Il diagramma delle categorie di Geant4. Il cerchio nelle linee di giunzione indica una relazione; la categoria adiacente al cerchio utilizza la categoria giunta.

1.1.2 Sviluppo di una applicazione

Creare una applicazione in Geant4 vuol dire derivare ed implementare alcune classi concrete, partendo dalle classi astratte fornite dal kernel del toolkit. Questa operazione è possibile grazie alle proprietà della programmazione OO che permettono il *polimorfismo* ed il *dynamic-binding*. Esistono due tipi di classi: quelle obbligatorie (*mandatory user classes*) e quelle opzionali (*optional user classes*).

Le classi obbligatorie sono tre, di cui due sono definite *user initialization classes*, utilizzate per l'inizializzazione dell'applicazione, ed una *user action class*, utilizzata per l'inizializzazione dell'esecuzione.

Le classi astratte da cui derivare le proprie classi sono:

- `G4VUserDetectorConstruction`: nella classe derivata da questa classe astratta è necessario definire l'area di funzionamento della simulazione con la descrizione di tutte le geometrie, i materiali, le aree sensibili e le superfici.
- `G4VUserPhysicsList`: nella classe derivata da questa classe astratta è necessario definire tutte le particelle ed i processi fisici che interagiranno durante la simulazione.
- `G4VUserPrimaryGenerationAction`: questa è la *user action class* e nella classe derivata da questa è necessario specificare il modo in cui devono essere generate le particelle primarie.

L'esistenza di queste tre classi viene verificata all'inizio della simulazione dal `G4RunManager` al momento dell'invocazione dei metodi `initialize()` e `beamOn()`.

Le classi opzionali, invece, permettono all'utente la modifica e la personalizzazione del comportamento di default di Geant4. Le cinque principali *optional user classes* sono:

- `G4UserRunAction`: per impostare azioni da eseguire all'inizio ed alla fine di ogni run della simulazione.
- `G4UserEventAction`: per impostare azioni da eseguire all'inizio ed alla fine di ogni evento della simulazione.
- `G4UserStackingAction`: per personalizzare l'accesso allo *stack* in cui vengono memorizzate le informazioni di tracciamento delle particelle.
- `G4UserTrackingAction`: per impostare azione da eseguire all'inizio alla creazione ed al completamento di ogni tracciato.
- `G4UserSteppingAction`: per personalizzare le azioni da eseguire ad ogni step del processo di simulazione.

Ad esempio è possibile ottimizzare la priorità di processamento di ogni tipo di particella implementando la classe `G4UserStackingAction`.

2. Il problema fisico

Il cancro è una della maggiori cause di decesso nella società moderna subito dopo le malattie cardiache e di circolazione. Al momento della diagnosi, circa il 58% dei tumori non è diffuso nel corpo e non ha formato metastasi. In questo caso i tumori sono potenzialmente curabili con terapie localizzate, come l'intervento chirurgico, la chemioterapia, la radioterapia o una combinazione di questi. Circa il 22% di tutti i pazienti affetti da cancro sono curati con chirurgia. I metodi attuali di radioterapia curano invece il 12% dei casi, mente un ulteriore 6% riceve una combinazione tra chirurgia e radioterapia. Comunque, le modalità di trattamento tumorali, falliscono per il 18% dei casi, che causano circa 280.000 decessi annuali nella sola Unione Europea *[Rif. 5]*. Questo è dovuto all'impossibilità di rimuovere totalmente il tumore o dalla incapacità di applicare una corretta dose radioterapica per sterilizzare tutte le cellule affette dal cancro. Se non tutti, una parte consistente di questi pazienti potrebbero essere curati in futuro, se fossero migliorate le tecniche di trattamento localizzato dei tumori, in particolare la radioterapia.

Il progetto attualmente in corso al GSI (Gesellschaft für Schwerionenforschung - Società per la ricerca di ioni pesanti) di Darmstadt, è certamente uno dei migliori esempi di sviluppo: sin dal Dicembre del 1997 oltre 250 pazienti incurabili con radioterapia, a causa della vicinanza del tumore con organi a rischio, ad es. alla testa ed al collo, oppure alla zona pelvica, sono stati trattati con ioni di carbonio ad alta energia ottenendo risultati clinici molto promettenti. Lo scopo di questo progetto è di riuscire ad

utilizzare i vantaggi radiobiologici degli ioni di carbonio per radioterapia ad alta precisione, permettendo quindi l'irradiazione dei tumori localizzati nella vicinanza di organi a rischio.

La PET in-beam è attualmente l'unico metodo per il monitoraggio e lo studio delle conformazioni tumorali contemporaneamente alla loro cura con un metodo innovativo: l'adroterapia, un metodo che riduce gli effetti clinici del trattamento radioterapeutico classico con elettroni o raggi X (fotoni), implementando dei margini di sicurezza attorno al tumore e scegliendo il fascio adatto da impiegare.

L'adroterapia utilizza fasci di protoni (ioni di idrogeno), di ioni carbonio e di neutroni, che sono tutte particelle più pesanti degli elettroni e sono dette "adroni".

2.1 Radioterapia convenzionale

Per capire come l'adroterapia possa essere più precisa ed efficace consideriamo prima brevemente le caratteristiche degli elettroni e dei fotoni, vale a dire le radiazioni che la radioterapia convenzionale utilizza.

- Gli elettroni non penetrano in profondità nel corpo ma cedono tutta la loro energia nei primi 2 o 3 cm di tessuto al di sotto della superficie cutanea. Gli elettroni sono quindi molto utili per trattare i tumori della cute o comunque i tumori localizzati alla superficie del corpo, ma non possono essere utilizzati per i tumori profondi, che costituiscono la maggioranza dei casi.
- I raggi X penetrano invece in profondità nel corpo del paziente, lo attraversano e fuoriescono dalla parte opposta, cedendo progressivamente la loro energia. Le porzioni del corpo che vengono attraversate per prime ricevono una dose maggiore mentre quelle più lontane ricevono una dose minore.

Se un tumore situato profondamente nel corpo del paziente viene irradiato con un fasci di raggi X, ci sarà un "corridoio di ingresso", costituito dai tessuti che il fascio incontra prima di raggiungere il tumore, dopo ci sarà il tumore, ed infine un "corridoio d'uscita" costituito da tutti i tessuti che il fascio incontra dopo aver colpito il tumore. Il corridoio di ingresso riceverà la dose di radiazioni più alta, il tumore riceverà una dose intermedia ed il corridoio di uscita riceverà anch'esso una dose di radiazioni, anche se minore di quella ricevuta dal tumore. Il fatto che ci sia una porzione del corpo del

paziente (il corridoio di ingresso) che riceve una dose più alta di quella ricevuta dal tumore stesso è ovviamente uno svantaggio. Per superare questo limite non si utilizza un solo fascio ma si colpisce il tumore con tanti fasci da angolazioni diverse, di modo che i corridoi di ingresso e di uscita ricevono ognuno la dose da un singolo fascio mentre nel tumore si concentrano i contributi di tutti i fasci (vedi *Figura 2.1*).

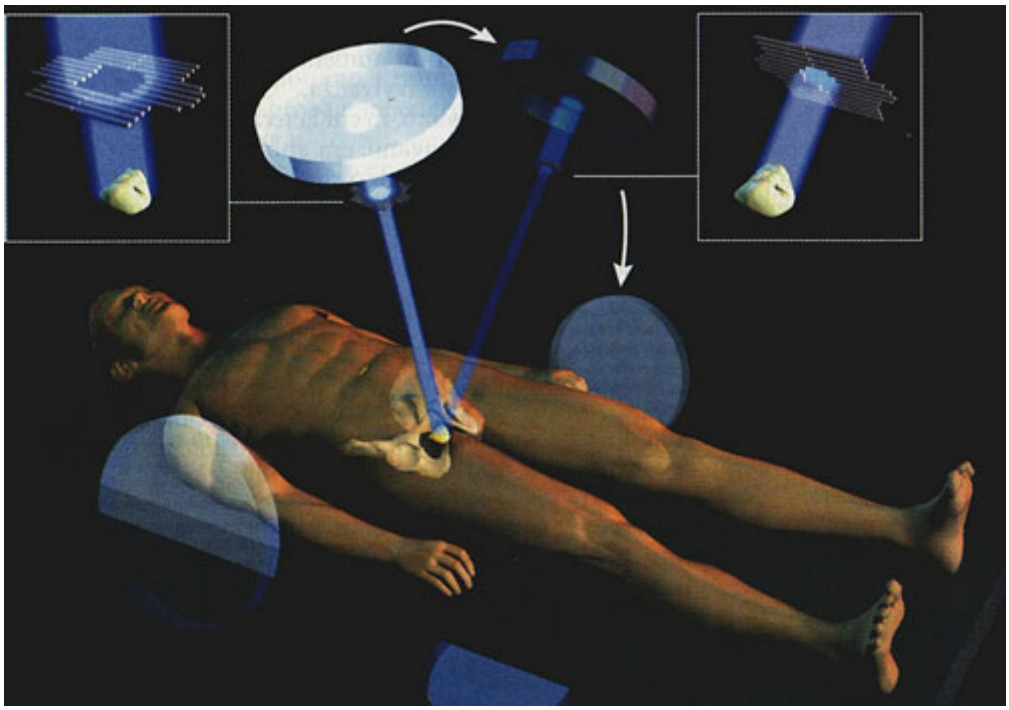


Figura 2.1: Schema funzionale della radioterapia convenzionale [Rif. 1].

Inoltre, la qualità delle radiazioni X è costante lungo tutto il percorso che compiono attraversando il corpo umano. Il danno che sono in grado di determinare è quindi lo stesso nel corridoio di entrata, nel tumore e nel corridoio di uscita. L'unica variabilità è nella quantità del danno, parametro che dipende dalle diversità della dose rilasciata.

2.2 Adroterapia

Consideriamo invece come si comporta un fascio di protoni che venga indirizzato sul corpo di un paziente. Tale fascio cede la sua energia in maniera costante fino ad una determinata profondità. Raggiunto questo punto (chiamato Picco di Bragg), esso cede una quantità di energia molto maggiore in uno spessore di pochi millimetri ed infine si arresta completamente. I tessuti che si trovano anche solo pochi millimetri più in profondità non ricevono quindi alcuna dose di radiazioni. La profondità a cui si arresta il fascio di protoni può essere variata usando protoni più o meno veloci: particelle più veloci penetrano in profondità mentre particelle più lente si fermano in superficie.

E' facilmente comprensibile come queste caratteristiche si prestino bene per irradiare i tumori profondi. Infatti il corridoio di uscita non riceve radiazioni ed il corridoio di ingresso riceve una dose minore di quella ceduta al tumore. La difficoltà nasce dal fatto che i tumori hanno in genere dimensioni di qualche centimetro e quindi molto maggiori dell'ampiezza del picco di Bragg: bisogna trovare quindi una tecnica per allargare tale picco.

A tal proposito si utilizza una tecnica di scansione (del genere Raster Scan) in cui l'irradiazione viene praticata con un piccolo fascio di protoni, quasi una specie di "pennellino" del diametro di qualche millimetro che penetra nel paziente e cede la maggior parte della dose a fondo corsa, nella regione del picco. Tale pennellino irradia quindi un piccolo volume del corpo umano. Durante la pianificazione del trattamento il tumore viene idealmente diviso in tanti piccoli volumi e questi vengono irradiati uno per

volta in rapida successione, facendo muovere il pennellino in alto, in basso, a destra ed a sinistra, mediante una serie di magneti controllati in modo molto preciso. Con questa tecnica si modifica la distribuzione XY del fascio, mentre per agire sulla profondità di irradiazione, quindi sulla Z del picco di Bragg, si cambia l'energia del fascio, oppure si modificano i parametri di macchina o si interpone tra il fascio ed il paziente un opportuno spessore di materiale che rallenta le particelle in maniera controllata (degrader). In questo modo il picco (per così dire la "punta del pennello") viene spostata più in superficie o in profondità al fine di colpire la zona da curare.

Inoltre, i raggi X, attraversando il corpo del paziente, non vengono modificati fino a che non giungono nel punto in cui interagiscono e quindi provocano danni dello stesso tipo a tutte le profondità. Gli adroni si comportano invece in modo del tutto diverso. Ogni adrone cede infatti la sua energia non in una singola interazione come i fotoni ma in tanti piccoli urti successivi. Dopo ogni urto un adrone, avendo perso un po' della sua energia, risulta lievemente rallentato. Quindi negli strati più superficiali del corpo del paziente avremo un fascio di adroni che ha avuto solo poche interazioni e perciò conserva tutta la velocità iniziale. A maggior profondità il fascio sarà invece composto da particelle che avendo ormai urtato parecchie volte sono più lente; ed infine nella regione del picco di Bragg ci saranno particelle lentissime che stanno quasi per fermarsi. Gli strati superficiali e quelli profondi sono quindi esposti a radiazioni con caratteristiche diverse: gli strati superficiali sono colpiti da particelle veloci e quelli profondi da particelle lente. Quando si usano i protoni i danni determinati da quelli veloci in superficie e dai lenti in profondità non differiscono in maniera significativa. Quando si usano altri adroni in genere succede che i danni prodotti dalle particelle lente

sono molto più difficili da riparare di quelli prodotti dalle particelle veloci. C'è quindi una parte superficiale del corpo che riceve una radiazione di una qualità simile ai fotoni X ed una parte più profonda che riceve una radiazione più efficace nel produrre danni.

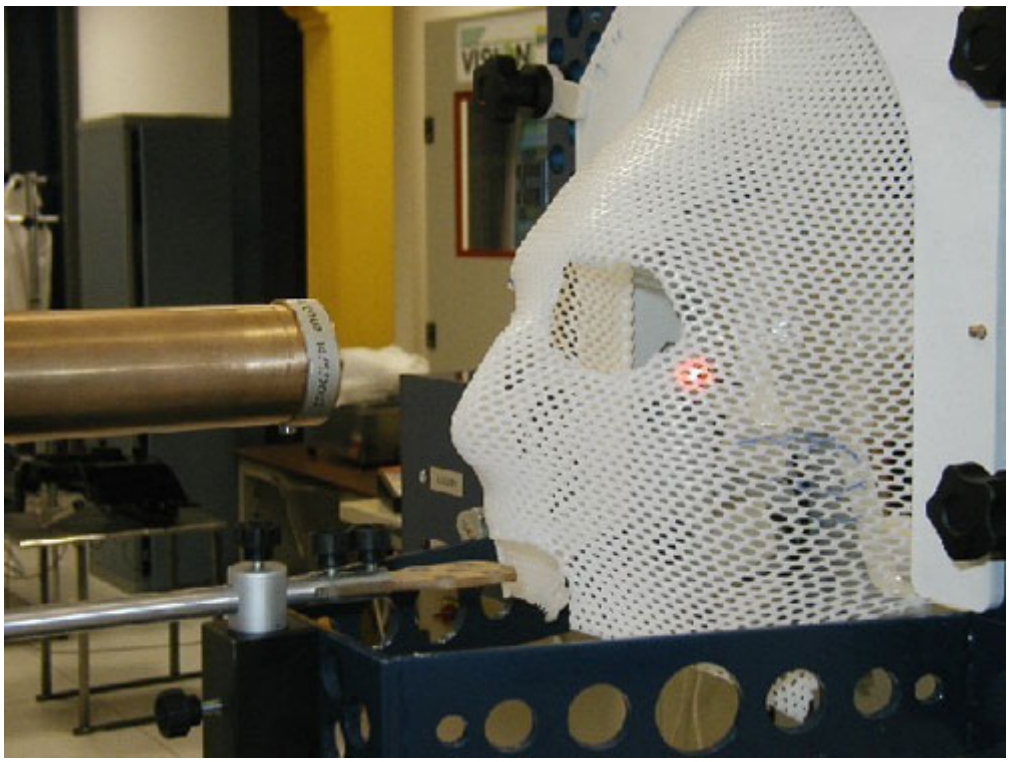


Figura 2.2: Maschera per adroterapia per la cura di tumori all'occhio ai LNS-INFN

In passato sono stati impiegati adroni diversi (come ad esempio gli ioni ossigeno, neon e le particelle alfa) ma attualmente si ritiene che gli ioni carbonio siano i più vantaggiosi perché quando attraversano i tessuti sani del corridoio di ingresso, che si vogliono risparmiare, non solo cedono una dose più bassa e quindi creano meno danni, ma quei pochi sono del tipo più facilmente riparabile. Invece quando hanno raggiunto il tumore, che si vuole distruggere, e che coincide con il picco di Bragg, da un lato cedono una dose più alta e quindi fanno molti più danni, e dall'altro lato si trovano ad avere

rallentato a sufficienza perché i molti danni che fanno all'interno del tumore siano del tipo che è quasi impossibile riparare.

In effetti gli ioni carbonio sono in grado di danneggiare il DNA in modo tale da renderlo irriconoscibile ai meccanismi di riparazione. Di fronte ai danni fatti dagli ioni carbonio non esistono tumori quindi radioresistenti.

2.3 La PET come strumento di diagnosi

La tomografia ad emissione di positroni (PET, Positron Emission Tomography) è una tecnica di medicina nucleare e diagnostica medica che produce immagini tridimensionali o mappe dei processi funzionali all'interno del corpo. La PET gioca un ruolo sempre maggiore nella verifica della risposta a terapie mediche, specialmente nella lotta contro i tumori [Rif. 6].

Figura 2.3: Una tipica PET (Tomografia ad Emissione di Positroni) [Rif. 2]

2.3.1 Decadimento β+ ed annichilazione

Nello studio delle cellule tumorali a metabolismo anaerobio è stata riscontrata una efficienza glicolitica più elevata rispetto alle cellule sane: viene quindi utilizzato un emettitore che possa essere facilmente veicolato. Caso tipico è l'impiego dell'isotopo ^{18}F-FDG (Fluoro-Desossiglucosio) che si concentra maggiormente nelle cellule di maggiore attività tumorali e meno in quelle sane.

Il processo di diagnosi comincia quindi con l'iniezione via endovena del liquido radio-tracciante. Dopo un tempo di attesa durante il quale la molecola metabolicamente attiva, raggiunge una determinata concentrazione all'interno dei tessuti organici da analizzare, il soggetto viene posizionano nello scanner.

Isotopo	**Tempo di dimezzamento** [min]	**Energia massima del positrone** [MeV]	**Metodo di produzione**
^{11}C	20.3	0.96	Ciclotrone
^{13}N	9.97	1.19	Ciclotrone
^{15}O	2.03	1.70	Ciclotrone
^{18}F	109.8	0.64	Ciclotrone
^{68}Ga	67.8	1.89	Generatore
^{82}Rb	1.26	3.15	Generatore

Tabella 2.1: Le proprietà dei principali isotopi [Rif. 11]

Gli isotopi sono atomi con un disequilibrio tra il numero di protoni ed il numero di neutroni all'interno del proprio nucleo. Questo disequilibrio porta ad un processo di

decadimento radioattivo, chiamato “decadimento beta” (β+), in cui un protone decade in tre particelle: neutrone, positrone e neutrino.

$$p \rightarrow n + e^{+} + \nu$$

Il neutrino è una particella dagli effetti trascurabili che non verrà trattata in tale elaborato. Il neutrone rimane all’interno del nucleo. Il positrone, invece, viene espulso verso l’esterno e, a causa delle collisioni con gli elettroni della materia circostante, perde la propria energia cinetica, dopo al massimo 1 mm. La sua velocità va quindi sempre a decrescere, fino a raggiungere l’energia termica, fin quando, avvicinandosi ad un elettrone, da inizio al processo di annichilazione.

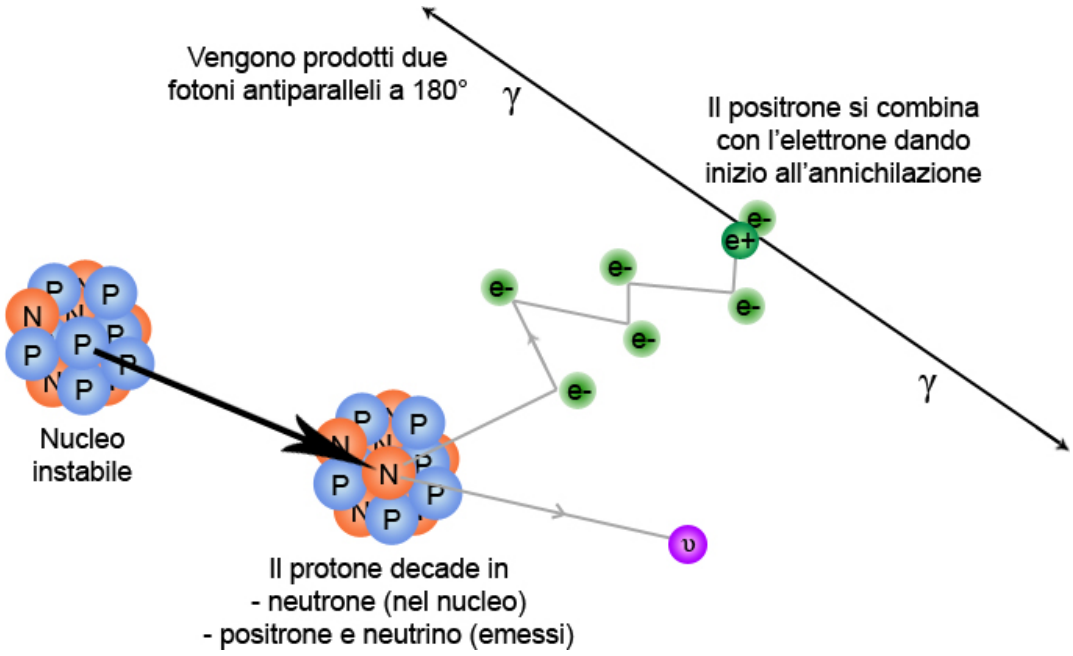


Figura 2.4: Decadimento ed annichilazione

Elettrone e positrone rappresentano rispettivamente materia ed antimateria che, a contatto, causano la trasformazione della propria massa in energia. Vengono infatti prodotti 2 fotoni γ con energia pari a 511 keV cadauno, emessi con angolo di 180° l'uno rispetto all'altro per conservare l'impulso totale del sistema.

$$e^+ + e^- \rightarrow 2\gamma$$

Per la Legge di conservazione dell'energia, la massa degli elettroni è completamente trasformata in energia (fotoni):

$$M(e^+) + M(e^-) \rightarrow 2E(\gamma)$$

quindi:

$$511keV + 511keV \rightarrow 2 \cdot 511keV$$

I fotoni generati vengono quindi rilevati quando raggiungono un materiale scintillante, nel dispositivo di scansione, dove creano un lampo luminoso, rilevato attraverso dei tubi fotomoltiplicatori o dei fotosensori. I fotoni che non raggiungono il rivelatore in coppia, cioè entro un intervallo di tempo di pochi nanosecondi con un angolo di 180°, non sono presi in considerazione.

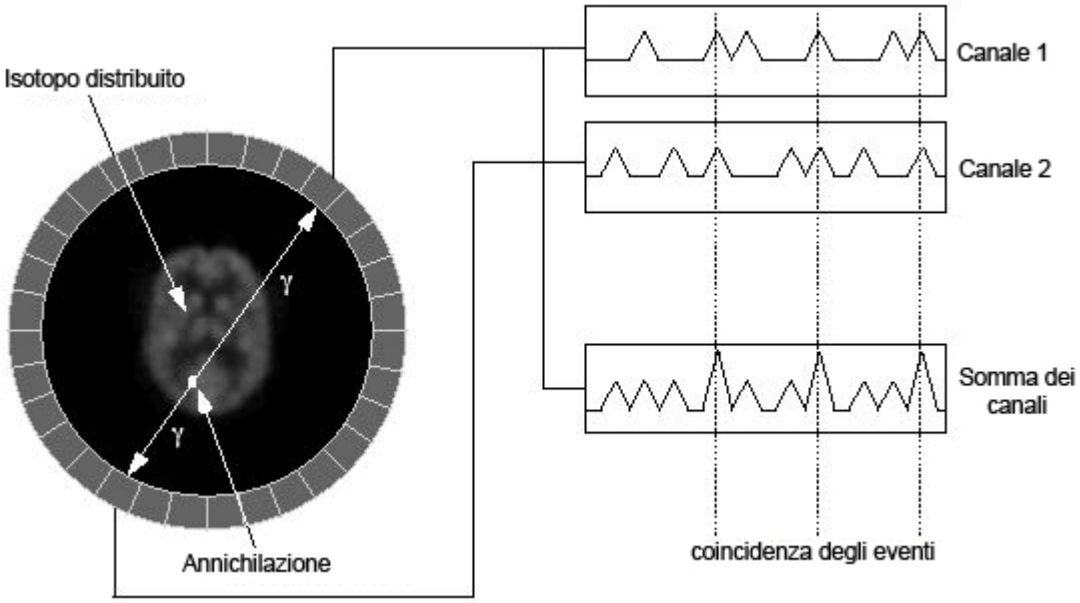


Figura 2.5: Vengono considerati esclusivamente i fotoni che raggiungono il rivelatore in coppia, ovvero in un intervallo di tempo di pochi nanosecondi con un angolo di 180°

Un importante effetto indesiderato da tenere in considerazione è l'*effetto Compton*, in cui il fotone interagisce con l'elettrone della materia del corpo umano. Questo processo causa l'aumento dell'energia cinetica dell'elettrone e la deviazione del fotone.

L'energia del fotone dopo l'interazione è data da:

$$E' = \frac{E}{1 + \left(\frac{E}{m_0 c^2}\right)(1 - \cos\theta)}$$

Dove:

- E è l'energia iniziale del fotone.
- E' è l'energia del fotone deviato.

- $m_o c^2$ è la massa dell'elettrone
- θ è l'angolo di deviazione

Questa equazione indica come a piccoli angoli di deviazione corrispondono basse perdite di energia; per esempio, per fotoni di 511 keV, l'effetto Compton causa una perdita del 10% di energia nel caso di un angolo di deviazione di 25 gradi.

Dalla misurazione della posizione in cui i fotoni colpiscono il rilevatore, si può ricostruire la posizione del corpo da cui sono stati emessi, permettendo la determinazione dell'attività o dell'utilizzo chimico all'interno delle parti del corpo investigate. Lo scanner utilizza la rilevazione delle coppie di fotoni per mappare la densità dell'isotopo nel corpo, sotto forma di immagini di sezioni (generalmente trasverse) separate fra loro di 5 mm circa. La mappa risultante rappresenta i tessuti in cui la molecola campione si è maggiormente concentrata e viene letta e interpretata da uno specialista in medicina nucleare o in radiologia al fine di determinare una diagnosi ed il conseguente trattamento.

2.4 La PET in-beam per il monitoraggio durante adroterapia

Le proprietà degli ioni di carbonio rendono il fascio come un bisturi molto affilato che deve essere utilizzato con molta precauzione, cosicché il tessuto esposto ad alta dose venga ristretto esclusivamente alla superficie tumorale da trattare. Per questa ragione, le tecniche di ricostruzione di immagine per monitorare la localizzazione del fascio sono molto utili.

Uno tra i metodi attualmente in fase di sviluppo è proprio la PET in-beam, in cui si fa uso della fisica della frammentazione nucleare tra proiettili di ^{12}C del fascio e nuclei del target. In particolare una piccolissima, ma significativa, frazione delle particelle di ^{12}C del fascio, frammentandosi per collisione lungo il loro percorso danno luogo a ^{11}C o ^{10}C, entrambi radioattivi $\beta+$. I positroni emessi dal decadimento vanno incontro, come detto, ad annichilazione e dunque producono una coppia di gamma da 511 keV che possono essere rivelati. Questo è il metodo utilizzato al GSI, con l'utilizzo di uno speciale scanner di positroni per la misura dell'attività radioattiva durante l'irradiazione del paziente. La PET in-beam è capace di rilevare, durante l'operazione di irradiazione, eventuali e indesiderate deviazioni del fascio e modifiche anatomiche del corpo. Riesce quindi a verificare l'accuratezza del fascio fornendo al radioterapista una stima della differenza di dosaggio tra quello emesso e quello pianificato. I primi prototipi di PET in-beam, implementati al GSI sono stati ottimizzati per tumori di piccole dimensioni. Per campi di irraggiamento più ampi, le ricostruzioni delle immagini del corpo e del tumore, sono soggette a disturbi, non permettendo quindi una esatta riproduzione in 3D.

2.5 I sensori

In questo paragrafo sono illustrate le basi per la corretta comprensione del funzionamento dei sensori utilizzati nelle PET moderne, spaziando dai processi di interazione fotone materia, al funzionamento di scintillatori e sensori.

2.5.1 Processi di interazione fotone materia

La rivelazione dei gamma è legata al processo di perdite di energia del fotone quando questo attraversa la materia che costituisce il rivelatore. Il numero di fotoni soppressi è proporzionale alla luminosità della sorgente ed allo spessore della materia attraversata. Consideriamo i tre principali processi di interazione fotone materia, che possono essere schematizzati come segue, in funzione dell'energia del fotone:

- 1 eV–100 keV: *Effetto fotoelettrico*
- 100 keV–1 MeV: *Effetto Compton*
- 1,022 MeV in poi: *Produzione di coppia*

Ognuno dei tre processi elimina o devia il fotone dalla direzione del fascio, quindi i fotoni che vengono osservati lungo tale direzione dopo aver attraversato il mezzo sono quelli che non hanno subito nessuna interazione e quindi possiedono l'energia originale.

Mediamente, in funzione del regime energetico, una parte di fotoni si ferma non appena entrata nel rivelatore, il resto continua a procedere e si assisterà ad un processo di rallentamento globale. Analizziamo ora più in dettaglio i tre processi di interazione.

L'*effetto fotoelettrico*, mostrato in *Figura 2.6*, consiste nell'emissione di cariche elettriche negative da un materiale, quando questo viene colpito da una radiazione elettromagnetica, come ad esempio la luce visibile o la radiazione ultravioletta. Nel nostro caso il fotone che incide su un atomo, viene assorbito e l'atomo rilascia un elettrone.

L'*effetto Compton* descrive l'urto elastico di un fotone su un elettrone. Questo effetto può essere spiegato semplicemente se si pensa ai fotoni come a particelle che urtano elasticamente contro gli elettroni presenti negli atomi, cedendogli energia. Lo scattering Compton avviene su elettroni liberi non legati al nucleo, contrariamente all'effetto fotoelettrico. Tuttavia se l'energia del fotone è alta rispetto all'energia di legame, questa si può trascurare in modo da considerare gli elettroni come liberi.

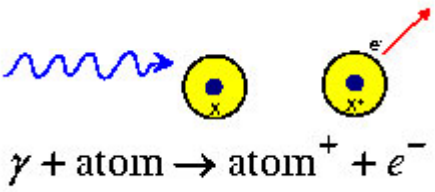


Figura 2.6: Effetto fotoelettrico

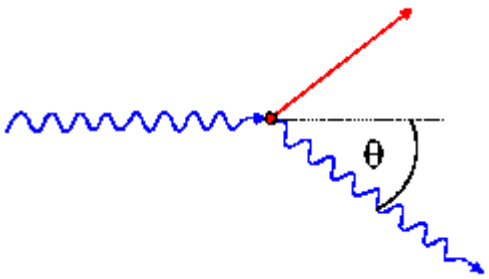


Figura 2.7: Effetto Compton

Nell'interazione, il fotone trasferisce ad un elettrone, che si suppone fermo, parte della sua energia e del suo impulso, come mostrato in *Figura 2.7*. Come risultato si avrà un fotone diffuso e l'elettrone deflesso.

Il *processo di produzione di coppia* è, invece, una reazione in cui un raggio gamma interagisce con la materia convertendo la sua energia in materia ed antimateria. Vedi *Figura 2.8*. Se un fotone gamma altamente energetico va ad impattare contro un bersaglio, subisce un urto anelastico che lo spacca in due materializzandone l'energia, e producendo una coppia di particelle composta da un elettrone (materia) e un positrone (antimateria). Il processo può avvenire solo quando l'energia del fotone è pari almeno alla somma delle masse delle particelle create, cioè per $E_\gamma > 2m_e c^2$ (1.022 Mev) e in presenza di un terzo corpo, in genere un nucleo, affinché ci sia conservazione della quantità di moto.

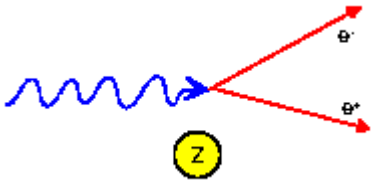


Figura 2.8: Processo di produzione di coppia

L'attenuazione del fascio, causata dai tre effetti appena descritti, dipende dal numero di fotoni che hanno interagito ed ha un andamento di tipo esponenziale:

$$I(x) = I_0 e^{-\mu x} = I_0 e^{-\frac{x}{\lambda}}$$

Dove:

I_0: intensità del fascio incidente

x: spessore dell'assorbitore

μ: coefficiente di assorbimento

$\lambda=1/\mu$: lunghezza di assorbimento

La lunghezza di assorbimento $\lambda=1/\mu$ rappresenta quella lunghezza per cui il numero di fotoni è pari ad 1/e-esimo del numero di fotoni iniziali.

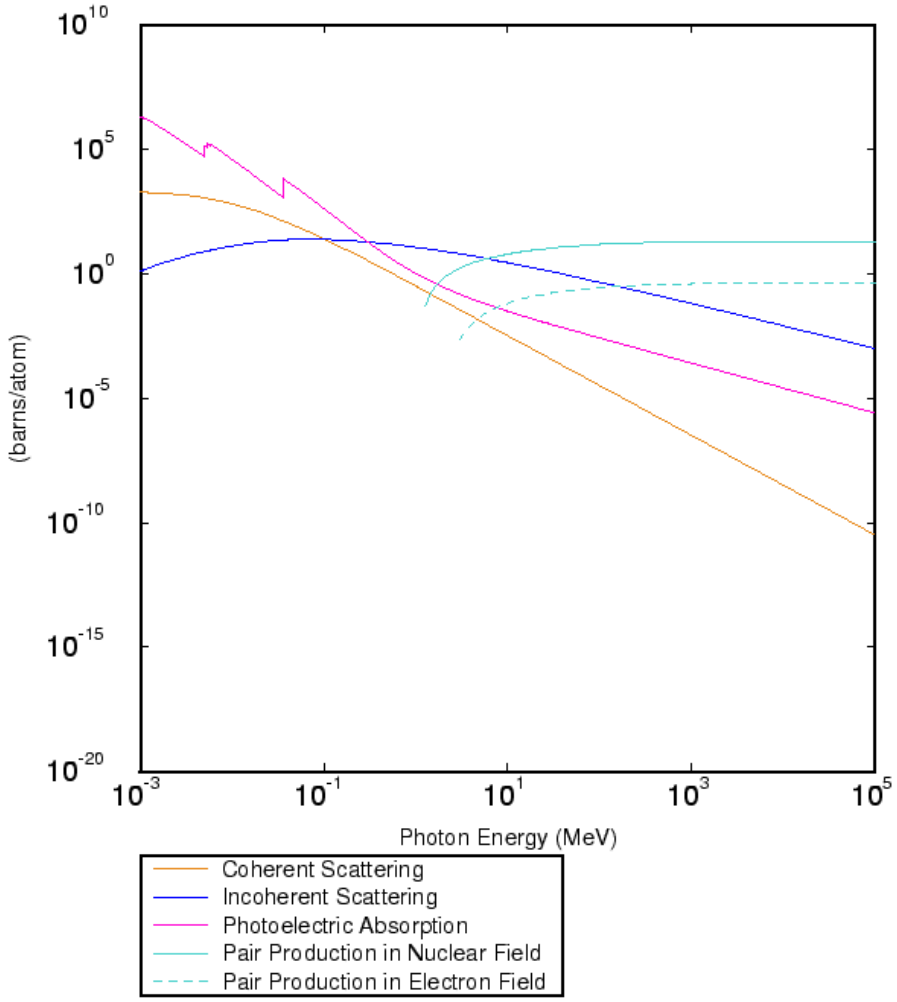


Figura 2.9: Processi di interazione fotone materia in funzione dell'energia, per il Cesio [Rif. 10]

2.5.2 Gli scintillatori

Uno scintillatore è un materiale capace di emettere impulsi di luce, in genere visibile o ultravioletta, quando viene attraversato da fotoni di alta energia o da particelle cariche. Al proprio passaggio la particella incidente cede parte della propria energia allo scintillatore causando, ad esempio, l'eccitazione di un elettrone che si sposta in un livello ad energia superiore. Quando l'elettrone decade al livello che occupava prima dell'eccitazione emette un fotone di energia relativamente bassa, tipicamente nel visibile. Tale impulso di luce viene poi rivelato ed amplificato da opportuni sensori, ad esempio da un fotomoltiplicatore. Esistono diverse tipologie di scintillatore che si distinguono per tipo di materiale da cui sono composti, i tempi di risposta, le lunghezze d'onda emesse ecc.: scintillatori a cristalli organici, scintillatori organici in soluzione, scintillatori plastici, scintillatori a gas ed infine *scintillatori a cristalli inorganici*, che sono proprio quelli utilizzati nelle PET, perché sono i più pesanti e quindi con Z più alto. Tali benefici saranno illustrati in dettaglio nei paragrafi successivi. Le loro principali caratteristiche sono riportate in *Tabella 2.2*.

In questo elaborato, le simulazioni sono state eseguite con quattro tipi di materiale scintillante, di cui tre pesanti (CsI [Rif. 14][Rif. 15], LSO [Rif. 16][Rif. 17] e BGO [Rif. 13]) ed uno leggero (BC408 [Rif. 12]), le cui proprietà sono mostrate in *Tabella 2.3*. Come illustrato nel Capitolo 4, lo scintillatore BC408 non è adatto per utilizzi come unico materiale scintillante nei sensori delle PET, ma può essere adoperato in combinazione ad uno degli scintillatori pesanti per migliorare le prestazioni globali del sensore.

Scintill.	**Comp.**	**ρ** [g/cm^3]	**Z_{eff}**	**λ=1/μ** [mm]	**N_{photon} /MeV** (x 10^3)	**τ_{scint}** [ns]	**Indice di rifrazione**
NaI	NaI:Tl	3.7	51	29,1	41	230	1,85
BGO	$Bi_4Ge_3O_{12}$	7.1	75	10,4	9	300	2,15
LSO	Lu_2SiO_5:Ce	7.4	66	11,4	26	40	1,82
GSO	Gd_2SiO_5:Ce	6.7	59	14,1	25	60	1,85
CsI	CsI:Tl	4.5	52	22,9	50	1000	1,80

Tabella 2.2: Le proprietà dei principali materiali scintillanti utilizzati oggi nelle PET

BC408		1.0			10	2,1	1,58

Tabella 2.3: Le proprietà del BC408, lo scintillatore leggero utilizzato nella simulazione

Nei grafici seguenti sono riportate le rese di scintillazione dei quattro scintillatori utilizzati nella nostra simulazione.

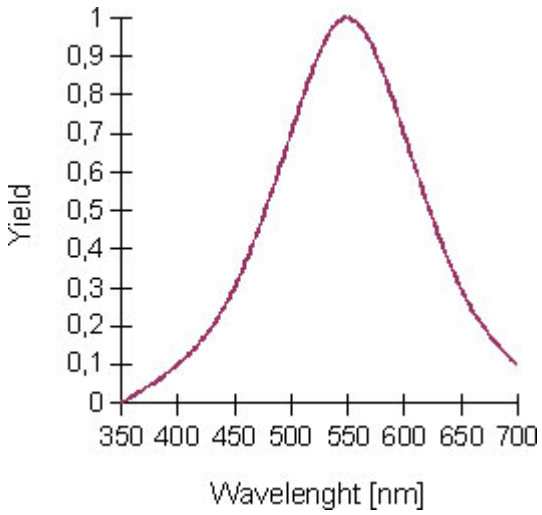


Figura 2.10: Resa di scintillazione del CsI

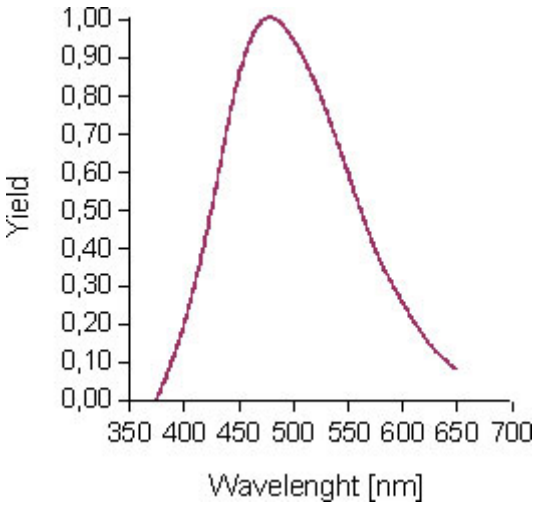


Figura 2.11: Resa di scintillazione del BGO

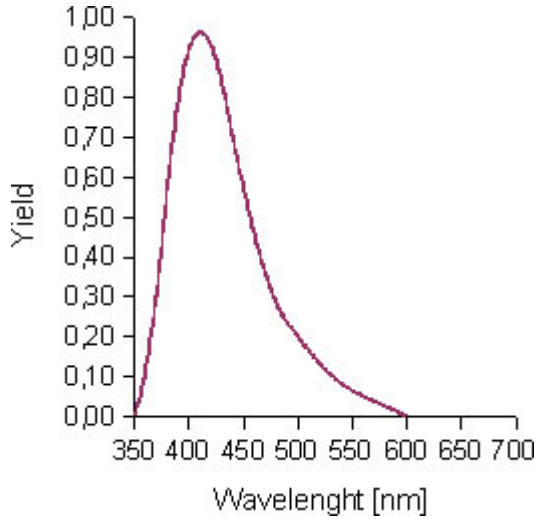


Figura 2.12: Resa di scintillazione del LSO

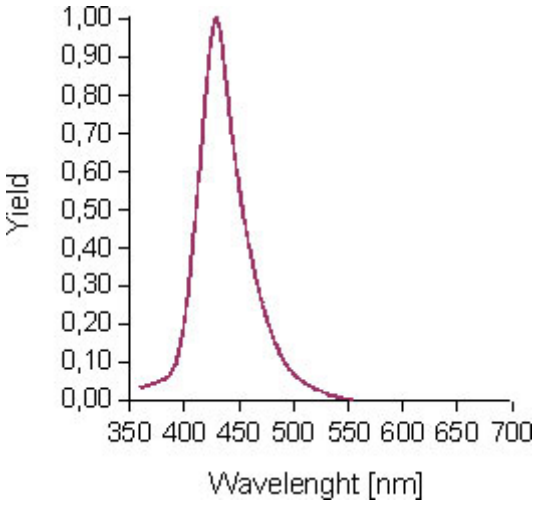


Figura 2.13: Resa di scintillazione del BC40

3. Applicazione sviluppata in Geant4

In questo capitolo sono descritti i dettagli del software che ho implementato in Geant4, con maggiori chiarimenti sul principio di funzionamento, sulle classi sviluppate e sulle scelte che ho effettuato, sia per la creazione che per l'ottimizzazione del codice. E' importante notare che in questo capitolo viene esclusivamente illustrata l'implementazione del software con la *geometria a cubo* mostrata nel paragrafo 2.1, ma il lavoro completo di questo elaborato consiste nella creazione di ulteriori geometrie e quindi nell'esecuzione di più simulazioni con il conseguente confronto tra i risultati da esse ottenute, come mostrato nel capitolo seguente.

Per la produzione di questo elaborato sono stato ospite del centro di calcolo dei LNS-INFN (Laboratori Nazionali del Sud – Istituto Nazionale di Fisica Nucleare) di Catania, in cui ho effettuato gli studi di ricerca e di sviluppo del codice.

La macchina su cui ho lavorato è un Pentium IV con CPU a 2,80 GHz e 256 MB di memoria RAM. Il sistema operativo è la distribuzione Red Hat Linux release 9 (Shrike) con Kernel 2.4.20-8.

3.1 Processi di funzionamento del software

La prima osservazione da effettuare consiste nel metodo di funzionamento del software. Geant4 gestisce le chiamate ai metodi con il modello *ad eventi*. Ovvero, ogni volta che uno step della simulazione genera un evento, vengono richiamati i metodi ad esso associati, meccanismo simile al *callback* utilizzato ampiamente in linguaggi procedurali come il C. Il compito del programmatore è quindi quello di decidere se catturare o meno l'evento, implementando il codice necessario per la sua gestione.

Come già descritto brevemente nel Capitolo 1 di questo elaborato, Geant4 gestisce due tipi di classi, quelle obbligatorie (*mandatory user classes*) e quelle opzionali (*optional user classes*). Cominciamo con l'analizzare le classi sviluppate, per comprenderne in dettaglio il funzionamento.

3.1.1 DetectorConstruction

In questo paragrafo intendo illustrare la creazione della geometria mostrata in *Figura 3.1*, dove la parte in grigio chiaro è un materiale scintillante composto da *Ioduro di Cesio* (CsI), lo strato in azzurro è composto da *vetro*, mentre quello in grigio scuro è il sensore di *Silicio*.

Le dimensioni di ogni sezione sono le seguenti:

- Scintillatore: 5 x 5 x 5 cm

- Strato di vetro: 1 x 5 x 5 cm
- Sensore di Silicio: 0.05 x 5 x 5 cm

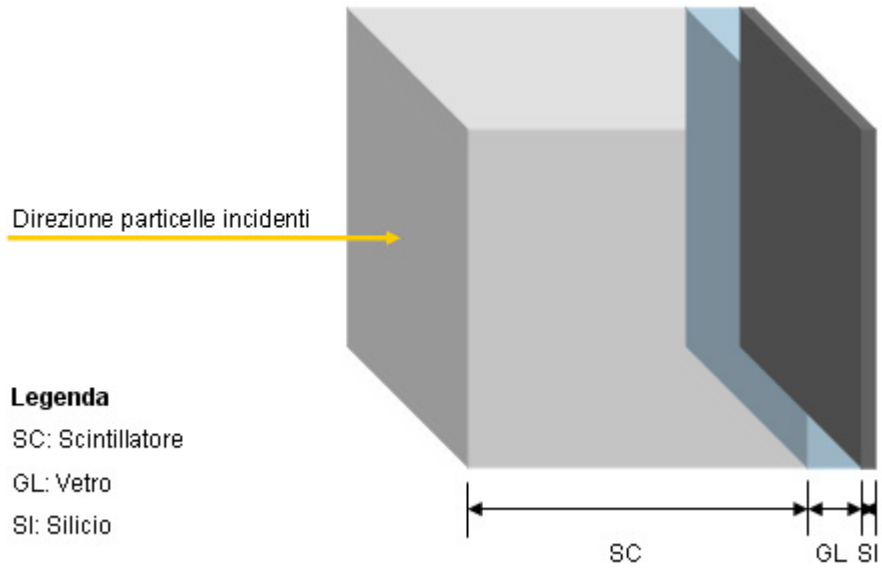


Figura 3.1: La geometria di test creata nella classe DetectorConstruction

La classe `DetectorConstruction` è la prima classe obbligatoria e deve sempre essere implementata. Estende `G4VUserDetectorConstruction` e permette la definizione di tutti i dettagli della geometria utilizzata, delle proprietà dei materiali e delle zone sensibili, ovvero dei *Sensitive Detector*.

Nel *Codice 3.1* è mostrata la creazione della classe `DetectorConstruction`, con tutti i propri metodi ed attributi, salvata all'interno del file *DetectorConstruction.hh*.

Oltre al costruttore ed al distruttore, deve sempre essere definito il metodo `Construct()`, all'interno del quale vengono inizializzate tutte le geometrie del sistema.

Tale metodo deve restituire un riferimento all'oggetto che rappresenta l'intero volume fisico realizzato.

```
class DetectorConstruction : public G4VUserDetectorConstruction {

  public:
    DetectorConstruction();
    ~DetectorConstruction();

    G4VPhysicalVolume* Construct();

  private:
    // Logical volumes
    //
    G4LogicalVolume* experimentalHall_log;
    G4LogicalVolume* Scintillator_log;
    G4LogicalVolume* Glass_log;
    G4LogicalVolume* Detector_log;

    // Physical volumes
    //
    G4VPhysicalVolume* experimentalHall_phys;
    G4VPhysicalVolume* Scintillator_phys;
    G4VPhysicalVolume* Glass_phys;
    G4VPhysicalVolume* Detector_phys;

};
```

Codice 3.1: Definizione di attributi e metodi della classe DetectorConstruction

Inoltre ho definito otto attributi, di cui i primi quattro rappresentano i volumi logici e gli altri quattro i volumi fisici, tutti necessari per la definizione delle quattro aree principali:

l'ambiente di simulazione (experimentalHall_log e experimentalHall_phys)

- lo scintillatore (`Scintillator_log` e `Scintillator_phys`)
- lo strato di vetro (`Glass_log` e `Glass_phys`)
- il sensore di silicio (`Detector_log` e `Detector_phys`)

L'implementazione del metodo `Construct()` deve essere effettuata all'interno del file *DetectorConstruction.cc*. Qui è necessario, innanzitutto, dichiarare tutti i materiali che verranno utilizzati per la simulazione.

```
G4NistManager* man = G4NistManager::Instance();

G4Material* Air   = man->FindOrBuildMaterial("G4_AIR");
G4Material* CsI   = man->FindOrBuildMaterial("G4_CESIUM_IODIDE");
G4Material* Glass = man->FindOrBuildMaterial("G4_GLASS_PLATE");
G4Material* Si    = man->FindOrBuildMaterial("G4_Si");
```

Codice 3.2: Definizione di alcuni materiali con l'utilizzo del metodo FindOrBuidMaterial

Ho definito il materiale aria (Air), lo Ioduro di Cesio (CsI), il vetro (Glass) ed il silicio (Si) accedendo ad un database di materiali già definiti [Rif. 3], così come mostrato nel *Codice 3.2*. E' anche possibile definire il proprio materiale componendo la molecola elemento per elemento. Questa è una operazione che, di solito, è necessaria solo nel caso in cui il materiale che si intende usare non è presente all'interno del database già citato.

Ad esempio, in alcune simulazioni, mi è servito definire il BGO, non presente tra quelli predefiniti, e quindi ho sviluppato il *Codice 3.3* che mi ha permesso la definizione dei singoli elementi e quindi la loro composizione per la creazione del materiale.

Nel passo successivo ho impostato le proprietà fisiche di ciascun materiale, come mostrato nel *Codice 3.4*. Questa operazione è il frutto di una accurata ricerca, che ha permesso una catalogazione ed una trascrizione in tabella, come già illustrato nel capitolo precedente.

```
G4String name, symbol;
G4double a, z, BGO_density = 7.13*g/cm3;
G4int natoms, ncomponents;

a = 208.98*g/mole;
G4Element* elBi = new G4Element
                        (name="Bismuth" , symbol="Bi", z=83. , a);

a = 72.61*g/mole;
G4Element* elGe = new G4Element
                        (name="Germanium", symbol="Ge", z=32., a);

a = 16.00*g/mole;
G4Element* elO  = new G4Element
                        (name="Oxygen"   , symbol="O" , z=8. , a);

G4Material* BGO = new G4Material
                        (name="BGO", BGO_density, ncomponents=3);

BGO->AddElement(elBi, natoms=4 );
BGO->AddElement(elGe, natoms=3 );
BGO->AddElement(elO , natoms=12);
```

Codice 3.3: Composizione del BGO elemento per elemento

Per mettere in atto l'operazione di definizione, è necessario impostare tali proprietà attraverso un oggetto di tipo `G4MaterialPropertiesTable` con l'ausilio dei metodi `AddProperty()` e `AddConstProperty()`. A tal proposito ho quindi creato diversi array, uno per ogni caratteristica da definire.

Le proprietà di rifrazione (`RefractiveIndex[]`), di resa (`Yield[]`) e di lunghezza di assorbimento (`AbsorptionLength[]`) variano in funzione dell'energia dei fotoni (`PhotonEnergy[]`). Mentre la costante di decadimento (`FASTTIMECONSTANT`), la resa di scintillazione (`SCINTILLATIONYIELD`) e la risoluzione (`RESOLUTIONSCALE`) sono proprietà costanti.

Importante notare come Geant4 effettui una *interpolazione lineare* di tutti i parametri che vengono impostati attraverso gli oggetti di tipo

`G4MaterialPropertiesTable`, cosicché, a qualunque energia compresa tra il minimo ed il massimo dell'array `PhotonEnergy[],` si riuscirà ad avere un valore continuo. E' facile intuire come, ad una quantità maggiore di dati inseriti all'interno della tabella, corrisponderà una maggiore accuratezza dell'interpolazione.

```
const G4int NUM_ENTRIES = 7;

G4MaterialPropertiesTable* CsI_PropertyTable = new
                                          G4MaterialPropertiesTable();

G4double PhotonEnergy[NUM_ENTRIES] =
   { 3.094*eV,2.750*eV,2.475*eV,2.250*eV,2.063*eV,1.904*eV,1.768*eV };

G4double CsI_RefractiveIndex[NUM_ENTRIES] =
   { 1.852,   1.824,   1.806,   1.793,   1.784,   1.779,   1.774    };

G4double CsI_Yield[NUM_ENTRIES] =
   { 0.1,     0.3,     0.7,     1.0,     0.7,     0.3,     0.1      };

G4double CsI_AbsorptionLength[NUM_ENTRIES] =
   { 35*cm,   35*cm,   35*cm,   35*cm,   35*cm,   35*cm,   35*cm    };

CsI_PropertyTable->AddProperty
             ("FASTCOMPONENT", PhotonEnergy, CsI_Yield, NUM_ENTRIES);

PropertyTable->AddProperty
          ("RINDEX", PhotonEnergy, CsI_RefractiveIndex, NUM_ENTRIES);

PropertyTable->AddProperty
       ("ABSLENGTH", PhotonEnergy, CsI_AbsorptionLength, NUM_ENTRIES);

CsI_PropertyTable->AddConstProperty("FASTTIMECONSTANT",   1000.*ns);

CsI_PropertyTable->AddConstProperty("RESOLUTIONSCALE",    1.0);

CsI_PropertyTable->AddConstProperty("SCINTILLATIONYIELD", 50000./MeV);

CsI->SetMaterialPropertiesTable(CsI_PropertyTable);
```

Codice 3.4: Definizione delle proprietà del CsI

Successivamente ho effettuato una operazione analoga con l'impostazione delle proprietà del vetro (*Codice 3.5*), del silicio (*Codice 3.6*) e dell'aria (*Codice 3.7*).

Queste operazioni concludono quelle di settaggio dei materiali e delle loro proprietà ed è ora quindi possibile creare la geometria del dispositivo da realizzare.

```
G4MaterialPropertiesTable* Glass_PropertyTable = new
                                          G4MaterialPropertiesTable();

G4double Glass_RefractiveIndex[NUM_ENTRIES] =
            { 1.5,    1.5,    1.5,    1.5,    1.5,    1.5,    1.5   };

G4double Glass_AbsorptionLength[NUM_ENTRIES] =
            { 100*m,  100*m,  100*m,  100*m,  100*m,  100*m,  100*m };

Glass_PropertyTable->AddProperty
         ("RINDEX", PhotonEnergy, Glass_RefractiveIndex, NUM_ENTRIES);

Glass_PropertyTable->AddProperty
     ("ABSLENGTH", PhotonEnergy, Glass_AbsorptionLength, NUM_ENTRIES);

Glass->SetMaterialPropertiesTable(Glass_PropertyTable);
```

Codice 3.5: Definizione delle proprietà del vetro

```
G4MaterialPropertiesTable* Si_PropertyTable = new
                                          G4MaterialPropertiesTable();

G4double Si_RefractiveIndex[NUM_ENTRIES] =
      { 1.5,     1.5,     1.5,     1.5,     1.5,     1.5,     1.5   };

G4double Si_AbsorptionLength[NUM_ENTRIES] =
     { 0.01*cm, 0.01*cm, 0.01*cm, 0.01*cm, 0.01*cm, 0.01*cm, 0.01*cm};

Si_PropertyTable->AddProperty
            ("RINDEX", PhotonEnergy, Si_RefractiveIndex, NUM_ENTRIES);

Si_PropertyTable->AddProperty
        ("ABSLENGTH", PhotonEnergy, Si_AbsorptionLength, NUM_ENTRIES);

Si->SetMaterialPropertiesTable(Si_PropertyTable);
```

Codice 3.6: Definizione delle proprietà del silicio

```
G4MaterialPropertiesTable* Air_PropertyTable =
                                        new G4MaterialPropertiesTable();

G4double Air_RefractiveIndex[NUM_ENTRIES] =
                                          { 1., 1., 1., 1., 1., 1., 1.};

Air_PropertyTable->AddProperty
           ("RINDEX", PhotonEnergy, Air_RefractiveIndex, NUM_ENTRIES);

Air->SetMaterialPropertiesTable(Air_PropertyTable);
```

Codice 3.7: Definizione delle proprietà dell'aria

Il primo passo è stato quello di ricreare l'ambiente di simulazione, che ho chiamato *experimentalHall*. In funzione delle ridotte dimensioni del sensore da riprodurre, ho scelto di creare un *experimentalHall* delle dimensioni 50x50x50cm, attraverso l'utilizzo della classe `G4Box`.

Ho riempito il volume logico con il materiale `Air` e quindi, ho posizionato il corrispondente volume fisico attraverso l'utilizzo della classe `G4ThreeVector()` che, in questo caso, restituisce come *(x, y, z)* le coordinate *(0, 0, 0)*.

```
G4double expHall_x = 50.*cm;
G4double expHall_y = 50.*cm;
G4double expHall_z = 50.*cm;

G4Box* experimentalHall_box = new G4Box
                     ("expHall_box", expHall_x, expHall_y, expHall_z);

experimentalHall_log = new G4LogicalVolume
                   (experimentalHall_box, Air, "expHall_log", 0, 0, 0);

experimentalHall_phys = new G4PVPlacement
   (0, G4ThreeVector(), experimentalHall_log, "expHall", 0, false, 0);
```

Codice 3.8: Definizione e posizionamento dell'experimentalHall

Successivamente ho effettuato una operazione simile per la creazione ed il posizionamento dello scintillatore. Nel *Codice 3.9* è mostrato il codice che ho implementato per la creazione di un cubo 5x5x5cm di *Ioduro di Cesio* (CsI), posizionato all'origine dell'*experimentalHall*. E' interessante notare come il simulatore impone che qualsiasi oggetto deve essere posizionato simmetricamente rispetto all'origine, infatti le dimensioni `Scintillator_dim_x`, `Scintillator_dim_y` e `Scintillator_dim_z` sono pari a 2,5 cm, ovvero la metà dei 5cm del lato del cubo.

```
G4double Scintillator_dim_x = 2.5*cm;
G4double Scintillator_dim_y = 2.5*cm;
G4double Scintillator_dim_z = 2.5*cm;

G4double Scintillator_pos_x = 0.*cm;
G4double Scintillator_pos_y = 0.*cm;
G4double Scintillator_pos_z = 0.*cm;

G4ThreeVector Scintillator_position = G4ThreeVector
         (Scintillator_pos_x, Scintillator_pos_y, Scintillator_pos_z);

G4Box* Scintillator_box = new G4Box("Scintillator_box",
          Scintillator_dim_x, Scintillator_dim_y, Scintillator_dim_z);

Scintillator_log = new G4LogicalVolume
                          (Scintillator_box, CsI, "Scintillator_log");

Scintillator_phys = new G4PVPlacement(0, Scintillator_position,
    Scintillator_log, "Scintillator", experimentalHall_log, false, 0);
```

Codice 3.9: Definizione e posizionamento dello scintillatore

Per completare la creazione del dispositivo, illustrato in precedenza in *Figura 3.1*, è stato necessario inserire lo strato di vetro dello spessore di 1 cm ed il sensore di silicio dello spessore di 500 micron, così come mostrato nelle porzioni di *Codice 3.10* e *Codice 3.11*. In entrambi i casi si continua ad utilizzare la classe `G4Box` che in questa

situazione permette la creazione della geometria desiderata. Più avanti si vedrà come impostare il sensore di silicio sensibile ai fotoni che lo colpiscono, in modo da essere definito *Sensitive Detector*.

```
G4double Glass_dim_x = 0.5*cm;
G4double Glass_dim_y = 2.5*cm;
G4double Glass_dim_z = 2.5*cm;

G4double Glass_pos_x = 3.*cm;
G4double Glass_pos_y = 0.*cm;
G4double Glass_pos_z = 0.*cm;

G4ThreeVector Glass_position = G4ThreeVector
                              (Glass_pos_x, Glass_pos_y, Glass_pos_z);

G4Box* Glass_box = new G4Box("Glass_box",
                              Glass_dim_x, Glass_dim_y, Glass_dim_z);

Glass_log = new G4LogicalVolume(Glass_box, Glass, "Glass_log");

Glass_phys = new G4PVPlacement(0, Glass_position,
                 Glass_log, "Glass", experimentalHall_log, false, 0);
```

Codice 3.10: Definizione e posizionamento dello strato di vetro

```
G4double Detector_dim_x = 0.025*cm;
G4double Detector_dim_y = 2.5*cm;
G4double Detector_dim_z = 2.5*cm;

G4double Detector_pos_x = 3.525*cm;
G4double Detector_pos_y = 0.*cm;
G4double Detector_pos_z = 0.*cm;

G4ThreeVector Detector_position = G4ThreeVector
                    (Detector_pos_x, Detector_pos_y, Detector_pos_z);

G4Box* Detector_box = new G4Box("Detector_box",
                     Detector_dim_x, Detector_dim_y, Detector_dim_z);

Detector_log = new G4LogicalVolume(Detector_box, Si, "Detector_log");

Detector_phys = new G4PVPlacement(0, Detector_position,
           Detector_log, "Detector", experimentalHall_log, false, 0);
```

Codice 3.11: Definizione e posizionamento del sensore di silicio

Il penultimo step di questa fase iniziale, ha previsto la definizione delle proprietà delle superfici.

In generale, se questo passaggio viene omesso, non verranno impostate alcune proprietà, ma dato che nella nostra simulazione si desidera che le superfici dello scintillatore e dello strato di vetro siano *polished* e riflettenti, in modo da "intrappolare" i fotoni generati dal processo di scintillazione all'interno del dispositivo, dovremo impostarne le caratteristiche. Anche in questo caso, così come per i materiali, le proprietà variano in funzione dell'energia dei fotoni (vedi array `PhotonEnergy[]` già definito in precedenza). Ho quindi utilizzato alcuni array che, passati come parametro al metodo `AddProperty()`, permettono il settaggio di tali proprietà.

E' interessante notare come nella porzione di codice in cui si crea l'oggetto `G4LogicalBorderSurface` si specifichino i due volumi fisici (`Scintillator_phys` e `experimentalHall_phys`) tra cui deve essere creata la superficie.

Cominciamo con l'analizzare i metodi invocati su `ScintillatorOpticalSurface` che permettono l'impostazione delle proprietà più importanti della superficie.

Il metodo `SetType()` permette di specificare il tipo di superficie ed i valori ammessi sono `dielectric_metal` e `dielectric_dielectric`, in funzione della composizione dei materiali tra cui si intende definire la superficie. Nel nostro caso ho utilizzato `dielectric_metal` per la superficie tra lo scintillatore e l'aria, mentre `dielectric_dielectric` per quella tra lo strato di vetro ed il sensore di silicio, così come mostrato nelle porzione di *Codice 3.12* e *Codice 3.13*.

```
G4OpticalSurface* ScintillatorOpticalSurface =
                   new G4OpticalSurface("ScintillatorOpticalSurface");

new G4LogicalBorderSurface("ScintillatorOpticalSurface",
                           Scintillator_phys, experimentalHall_phys,
                                        ScintillatorOpticalSurface);

ScintillatorOpticalSurface->SetType(dielectric_metal);
ScintillatorOpticalSurface->SetFinish(polishedfrontpainted);
ScintillatorOpticalSurface->SetModel(unified);
ScintillatorOpticalSurface->SetSigmaAlpha(0.1);

G4double ScintillatorSurfaceReflectivity[NUM_ENTRIES] =
                         { 0.95, 0.95, 0.95, 0.95, 0.95, 0.95, 0.95 };

G4double ScintillatorSurfaceEfficiency[NUM_ENTRIES] =
                         { 0.,   0.,   0.,   0.,   0.,   0.,   0.   };

G4double ScintillatorSurfaceSpecularLobe[NUM_ENTRIES] =
                         { 1.,   1.,   1.,   1.,   1.,   1.,   1.   };

G4double ScintillatorSurfaceSpecularSpike[NUM_ENTRIES] =
                         { 0.,   0.,   0.,   0.,   0.,   0.,   0.   };

G4double ScintillatorSurfaceBackScatter[NUM_ENTRIES] =
                         { 0.,   0.,   0.,   0.,   0.,   0.,   0.   };

G4MaterialPropertiesTable* ScintillatorOpticalSurfaceProperty =
                                      new G4MaterialPropertiesTable();

ScintillatorOpticalSurfaceProperty->AddProperty("REFLECTIVITY",
         PhotonEnergy, ScintillatorSurfaceReflectivity,  NUM_ENTRIES);

ScintillatorOpticalSurfaceProperty->AddProperty("EFFICIENCY",
           PhotonEnergy, ScintillatorSurfaceEfficiency, NUM_ENTRIES);

ScintillatorOpticalSurfaceProperty->AddProperty(
                                "SPECULARLOBECONSTANT", PhotonEnergy,
                       ScintillatorSurfaceSpecularLobe, NUM_ENTRIES);

ScintillatorOpticalSurfaceProperty->AddProperty(
                               "SPECULARSPIKECONSTANT", PhotonEnergy,
                       ScintillatorSurfaceSpecularSpike, NUM_ENTRIES);

ScintillatorOpticalSurfaceProperty->AddProperty("BACKSCATTERCONSTANT",
           PhotonEnergy, ScintillatorSurfaceBackScatter, NUM_ENTRIES);

ScintillatorOpticalSurface->SetMaterialPropertiesTable
                                  (ScintillatorOpticalSurfaceProperty);
```

Codice 3.12: Proprietà della superficie tra il CsI e l'aria

Il metodo `SetFinish()` permette di definire la rifinitura della superficie, se rugosa, levigata, pitturata o meno. I valori possibili sono: `ground`, `groundfrontpainted`, `groundbackpainted`, `polished`, `polishedfrontpainded` e `polishedbackpainted`. Nel nostro caso ho utilizzato `polishedfrontpainted` in quanto la superficie deve essere liscia e pitturata.

```
G4OpticalSurface* SiGlassOpticalSurface =
                        new G4OpticalSurface("SiGlassOpticalSurface");

new G4LogicalBorderSurface("SiGlassOpticalSurface", Detector_phys,
                                    Glass_phys, SiGlassOpticalSurface);

SiGlassOpticalSurface->SetType(dielectric_dielectric);
SiGlassOpticalSurface->SetFinish(polished);

G4double SiGlassOpticalSurfaceReflectivity[NUM_ENTRIES] =
                                        { 0., 0., 0., 0., 0., 0., 0.};

G4double SiGlassOpticalSurfaceEfficiency[NUM_ENTRIES] =
                                        { 1., 1., 1., 1., 1., 1., 1.};

G4MaterialPropertiesTable* SiGlassOpticalSurfaceProperty =
                                     new G4MaterialPropertiesTable();

SiGlassOpticalSurfaceProperty->AddProperty("REFLECTIVITY",
        PhotonEnergy, SiGlassOpticalSurfaceReflectivity, NUM_ENTRIES);

SiGlassOpticalSurfaceProperty->AddProperty("EFFICIENCY",
          PhotonEnergy, SiGlassOpticalSurfaceEfficiency, NUM_ENTRIES);

SiGlassOpticalSurface->SetMaterialPropertiesTable
                                     (SiGlassOpticalSurfaceProperty);
```

Codice 3.13: Proprietà della superficie tra il vetro e l'aria

Il metodo `SetModel()` permette di riferirsi a uno tra i due modelli standard, per la gestione delle interazioni tra particella e superficie, ovvero l'*Unified* ed il *Glisur*. I

valori possibili sono infatti `unified` e `glisur`. Il modello *Glisur* è originario del Geant3, mentre il modello *Unified* è stato adottato dal software di simulazione Detect.

Il metodo `SetSigmaAlpha()` setta un parametro necessario al corretto funzionamento del modello *Unified*. Se si decide di utilizzare il *Glisur* il `SetSigmaAlpha()` viene ovviamente ignorato.

Infine, osservando l'array `ScintillatorSurfaceReflectivity[]`, è possibile notare che ho considerato una riflettività costante della superficie `polishedfrontpainded` del 95%. Questo mi permette di simulare piuttosto realisticamente la non perfetta efficienza di riflessione causata dalle impurità nella superficie. L'approssimazione al 95% è comunque una ottima efficienza, basti pensare che i comuni specchi hanno una riflettività di circa l'80%.

Operazioni medesime di impostazione delle proprietà devono essere effettuate anche per ogni altra superficie di cui si ritiene importante specificarne le caratteristiche. Nel *Codice 3.13* è mostrato esclusivamente il settaggio della superficie tra lo strato di silicio e lo strato di vetro.

L'ultimo passo è stato quello di impostare il *Sensitive Detector*, in modo che lo strato di silicio diventi sensibile alle collisioni di particelle al suo interno. Questo permette, quindi, la collezione degli *hit* per l'estrazione dei dati necessari al fine della simulazione. A tal proposito ho sviluppato la classe `Sensor` le cui caratteristiche saranno illustrate più avanti in questo capitolo. Come è possibile leggere nel *Codice 3.14*, le operazioni consistono nella creazione preliminare di un oggetto gestore di *Sensitive Detector*, chiamato `SDman`. Successivamente deve essere istanziato un oggetto di tipo `Sensor`, chiamato `SDSensor`, e quindi deve essere passato come parametro ai due

metodi successivi, di cui il primo, `AddNewDetector()`, ne permette l'inserimento tra la raccolta dei sensori, mentre il secondo, `SetSensitiveDetector(),` permette di legarlo logicamente con il volume logico `Detector_log`.

```
G4SDManager* SDman = G4SDManager::GetSDMpointer();
G4String SDname;

G4VSensitiveDetector* SDSensor = new Sensor(SDname="/SDSensor");
SDman->AddNewDetector(SDSensor);
Detector_log->SetSensitiveDetector(SDSensor);

return experimentalHall_phys;
```

Codice 3.14: Impostazione del Sensitive Detector

Infine, come anticipato all'inizio del paragrafo, l'ultima operazione del metodo `Construct()` deve consistere nel ritorno del riferimento all'oggetto rappresentate l'intero volume fisico appena creato.

3.1.2 PhysicsList

`G4VUserPhysicsList` è la classe nella quale devono essere definiti tutti i processi fisici e tutte le particelle utilizzate durante la simulazione. Ho quindi implementato la classe `PhysicsList` che deriva `G4VUserPhysicsList` e deve implementare il metodo virtuale `ConstructProcess()`.

Oltre al costruttore ed al distruttore, devono essere creati due metodi di cui il primo è il `ConstructParticle()`, che si occupa della creazione delle particelle necessarie alla simulazione.

```
class PhysicsList : public G4VUserPhysicsList
{
  public:
   PhysicsList();
   ~PhysicsList();

  public:
    void ConstructParticle();
    void ConstructProcess();

    void SetCuts();


    //these methods Construct particles
    //
    void ConstructBosons();
    void ConstructLeptons();
    void ConstructMesons();
    void ConstructBaryons();

    //these methods Construct physics processes and register them
    //
    void ConstructGeneral();
    void ConstructEM();
    void ConstructOp();

  private:
    G4Scintillation*     theScintillationProcess;
    G4OpAbsorption*      theAbsorptionProcess;
    G4OpRayleigh*        theRayleighScatteringProcess;
    G4OpBoundaryProcess* theBoundaryProcess;
};
```

Codice 3.15: Definizione della classe PhysicsList

Dentro questo ho richiamato in sequenza i metodi `ConstructBosons()`, `ConstructLeptons()`, `ConstructMesons()` e `ConstructBaryons()`, per la costruzione rispettivamente dei Bosoni, Leptoni, Mesoni e Barioni (vedi *Codice 3.16*).

Ho preferito creare un metodo per ogni tipo di particella in modo da tenere il codice più ordinato, leggibile e scalabile.

```
void PhysicsList::ConstructParticle()
{
  ConstructBosons();
  ConstructLeptons();
  ConstructMesons();
  ConstructBaryons();
}
void PhysicsList::ConstructBosons()
{
  // pseudo-particles
  G4Geantino::GeantinoDefinition();
  G4ChargedGeantino::ChargedGeantinoDefinition();

  // gamma
  G4Gamma::GammaDefinition();

  // optical photon
  G4OpticalPhoton::OpticalPhotonDefinition();
}
void PhysicsList::ConstructLeptons()
{
  // leptons
  G4Electron::ElectronDefinition();
  G4Positron::PositronDefinition();
  G4NeutrinoE::NeutrinoEDefinition();
  G4AntiNeutrinoE::AntiNeutrinoEDefinition();
  G4MuonPlus::MuonPlusDefinition();
  G4MuonMinus::MuonMinusDefinition();
  G4NeutrinoMu::NeutrinoMuDefinition();
  G4AntiNeutrinoMu::AntiNeutrinoMuDefinition();
}
void PhysicsList::ConstructMesons()
{
  // mesons
  G4PionPlus::PionPlusDefinition();
  G4PionMinus::PionMinusDefinition();
  G4PionZero::PionZeroDefinition();
}
void PhysicsList::ConstructBaryons()
{
  // barions
  G4Proton::ProtonDefinition();
  G4AntiProton::AntiProtonDefinition();
  G4Neutron::NeutronDefinition();
  G4AntiNeutron::AntiNeutronDefinition();
}
```

Codice 3.16: Il metodo ConstructParticle() si occupa della creazione delle particelle

```
PhysicsList::ConstructProcess()
{
  AddTransportation();
  ConstructGeneral();
  ConstructEM();
  ConstructOp();
}

void PhysicsList::ConstructEM()
{
  theParticleIterator->reset();
  while( (*theParticleIterator)() ){
    G4ParticleDefinition* particle = theParticleIterator->value();
    G4ProcessManager* pmanager = particle->GetProcessManager();
    G4String particleName = particle->GetParticleName();

    if (particleName == "gamma") {
    // gamma
      // Construct processes for gamma
      pmanager->AddDiscreteProcess(new G4GammaConversion());
      pmanager->AddDiscreteProcess(new G4ComptonScattering());
      pmanager->AddDiscreteProcess(new G4PhotoElectricEffect());
    } else if (particleName == "e-") {
      //electron
      // Construct processes for electron
      pmanager->AddProcess(new G4MultipleScattering(),-1, 1, 1);
      pmanager->AddProcess(new G4eIonisation(),       -1, 2, 2);
      pmanager->AddProcess(new G4eBremsstrahlung(),   -1, 3, 3);
    } else if (particleName == "e+") {
      //positron
      // Construct processes for positron
      pmanager->AddProcess(new G4MultipleScattering(),-1, 1, 1);
      pmanager->AddProcess(new G4eIonisation(),       -1, 2, 2);
      pmanager->AddProcess(new G4eBremsstrahlung(),   -1, 3, 3);
      pmanager->AddProcess(new G4eplusAnnihilation(),  0,-1, 4);
    } else if( particleName == "mu+" ||
               particleName == "mu-" ) {
     //muon
     // Construct processes for muon
     pmanager->AddProcess(new G4MultipleScattering(),-1, 1, 1);
     pmanager->AddProcess(new G4MuIonisation(),      -1, 2, 2);
     pmanager->AddProcess(new G4MuBremsstrahlung(),  -1, 3, 3);
     pmanager->AddProcess(new G4MuPairProduction(),  -1, 4, 4);
    } else {
      if ((particle->GetPDGCharge() != 0.0) &&
          (particle->GetParticleName() != "chargedgeantino")) {
       // all others charged particles except geantino
       pmanager->AddProcess(new G4MultipleScattering(),-1,1,1);
       pmanager->AddProcess(new G4hIonisation(),       -1,2,2);
     }
    }
  }
}
```

Codice 3.17: Il metodo ConstructProcess() per la definizione dei processi fisici

Le operazioni che questi metodi devono effettuare sono piuttosto comuni e prevedono esclusivamente l'invocazione dei metodi statici per la definizione di ogni singolo tipo di particella.

Il metodo successivo è il `ConstructProcess()` che si deve occupare dell'inizializzazione di tutti i processi utili per la simulazione, tra cui quelli elettromagnetici, che sono i processi di nostro maggior interesse.

Anche in questo la procedura è piuttosto standard in quanto i processi che influiscono nell'interazione di ogni singola particella sono ben definiti e noti.

Come è possibile notare, nel *Codice 3.17* è stato illustrato esclusivamente il metodo `ConstructEM()`, utile alla definizione di tutti i processi elettromagnetici in funzione della particella considerata. Tali processi, tra cui lo scattering di Compton, l'effetto fotoelettrico e la creazione di coppie, sono descritti nel capitolo precedente.

3.1.3 PrimaryGeneratorAction

La `G4VUserPrimaryGenerationAction` è l'ultima delle tre classi obbligatorie. Da questa ho derivato la classe concreta `PrimaryGenerationAction`, in cui ho specificato come deve essere generato l'evento primario della simulazione. Per evento primario si intende sostanzialmente la posizione, il momento, l'energia ed il tipo di particella che deve essere irradiata all'inizio della simulazione.

E’ importante sapere che queste impostazioni, definite in fase di progetto e quindi di compilazione, possono essere modificate, in fase di esecuzione, attraverso l’utilizzo delle *macro*, trattate più in dettaglio in seguito.

```
class PrimaryGeneratorAction : public G4VUserPrimaryGeneratorAction
{
  public:
    PrimaryGeneratorAction(DetectorConstruction*);
    ~PrimaryGeneratorAction();

  public:
    void GeneratePrimaries(G4Event* anEvent);

  private:
    G4ParticleGun* particleGun;
    DetectorConstruction* myDetector;
};
```

Codice 3.18: Definizione della classe PrimaryGeneratorAction

L’attributo principale di questa classe è il `particleGun`, ovvero l’oggetto che si occupa di effettuare il vero e proprio *shoot* della particella.

```
PrimaryGeneratorAction::PrimaryGeneratorAction
                         (DetectorConstruction* myDC):myDetector(myDC)
{
  G4int n_particle = 1;
  particleGun = new G4ParticleGun(n_particle);

  G4ParticleTable* particleTable= G4ParticleTable::GetParticleTable();

  G4ParticleDefinition* particle =
                                particleTable->FindParticle("proton");

  particleGun->SetParticleDefinition(particle);
  particleGun->SetParticleEnergy(500*keV);
  particleGun->SetParticlePosition(G4ThreeVector
                                            (-2.5*cm, 0.*cm, 0.*cm));

  particleGun->SetParticleMomentumDirection(G4ThreeVector(1.,0.,0.));
}
```

Codice 3.19: Costruttore di PrimaryGeneratorAction

Nel costruttore, illustrato nel *Codice 3.19*, ho creato l'oggetto `particleGun`, con il compito di lanciare un protone di energia 500 keV, con posizione iniziale *(x, y, z)* alle coordinate *(-2,5 cm, 0 cm, 0 cm)* e momento *(1, 0, 0)*, ovvero nella direzione dell'asse *x*. E' semplice capire che le coordinate settate corrispondono a quella del lato sinistro del cubo, ovvero del punto di impatto della freccia gialla mostrata in *Figura 3.1*. Prima di scegliere queste coordinate ho effettuato diverse prove, impostando il punto di lancio della particella ancora più a sinistra dei -2,5 cm.

Questo però portava, a volte, alla interazione delle particelle irradiate con l'aria dell'`experimentalHall`, ancora prima di impattare con il cubo scintillatore, causando quindi ulteriori effetti fisici non utili ai fini della nostra simulazione.

Analizziamo ora il metodo `GeneratePrimaries()` all'interno del quale devono essere settate le caratteristiche che il `particleGun` deve possedere, evento per evento. Questo metodo viene infatti invocato all'inizio di ogni evento ed è quindi possibile modificare le caratteristiche del `particleGun` in funzione dell'evento passato come parametro. Nel nostro caso, dato che deve essere ripetuto lo stesso comportamento per ogni evento, mi sono limitato a ri-settare le impostazioni iniziali della particella, così come mostrato nel *Codice 3.20*.

```
void PrimaryGeneratorAction::GeneratePrimaries(G4Event* anEvent)
{
  particleGun->SetParticlePosition(G4ThreeVector
                                          (-2.5*cm,0.*cm,0.*cm));

  particleGun->SetParticleMomentumDirection(G4ThreeVector(1.,0.,0.));

  particleGun->GeneratePrimaryVertex(anEvent);
}
```

Codice 3.20: Metodo GeneratePrimaries() invocato all'inizio di ogni evento

3.1.4 SensorHit

Dopo aver completato le tre *mandatory user classes*, proseguiamo ora con l'analizzare le classi opzionali, ovvero tutte quelle classi in cui non è strettamente necessaria una modifica per il corretto *run* della simulazione, ma in cui è indispensabile effettuarla per personalizzare il comportamento di default di Geant4, che altrimenti non permetterebbe l'elaborazione ed il salvataggio dei risultati.

`SensorHit` è la classe che ho creato, derivata da `G4VHit`, che si occupa della gestione delle informazioni riguardanti il singolo *hit* all'interno del *Sensitive Detector*, cioè il nostro sensore di silicio. E' importante salvare queste informazioni in modo da poterle elaborare al termine di ogni evento e quindi creare l'istogramma con i dati di interesse alla fine del *run* della simulazione.

```
class SensorHit : public G4VHit
{
  .....
  .....

  private:

    G4int m_PDGEncoding;     // G4 PDGEncoding
    G4double m_edep;         // energy deposit for the current hit
    G4double m_stepLength;   // length of the step for the current hit
    G4double m_time;         // time of the current hit
    G4ThreeVector m_pos;     // position of the current hit
    G4String m_process;      // process on the current hit
    G4int m_trackID;         // track ID
    G4int m_parentID;        // parent track ID
    G4int m_voxelCoordinates;// voxellized voxel number

  .....
  .....
}
```

Codice 3.21: Porzione di codice della definizione della classe SensorHit

Nel *Codice 3.21* è mostrata una porzione della definizione della classe `SensorHit`, in cui mi sono limitato ad illustrare gli attributi privati del singolo *hit*. Tra questi, l'unico attributo realmente utilizzato ai fini della nostra simulazione è `m_time` che indica il tempo trascorso dall'inizio dell'evento all'istante in cui avviene l'*hit*. Questo valore verrà utilizzato per creare lo spettro del tempo di impatto del primo fotone con il sensore di silicio per ogni evento.

All'interno dello stesso file *SensorHit.hh*, ho anche definito i tipi `SensorHitsCollection` e `SensorHitAllocator` (vedi *Codice 3.22*), utilizzati in seguito all'interno la classe `Sensor`.

```
typedef G4THitsCollection<SensorHit> SensorHitsCollection;
extern G4Allocator<SensorHit> SensorHitAllocator;
```

Codice 3.22: Definizione dei tipi SensorHitsCollection e SensorHitAllocator

Da questa semplice descrizione, si intuisce che `SensorHit` è una classe che si occupa esclusivamente di memorizzare i dati di ogni *hit*. Possiamo definirlo come un contenitore all'interno del quale tutte le informazioni utili del singolo *hit* vengono salvate per poi essere riutilizzate in seguito.

Il tipo definito `SensorHitsCollection` è, invece, una collezione di oggetti di tipo `SensorHit`. Tale classe è di tipo `G4THitsCollection`, che è una classe di Geant4 già predefinita creata ad hoc per tale scopo.

3.1.5 Sensor

`Sensor` è la classe che ho derivato da `G4VSensitiveDetector` che si occupa di gestire gli *hit* e decidere se collezionarli o meno. Il *Codice 3.23* mostra la definizione dei metodi e degli attributi di questa classe.

```
class Sensor : public G4VSensitiveDetector
{
  public:
      Sensor(G4String name);
      ~Sensor();

      void Initialize(G4HCofThisEvent* HCE);
      G4bool ProcessHits(G4Step* aStep, G4TouchableHistory* ROhist);
      void EndOfEvent(G4HCofThisEvent* HCE);

  private:
      SensorHitsCollection* hitsCollection;
      G4MaterialPropertiesTable* EfficiencyTable;
      G4int collectionID;

};
```

Codice 3.23: Definizione della classe Sensor

Con l'attributo `hitsCollection`, così come il nome stesso suggerisce, ho voluto rappresentare la collezione degli *hit*, appena citata nel paragrafo precedente, all'interno della quale verranno inseriti tutti gli *hit* dello stesso evento che saranno ritenuti opportuni.

L'attributo `EfficiencyTable` viene invece spiegato in dettaglio più avanti in questo paragrafo.

```
Sensor::Sensor(G4String name) : G4VSensitiveDetector(name)
{
  G4String HCname;
  collectionName.insert(HCname="hitsCollection");
  collectionID = -1;
}
```

Codice 3.24: Costruttore della classe Sensor

Il costruttore di questa classe, mostrato nel *Codice 3.24*, ha esclusivamente il compito di settare il nome alla collezione di *hit* del *Sensitive Detector* (in questo caso `hitsCollection`). Per far ciò deve essere utilizzato il metodo `insert()` sull'oggetto `collectionName`, che è un attributo della classe astratta `G4VSensitiveDetector` e rappresenta un vettore di stringhe all'interno del quale devono essere definiti i nomi delle collezioni di *hit*.

Focalizziamo ora l'attenzione sui metodi di rivelazione degli *hit* e sulle scelte che sono state prese per la loro gestione.

Dato che si intende simulare l'efficienza del sensore di silicio e considerando che un dispositivo reale non permetterebbe la rivelazione di tutti i gamma che lo colpirebbero, anche nella simulazione si è voluta utilizzare una funzione per legare l'energia dei fotoni con l'efficienza di rivelazione del sensore di Silicio, chiamata *efficienza quantica* [Rif. 4], riportata in *Tabella 3.1* e memorizzata mediante l'oggetto `EfficiencyTable`.

Wavelenght [nm]	**Energy** [eV]	**Efficiency**
350	3,536	0,10
400	3,094	0,17
450	2,750	0,25
500	2,475	0,37
550	2,250	0,44
600	2,063	0,45
650	1,904	0,41
700	1,768	0,33
750	1,650	0,26
800	1,547	0,20
850	1,456	0,15
900	1,375	0,10
950	1,303	0,05
1000	1,238	0,02

Tabella 3.1: Efficienza quantica del Silicio

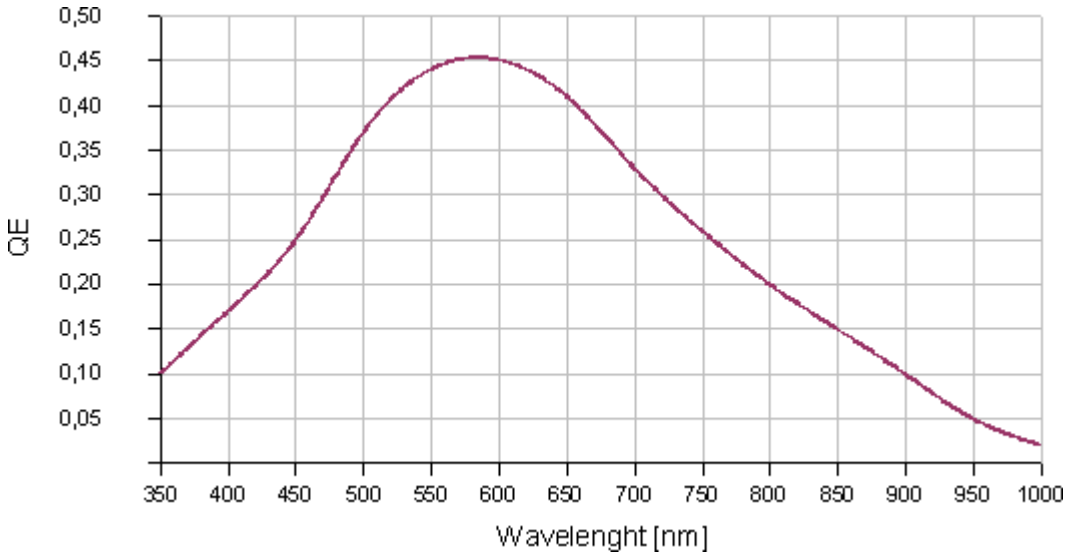


Figura 3.2: Efficienza quantica del Silicio in funzione della lunghezza d'onda

In realtà, come è possibile vedere, non è una vera funzione, ma una tabella che verrà interpolata linearmente da Geant4, come già accaduto in precedenza durante la definizione delle proprietà dei materiali e delle superfici, con l'utilizzo di oggetti di tipo `G4MaterialPropertiesTable`.

A tal proposito, all'interno del metodo `Initialize()`, richiamato automaticamente all'inizio della simulazione dal gestore degli eventi di Geant4 (vedi *Codice 3.25*), ho creato gli array `SiPhotonEnergy[]` e `SiEfficiency[]`, che contengono rispettivamente l'energia del fotone e l'efficienza ad essa associata. Ho quindi passato questi due array come parametri al metodo `AddProperty()` dell'oggetto `EfficiencyTable`.

```
void Sensor::Initialize(G4HCofThisEvent* HCE)
{
  .....
  .....

  const G4int NUM_EFF = 14;

  G4double SiPhotonEnergy[NUM_EFF] =
            { 3.536e-06, 3.094e-06, 2.750e-06, 2.475e-06, 2.250e-06,
              2.063e-06, 1.904e-06, 1.768e-06, 1.650e-06, 1.547e-06,
              1.456e-06, 1.375e-06, 1.303e-06, 1.238e-06            };

  G4double SiEfficiency[NUM_EFF] =
            { 0.10,      0.17,      0.25,      0.37,      0.44,
              0.45,      0.41,      0.33,      0.26,      0.20,
              0.15,      0.10,      0.05,      0.02                 };

  EfficiencyTable = new G4MaterialPropertiesTable();

  EfficiencyTable->AddProperty
        ("DetectorEfficiency", SiPhotonEnergy, SiEfficiency, NUM_EFF);

}
```

Codice 3.25: Definizione dell'efficienza quantica del Silicio

Analizziamo ora il metodo della classe `Sensor`, chiamato `ProcessHits()`. Questo metodo è invocato da `G4SteppingManager` quando viene effettuato uno step all'interno del `G4LogicalVolume` che ha il puntatore ad un *Sensitive Detector*. Nel nostro caso viene richiamato quando avviene un *hit* all'interno del sensore di silicio.

In questo metodo ho istanziato un oggetto di tipo `SensorHit`, nel caso in cui l'*hit* venga ritenuto significativo. Questa selezione viene effettuata trame la condizione *if*, illustrata in dettaglio nelle prossime pagine.

```
G4bool Sensor::ProcessHits(G4Step* aStep, G4TouchableHistory* ROhist)
{
  G4double check_energy = aStep->GetPreStepPoint()->GetTotalEnergy();

  if ( G4UniformRand() < (EfficiencyTable->GetProperty
                ("DetectorEfficiency")->GetProperty(check_energy)) ) {
      .....
      .....

      // time of the current step
      G4double aTime = newStepPoint->GetGlobalTime();

      SensorHit* aHit = new SensorHit();

      .....
      .....

      aHit->SetTime(aTime);

      .....
      .....

      hitsCollection->insert( aHit );
  }

  return true;

}
```

Codice 3.26: Gestione e salvataggio degli hit nel Sensitive Detector

Con la prima istruzione di questa porzione di codice, ho salvato, all'interno della variabile `check_energy`, l'energia del fotone che ha generato l'*hit*. Quindi, attraverso lo statement *if* successivo, ho generato un numero casuale tra 0 e 1 e l'ho confrontato con il valore dell'efficienza (`DetectorEfficiency`) corrispondente all'energia del fotone (`check_energy`). In tal modo ho effettuato una operazione di "setaccio" tra gli *hit*, in modo da simulare l'efficienza quantica reale del *Sensitive Detector*.

Nel caso in cui la condizione dell'*if* venga soddisfatta, procedo con l'acquisizione del tempo dell'*hit* all'interno della variabile `aTime`, quindi con la creazione dell'oggetto `aHit` di tipo `SensorHit` ed infine con l'inserimento di `aHit` all'interno della collezione `hitsCollection` da elaborare al termine dell'evento.

A questo punto è molto importante effettuare una serie di considerazioni.

La *Tabella 3.1* riporta in dettaglio l'*efficienza quantica* del silicio, ma non l'*efficienza geometrica* del sensore.

Infatti non è detto che tutti i sensori disponibili oggi sul mercato abbiano efficienza geometrica del 100%, ovvero che tutta l'area a disposizione sia realmente attiva. I tubi fotomoltiplicatori hanno efficienza geometrica 1, mentre i nuovi sensori SiPM, di cui i primi prototipi sono in via di sviluppo presso i LNS-INFN, hanno efficienza geometrica 0,36 [Rif. 18].

Per rendere meglio l'idea del concetto, si veda la *Figura 3.1*, in cui è schematizzata la rappresentazione logica di un sensore SiPM. I quadrati in grigio sono le aree attive.

Questi sensori hanno 70x70 celle delle dimensioni di 50x50 μm^2 cadauna, per un'area totale di 0,35x0,35cm^2 (queste dimensioni verranno anche utilizzate nel capitolo

successivo per sviluppare una geometria a *bastoncino*, differente da quella illustrata in questo capitolo).

L'area sensibile di ciascuna cella non è pero il 100%, ma una superficie di 30x30 μm^2.

Da rapidi calcoli si ottiene quindi:

$$\frac{3}{5} \times \frac{3}{5} = \frac{9}{25} = \frac{36}{100} = 0{,}36$$

Questo sta ad indicare che solo il 36% dei fotoni che raggiungono il sensore vengono rivelati ed è quindi solo su questi ultimi che è possibile applicare l'efficienza quantica.

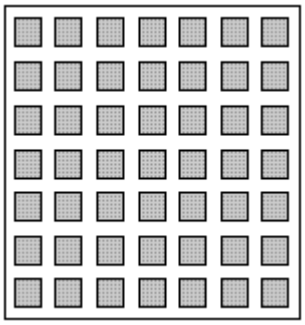


Figura 3.3: Struttura logica di un sensore SiPM

Ecco il motivo per cui, dopo le prime simulazioni di test, ho modificato la condizione *if* come mostrato nel *Codice 3.27* inserendo il fattore moltiplicativo 0,36, in modo da rendere ancora più realistica la risposta del sensore.

```
if ( G4UniformRand() < 0.36*(EfficiencyTable->GetProperty
                ("DetectorEfficiency")->GetProperty(check_energy)) ) {

   .....
   .....

}
```

Codice 3.27: Considerazione dell'efficienza geometrica del sensore

3.1.6 HistogramConstructor

Dopo aver gestito il sensore, ho progettato due classi, tra cui la presente, per permettere il salvataggio dei risultati della simulazione su file.

Ho scelto un formato piuttosto semplice per l'esportazione dei risultati su file, ovvero il formato testo, in modo da poterlo gestire con qualsiasi software per elaborare i risultati. Nel mio caso ho comunque utilizzato Microsoft Excel così come mostrato nel prossimo capitolo.

I file di testo rappresentano quindi gli istogrammi in cui ogni riga è una classe dell'istogramma. Gli istogrammi che ho memorizzato ai fini della nostra simulazione sono due e consentono la creazione dei seguenti spettri:

- Conteggio del numero di fotoni rivelato per evento
- Tempo di arrivo del primo fotone

Dato che la struttura del singolo istogramma è uguale per entrambi i casi, ho ritenuto opportuno creare la classe `HistogramConstructor` che si occupi della gestione e della manipolazione dei dati all'interno del singolo istogramma.

```
class HistogramConstructor
{
 private:

  G4int nElements;
  G4double* histo;       // pointer to a dynamic array

 public:

  HistogramConstructor(G4int n);
  ~HistogramConstructor();

  bool Add(G4int pos);
  void SetZero();
  bool Save(G4String FileName, bool Approximate);

  inline G4int GetSize() { return nElements; }

  inline void SetValue(G4int pos, G4double val) { histo[pos] = val; }
  inline G4double GetValue(G4int pos) { return histo[pos]; }

};
```

Codice 3.28: Definizione della classe HistogramConstructor

Visto però che le dimensioni degli istogrammi variano in funzione dello spettro da memorizzare, ho deciso di non creare un array di dimensioni fisse, ma bensì un array dinamico (vedi *Codice 3.28*), con nome `histo` e numero di elementi salvato in `nElements`, stabilito all'interno del costruttore (vedi *Codice 3.29*).

Analizziamo ora i tre metodi principali che ho sviluppato per questa classe, ovvero `Add()`, `SetZero()` e `Save()`.

```
HistogramConstructor::HistogramConstructor(G4int n)
{
  nElements = n;
  histo = new G4double[nElements];

  SetZero();
}
```

Codice 3.29: Costruttore della classe HistogramConstructor

Il più semplice dei tre è il `SetZero()`, che, come mostrato nel *Codice 3.30*, si occupa esclusivamente dell'azzeramento di tutti i valori dell'istogramma.

```
void HistogramConstructor::SetZero()
{
  for(G4int i=0; i<nElements; i++) {
    histo[i] = 0;
  }
}
```

Codice 3.30: Azzeramento dell'istogramma

Il metodo `Add()` si occupa, invece, di incrementare di 1 il valore dell'istogramma corrispondente alla posizione passata come argomento attraverso il parametro `pos` (vedi *Codice 3.31*).

```
bool HistogramConstructor::Add(G4int pos) {

  if(pos < 0 || pos >= nElements)
    return false;

  histo[pos] = histo[pos] + 1;
  return true;

}
```

Codice 3.31: Incremento di 1 del valore dell'istogramma

L'ultimo dei tre, il metodo `Save()`, permette il salvataggio dell'istogramma su un file di testo, il cui nome (`FileName`) è passato come parametro (vedi *Codice 3.32*).

Per il salvataggio su file, tra tutte le alternative fornite dal C++, ho preferito utilizzare l'*output file stream* (`ofstream`) in quanto piuttosto semplice da manipolare attraverso l'utilizzo dell'operatore <<. L'utilizzo di tale operatore, anche se meno efficiente di metodi ad esso analoghi, non influisce significativamente sulle prestazioni in quanto viene utilizzato una sola volta alla fine della simulazione.

```
bool HistogramConstructor::Save(G4String FileName) {

  ofstream HistoFile;

  HistoFile.open(FileName, ios::out);

  if(!HistoFile.is_open())
    return false;

  for(G4int i = 0; i < nElements; i++)
  {
      HistoFile << histo[i] << G4endl;
  }

  HistoFile.close();

  return true;

}
```

Codice 3.32: Salvataggio dell'istogramma su file di testo

3.1.7 Histogram

L'ulteriore classe che ho progettato è la `Histogram` che si occupa della gestione complessiva dei due istogrammi (`hCountHits` e `hFirstHit`) utili ai fini della simulazione.

```
class Histogram
{
 private:

  HistogramConstructor* hCountHits;
  HistogramConstructor* hFirstHit;

 public:

  Histogram();
  ~Histogram();

  bool FileSave();

  void InitializeHistograms();

  void SetCountHits(G4int c);
  void SetFirstHit(G4int c);

};
```

Codice 3.33: Definizione della classe Histogram

Si è detto in precedenza della necessità di creare istogrammi di dimensioni variabili. Il metodo `InitializeHistograms()` svolge proprio questa mansione, come mostrato nel *Codice 3.34*.

L'istogramma `hFirstHit`, che ho utilizzato per il salvataggio del tempo di impatto del primo fotone nel sensore, l'ho creato di 5000 elementi, ognuno corrispondente ad 1 picosecondo, per un totale di 5000 picosecondi.

```
void Histogram::InitializeHistograms(G4int n)
{
  hFirstHit  = new HistogramConstructor(5000);
  hCountHits = new HistogramConstructor(n);
}
```

Codice 3.34: Inizializzazione degli istogrammi

Dopo diversi test, mi sono reso conto come questo valore massimo sia ampiamente sufficiente per le simulazioni di nostro interesse, in quanto, l'avvenimento di un *hit* dopo i primi 5000 picosecondi è molto improbabile e comunque statisticamente irrilevante.

Considerazioni analoghe non possono essere fatte per l'istogramma `hCountHits`. A questo istogramma ho dato il compito di memorizzare il numero di *hit* per evento. Le sue dimensioni, quindi, variano in funzione della resa del materiale scintillante e dell'energia dei fotoni. Ad esempio, nel caso dello *Ioduro di Cesio*, che ha una resa di 50.000 fotoni/MeV, se s'inviano particelle d'energia pari a 500 keV, i fotoni generati dal processo di scintillazione saranno circa 25.000. Queste sono proprio le dimensioni massime da impostare nell'istogramma `hCountHits`. Per questo motivo, ho deciso di passare una variabile `n` come parametro al metodo `InitializeHistograms()`. Maggiori dettagli sul settaggio di `n` sono disponibili nel *Codice 3.40*.

I metodi successivi, `SetCountHits()` e `SetFirstHit()`, sono stati creati per permettere l'incremento di 1 degli istogrammi corrispondenti, così come mostrato nel *Codice 3.35* e nel *Codice 3.36*.

```
void Histogram::SetCountHits(G4int c) {

  if(c>= 0 && c < hCountHits->GetSize())
    hCountHits->Add(c);

}
```

Codice 3.35: Incremento di 1 dell'istogramma hCountHits

```
void Histogram::SetFirstHit(G4int c) {

  if(c>= 0 && c < hFirstHit->GetSize())
    hFirstHit->Add(c);

}
```

Codice 3.36: Incremento di 1 dell'istogramma hFirstHit

Infine l'ultimo dei metodi della classe `Histogram` si occupa del salvataggio su file di testo dei due spettri creati (vedi *Codice 3.37*), richiamando il metodo `Save()` sugli oggetti di tipo `HistogramConstructor`.

```
bool Histogram::FileSave() {

  bool h1 = hCountHits->Save("CsI_CountHistogram.txt");
  bool h2 = hFirstHit->Save("CsI_FirstHitHistogram.txt");

  if( h1 && h2 )
    return true;

  return false;

}
```

Codice 3.37: Salvataggio su files degli istogrammi

3.1.8 RunAction

In Geant4, il *run* è la più grande unità di simulazione, in cui si segue effettivamente il progresso, evento per evento, di tutto il processo. Sia la classe `RunAction`, che la classe `EventAction`, sono le più importanti per la gestione della sequenza di eventi da far eseguire al software.

La `RunAction` è una classe che ho derivato da `G4UserRunAction` e deve disporre dell'implementazione di due metodi obbligatori, ovvero `BeginOfRunAction()` e `EndOfRunAction()`, automaticamente richiamati rispettivamente all'inizio ed alla fine del run. Nel *Codice 3.38* è mostrata la definizione della classe `RunAction`.

```
class RunAction : public G4UserRunAction
{
  private:
    Histogram* TotalHisto;

  public:
    RunAction();
    ~RunAction();

  public:
    void BeginOfRunAction(const G4Run*);
    void EndOfRunAction(const G4Run*);

    inline void CountHits(G4int c) { TotalHisto->SetCountHits(c); }

    inline void FirstTimeHit(G4int c) { TotalHisto->SetFirstHit(c); }

};
```

Codice 3.38: Definizione della classe RunAction

Cominciamo con l'analizzare il costruttore, all'interno del quale ho inserito alcune righe per la generazione di una sequenza random di eventi. Questa porzione di codice

non è obbligatoria, ma se omessa causerà la ripetizione dei medesimi eventi ad ogni *run* della simulazione. A tal proposito, Geant4 propone differenti *random engine* e quello che ho utilizzato, come è possibile vedere nel *Codice 3.39*, è il `RanecuEngine`. Per i miei scopi, non c'era alcuna differenza tra un motore e l'altro, quindi la scelta di questo ultimo è puramente casuale.

```
RunAction::RunAction()
{
  TotalHisto = new Histogram;

  // Choose the Random engine at the start of the program
  //
  HepRandom::setTheEngine(new RanecuEngine); // selection of a random
  HepRandom::setTheSeed(time(0));            // changes the seed of
                                                the random engine
  HepRandom::showEngineStatus();             // shows the actual seed
}
```

Codice 3.39: Scelta del Random Engine all'inizio del run

Nel metodo `BeginOfRunAction()` ho eseguito l'inizializzazione degli istogrammi (vedi *Codice 3.40*), mentre all'interno dell'`EndOfRunAction()` mi sono occupato del loro salvataggio su file (vedi *Codice 3.41*).

```
void RunAction::BeginOfRunAction(const G4Run* aRun)
{

  G4double InitialParticleEnergy = 0.5;   // MeV
  G4int    ParticleYeld          = 50000; // fotoni/MeV
  G4int n = (int)(InitialParticleEnergy * ParticleYeld);

  TotalHisto->InitializeHistograms(n);

}
```

Codice 3.40: Inizializzazione degli istogrammi

```
void RunAction::EndOfRunAction(const G4Run* /*aRun*/)
{

  if(TotalHisto->FileSave())
    G4cout << "Histograms saved!" << G4endl;
  else
    G4cout << "Error: Histograms NOT saved!" << G4endl;

}
```

Codice 3.41: Salvataggio degli istogrammi

E' interessante notare che ho effettuato il salvataggio degli istogrammi esclusivamente al termine della simulazione. L'alternativa sarebbe potuta essere l'effettuare il salvataggio evento per evento, ma ciò avrebbe portato ad un dispendio di risorse, a causa della bassa efficienza nel caso di operazione di I/O, in quanto si sarebbe dovuto accedere al disco fisso decine di migliaia di volte, una volta per ogni evento della simulazione. Una alternativa a tale scelta sarebbe potuta consistere nel salvataggio dei file di testo ogni N eventi, in modo da poterne verificare il contenuto durante il *run* stesso della simulazione.

L'ultima osservazione va effettuata per comprendere la tecnica di esecuzione dei *run*. E' esclusivamente il metodo `beamOn()`, che, una volta invocato, provoca lo start del processo di simulazione. E' possibile invocare questo metodo direttamente all'interno del `main` del programma, oppure lanciarlo tramite *macro*, così come è stato scelto di fare nel nostro software, in modo da poter gestire il comportamento dei *run* durante la fase di esecuzione.

3.1.9 Event Action

Come già accennato in precedenza, la `EventAction` è quella classe che si occupa della gestione del comportamento del programma evento dopo evento. Ho derivato questa classe da `G4UserEventAction`. Deve implementare due metodi obbligatori che vengono richiamati all'inizio ed alla fine di ogni evento, ovvero `BeginOfEventAction()` ed `EndOfEventAction()`, come mostrato nel *Codice 3.42*.

```
class EventAction : public G4UserEventAction
{
  public:
   EventAction();
   ~EventAction();

  public:
    void BeginOfEventAction(const G4Event*);
    void EndOfEventAction(const G4Event*);

  private:
    int SDCID;

};
```

Codice 3.42: Definizione della classe EventAction

Il metodo `BeginOfEventAction()` non è stato utilizzato, in quanto tutte le operazioni utili alla nostra simulazione devono essere effettuate al termine dell'evento. Entriamo quindi nel dettaglio del metodo `EndOfRunAction()`, mostrato nel *Codice 3.43*.

Le prime righe del metodo presentano una operazione piuttosto comune attraverso la quale è possibile estrarre la collezione degli *hit* dall'oggetto `G4Event` passato come parametro.

```
void EventAction::EndOfEventAction(const G4Event* evt)
{

  G4String colName;
  G4SDManager* SDman = G4SDManager::GetSDMpointer();
  SDCID = SDman->GetCollectionID(colName="SDSensor/hitsCollection");

  G4HCofThisEvent* HCE = evt->GetHCofThisEvent();
  SensorHitsCollection* SDHC = 0;

  if(HCE)  {
    SDHC = (SensorHitsCollection*)(HCE->GetHC(SDCID));
  }

  if(SDHC)  {

    G4int n_hit = SDHC->entries();

    G4int posit;
    G4double HitTime;
    G4double MinTime = 10000; // ns

    for(G4int i=0; i<n_hit; i++)  {
      SensorHit* aHit = (*SDHC)[i1];
      HitTime = aHit->GetTime();
      posit = (int)HitTime;
      if(HitTime < MinTime)
          MinTime = HitTime;
    }

    RunAction* usrRunAction;
    const G4RunManager* runManager = G4RunManager::GetRunManager();
    usrRunAction = (RunAction*)runManager->GetUserRunAction();

    usrRunAction->CountHits(n_hit);

    usrRunAction->FirstTimeHit((int)(MinTime*1000));

  }
}
```

Codice 3.43: Implementazione del metodo EndOfEvent()

La prima operazione significativa consiste nell'inserire all'interno della variabile `SDCID` l'ID della collezione di *hit* che si desidera elaborare, nel nostro caso `SDSensor/hitsCollection`, attraverso il metodo `GetCollectionID()` applicato all'oggetto `SDman`, ovvero il gestore dei *Sensitive Detector*. Successivamente viene assegnato all'oggetto `HCE`, di tipo `G4HCofThisEvent`, la collezione di *hits* dell'evento corrente.

Successivamente, ho richiamato i due metodi `CountHits()` e `FirstTimeHit()`, creati in precedenza nella classe `RunAction`, per la gestione dei due istogrammi, passando rispettivamente i parametri `n_hit` e `MinTime`, calcolati come segue:

- il metodo `entries()` applicato sull'oggetto `SDHC` restituisce il numero di hit avvenuti all'interno dell'evento, salvato all'interno della variabile `n_hit`.
- il ciclo *for* permette la visita di tutti gli *hit* della collezione, l'estrazione del tempo di impatto con il metodo `GetTime()` ed il calcolo del tempo minimo, salvato dentro la variabile `MinTime`.

Importante notare come abbia applicato entrambi i metodi sull'oggetto `usrRunAction` che rappresenta il *run* corrente. Questo è stato possibile tramite il prelievo del puntatore a `RunAction` attraverso il metodo `GetUserRunAction()`.

3.2 Il main del programma

Dopo aver illustrato in dettaglio le funzionalità di ogni classe, volgiamo la nostra attenzione verso il `main()` del programma, utilizzato esclusivamente per la creazione degli oggetti, di cui sopra le classi, e per l'avvio della simulazione.

Il `main` di qualsiasi programma Geant4 è pressoché identico, in quanto tutte le operazioni per la gestione della simulazione vanno effettuate dentro le classe.

Diamo quindi una rapida visione di insieme delle varie porzioni di codice che compongono il `main()`.

```
G4RunManager* runManager = new G4RunManager;
```

Codice 3.44: Creazione dell'oggetto G4RunManager

`G4RunManager` è la classe per il controllo dei *run* ed è l'unica all'interno del kernel di Geant4 che necessita della creazione di un oggetto all'interno del `main()`.

```
DetectorConstruction* detector = new DetectorConstruction;
runManager->SetUserInitialization(detector);

G4VUserPhysicsList* physics = new PhysicsList;
runManager->SetUserInitialization(physics);
```

Codice 3.45: Inizializzazione delle mandatory user classes

`DetectorConstruction` e `PhysicsList` sono le due classi descritte in precedenza, quelle che ho creato per il settaggio delle geometrie, dei materiali, delle superfici e delle interazione fisiche delle particelle con la materia.

```
#ifdef G4VIS_USE
   G4VisManager* visManager = new G4VisExecutive;
   visManager->Initialize();
#endif
```

Codice 3.46: Creazione dell'oggetto G4VisManager

La classe `G4VisManager` crea e gestisce la visualizzazione delle scene, dei modelli, delle geometrie e dei tracciati. La creazione di un oggetto appartenente a questa classe non implica l'obbligatorietà dell'utilizzo della versione grafica della simulazione, ma è possibile definire tale modalità direttamente in fase di esecuzione attraverso l'utilizzo delle *macro*.

```
G4VUserPrimaryGeneratorAction* gen_action =
                                  new PrimaryGeneratorAction(detector);
runManager->SetUserAction(gen_action);

G4UserRunAction* run_action = new RunAction;
runManager->SetUserAction(run_action);

G4UserEventAction* event_action = new EventAction;
runManager->SetUserAction(event_action);

runManager->Initialize();
```

Codice 3.47: Creazione degli oggetti per le classi di "azione"

Durante la creazione del nostro software di simulazione è stato indispensabile l'utilizzo della visualizzazione grafica esclusivamente in fase di sviluppo della geometria, in modo da verificare la corrispondenza tra il dispositivo creato e quello progettato. Invece durante le vere simulazioni, la modalità grafica è stata disattivata in modo da aggravare il meno possibile sul carico di lavoro alla CPU, con il conseguente abbreviamento dei tempi di simulazione.

Nel *Codice 3.47* è mostrata la creazione degli oggetti appartenenti alle classi descritte nei paragrafi precedenti e quindi l'invocazione del metodo `SetUserAction()` sull'oggetto `runManager`. Infine viene eseguita l'inizializzazione del kernel di Geant4 attraverso l'invocazione del metodo `Initialize()`.

```
// Get the pointer to the User Interface manager
//
G4UImanager* UI = G4UImanager::GetUIpointer();

// G4UIterminal is a (dumb) terminal
//
G4UIsession* session = 0;

#ifdef G4UI_USE_TCSH
    session = new G4UIterminal(new G4UItcsh);
#else
    session = new G4UIterminal();
#endif
```

Codice 3.48: Creazione della sessione User Interface

La porzione di *Codice 3.48* viene utilizzata per l'inizializzazione della sessione d'interfaccia utente per l'immissione dei comandi da *shell*.

Importante evidenziare come esista una sintassi sviluppata appositamente per Geant4 utilizzata per l'immissione di comandi via *shell*, utilizzati ad esempio per

l'impostazione dell'energia della particella, del tipo di particella, della direzione, per il lancio del *run* della simulazione, per l'impostazione delle opzioni di visualizzazione grafica, come zoom, rotazione, posizione, etc.

Tali comandi sono proprio quelli inseriti all'interno delle *macro*, ovvero dei file di testo con una sequenza di istruzioni scritte con l'apposita sintassi.

```
UI->ApplyCommand("/control/execute sensore.mac");
session->SessionStart();
delete session;
```

Codice 3.49: Lancio della macro ed inizio della sessione

Nel *Codice 3.49* è mostrato il metodo `ApplyCommand()` utilizzato per l'esecuzione del comando `/control/execute` con parametro `sensore.mac`, ovvero il nome della *macro* salvata all'interno della stessa cartella dell'eseguibile.

```
#ifdef G4VIS_USE
   delete visManager;
#endif

delete runManager;

return 0;
```

Codice 3.50: Deallocazione delle risorse

Infine nel *Codice 3.50* viene effettuata la deallocazione delle risorse, sia del `visManager` che del `runManager`.

3.3 La macro

Le macro di Geant4 sono delle sequenze di istruzioni nella sintassi di Geant4 salvate all'interno di un file di testo. Tali istruzioni hanno il compito di modificare il comportamento del software a *runtime*, senza necessità di ricompilare il sorgente.

Vediamo ora in dettaglio la macro che ho utilizzato per la nostra simulazione, chiamata `sensore.mac`. Tale macro si suddivide in due parti, di cui la prima gestisce la visualizzazione grafica, mentre la seconda imposta le proprietà della particella e da inizio alla simulazione.

```
/vis/scene/create
```

Macro 3.1: Creazione della scena per la visualizzazione grafica

La prima istruzione, mostrata nella *Macro 3.1*, viene utilizzata per la creazione di una nuova scena, all'interno della quale saranno successivamente mostrate le geometrie e i tracciati delle particelle.

```
/vis/open OGLIX
#/vis/open RayTracer
#/vis/open OGLSX
#/vis/open DAWNFILE
#/vis/open HepRepFile
#/vis/open VRML2FILE
```

Macro 3.2: Modalità di output grafico

L'istruzione `/vis/open` ha lo scopo di impostare la modalità di output. Come mostrato nella *Macro 3.2* viene utilizzato il driver OPENGL, oppure, spostando

opportunamente il simbolo # di commento, è possibile impostare una modalità alternativa per permettere l'utilizzo di altri driver grafici, oppure l'esportazione della geometria in formati importabili con software di terze parti.

```
/vis/viewer/set/viewpointThetaPhi 90.0 0.0 deg
/vis/viewer/zoom 1.
/vis/viewer/flush
```

Macro 3.3: Settaggi di visualizzazione

Attraverso l'utilizzo del set di comandi all'interno di `/vis/viewer/` è possibile settare l'angolo di visuale e lo zoom, come mostrato nella *Macro 3.3*.

```
/tracking/storeTrajectory 1
/tracking/verbose 0
/vis/scene/add/trajectories
/vis/scene/endOfEventAction accumulate
```

Macro 3.4: Memorizzazione e visualizzazione delle traiettorie delle particelle

Successivamente viene indicato al software di memorizzare le traiettorie delle particelle per una successiva visualizzazione grafica, alla fine dell'evento. Se l'opzione `/tracking/storeTrajectory 1`, a causa dell'eccessivo numero di tracciati da memorizzare, provocasse l'errore di core dumping in fase di esecuzione, sarebbe possibile disabilitarla impostando il valore a `0`.

Infine con le istruzioni mostrate nella *Macro 3.5* si imposta il tipo di particella e l'energia e quindi si da inizio alla simulazione con l'istruzione `/run/beamOn 10000` che provocherà il lancio di 10000 eventi.

```
/gun/particle gamma
/gun/energy 511 keV
/run/beamOn 10000
```

Macro 3.5: Inizio della simulazione

3.4 Un esempio di run con visualizzazione comandi

Le stesse istruzioni mostrate nella *macro* del paragrafo precedente è possibile immetterle come input in fase di esecuzione, attraverso la *shell*, utilizzata sia per l'inserimento di comandi che per l'eventuale output testuale del programma.

In *Figura 3.4* è mostrato un semplice esempio di output per un run di test che ho effettuato con 20 eventi.

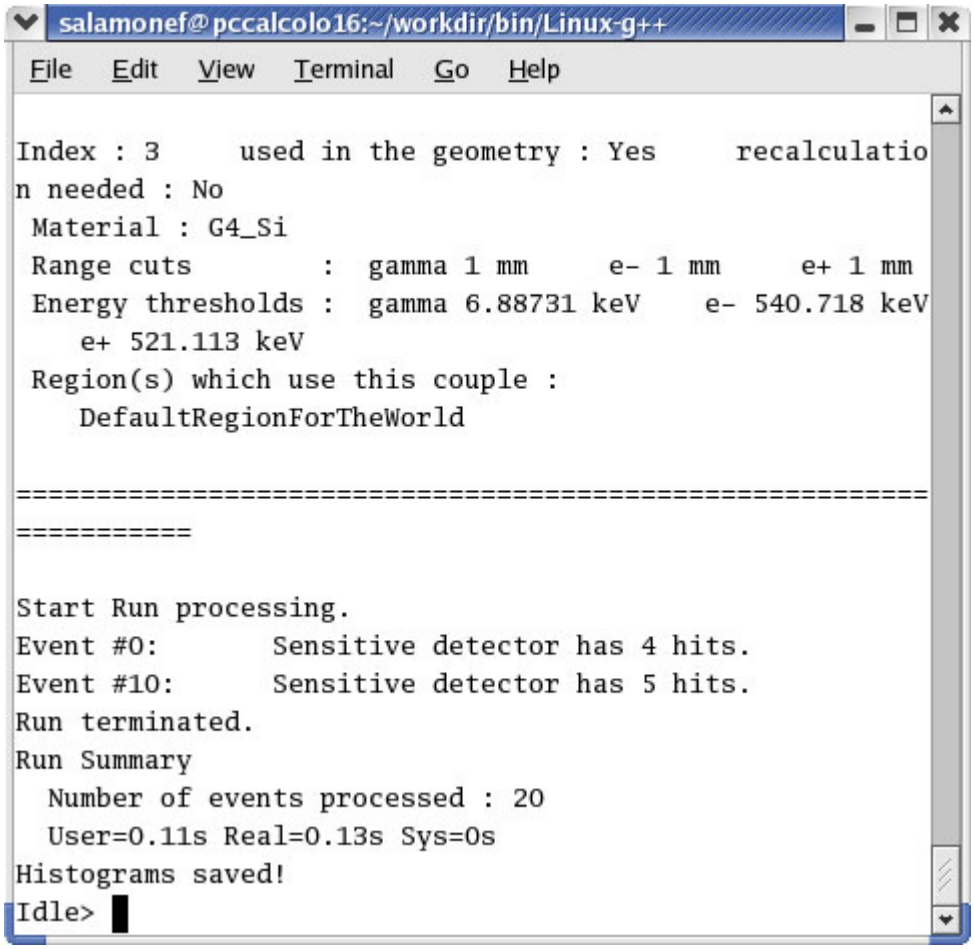


Figura 3.4: Shell testuale di Geant4

3.5 Un esempio di run con visualizzazione grafica

Dopo aver compilato il software con lo strumento *gmake*, è possibile lanciare l'eseguibile generato e vedere la geometria creata, che si presenterà così come mostrato in *Figura 3.5*.

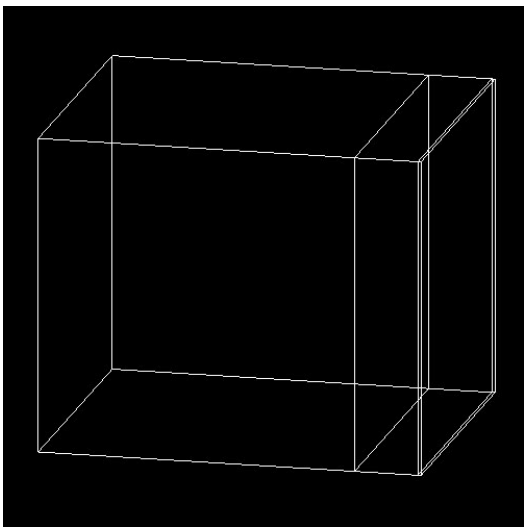


Figura 3.5: Geometria creata in Geant4

Successivamente è possibile dare il via al *run* della simulazione con il comando `/run/beamOn`, così come già illustrato nei paragrafi precedenti. Nelle figure seguenti sono indicate in verde le traiettorie dei fotoni prodotti dal processo di scintillazione. Come è possibile notare in

Figura 3.7, lo *shoot* di appena 20 particelle genera migliaia di fotoni con altrettante riflessioni all'interno del dispositivo. Ecco il motivo per cui si preferisce disabilitare la

visualizzazione grafica durante il "vero" *run,* in cui viene eseguito lo *shoot* di ben 10000 particelle.

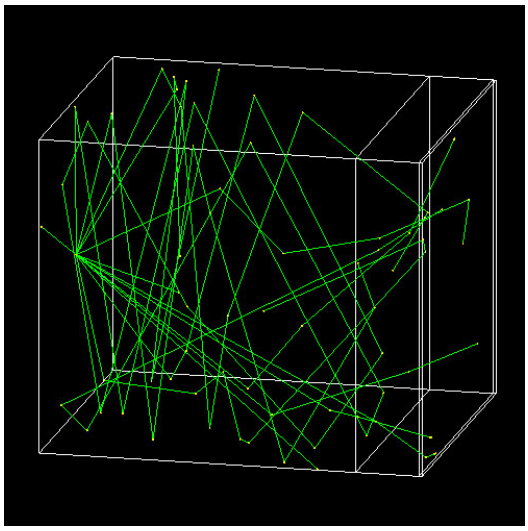


Figura 3.6: Run con 1 evento

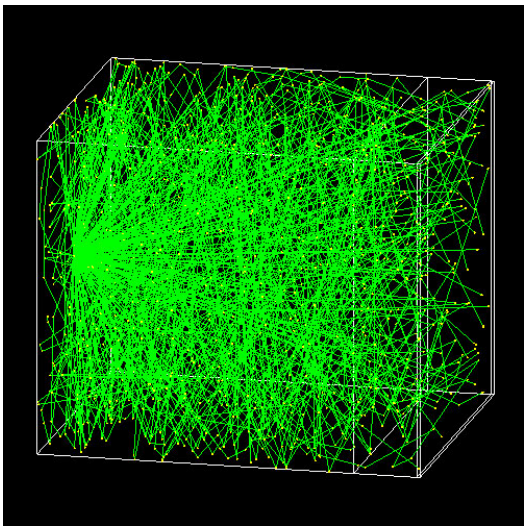


Figura 3.7: Run con 20 eventi

Il risultato di tale simulazione consiste nella produzione due file di testo, creati nella stessa cartella dell'eseguibile, che rappresentano gli istogrammi, da importare e trattare con Microsoft Excel, come mostrato nel prossimo capitolo.

4. Il run delle simulazioni

In questo capitolo verranno esaminati i dettagli delle differenti simulazioni eseguite con il Geant4, con particolare attenzione alle differenze del codice C++ tra una simulazione e l'altra, ma soprattutto con i commenti, le tabelle, i grafici e le osservazioni sui risultati ottenuti. Verranno quindi omessi i dettagli di funzionamento del software di simulazione, già ampiamente illustrati nel capitolo precedente.

4.1 Test del simulatore e verifiche di attendibilità

Lo sviluppo del codice mostrato nel capitolo precedente ha avuto lo scopo di riprodurre il comportamento noto di un sensore di cui se ne conoscono sperimentalmente le caratteristiche, in modo tale da poterne confrontare i risultati con quelli noti e quindi verificare l'attendibilità del software creato.

A tal proposito il test è stato effettuato irradiando protoni. La simulazione di test mi ha dato quindi la conferma che il lavoro svolto nel capitolo precedente è corretto e che quindi il software è affidabile. In tal modo ho avuto la possibilità per poter lanciare il *run* delle simulazioni con altri tipi di particelle (gamma da 511 keV) e successivamente con altre geometrie, certo che i risultati sarebbero stati attendibili.

Innanzitutto ho lanciato il simulatore in modalità grafica per verificare che la geometria creata da codice coincidesse effettivamente con la geometria desiderata. Successivamente, attraverso il comando `run/beamOn` ho dato inizio ad una simulazione grafica di test, per verificare che le pareti fossero riflettenti, mentre il sensore di silicio fosse assorbente.

In *Figura 4.1* è mostrata una visuale dall'alto del dispositivo. E' chiaramente visibile la suddivisione nelle tre parti, composte dal materiale scintillante, dal vetro e dal silicio. Le linee verdi indicano i tracciati che hanno seguito i fotoni luminosi generati dal processo di scintillazione, mentre i puntini gialli indicano le riflessioni o gli assorbimenti dei fotoni.

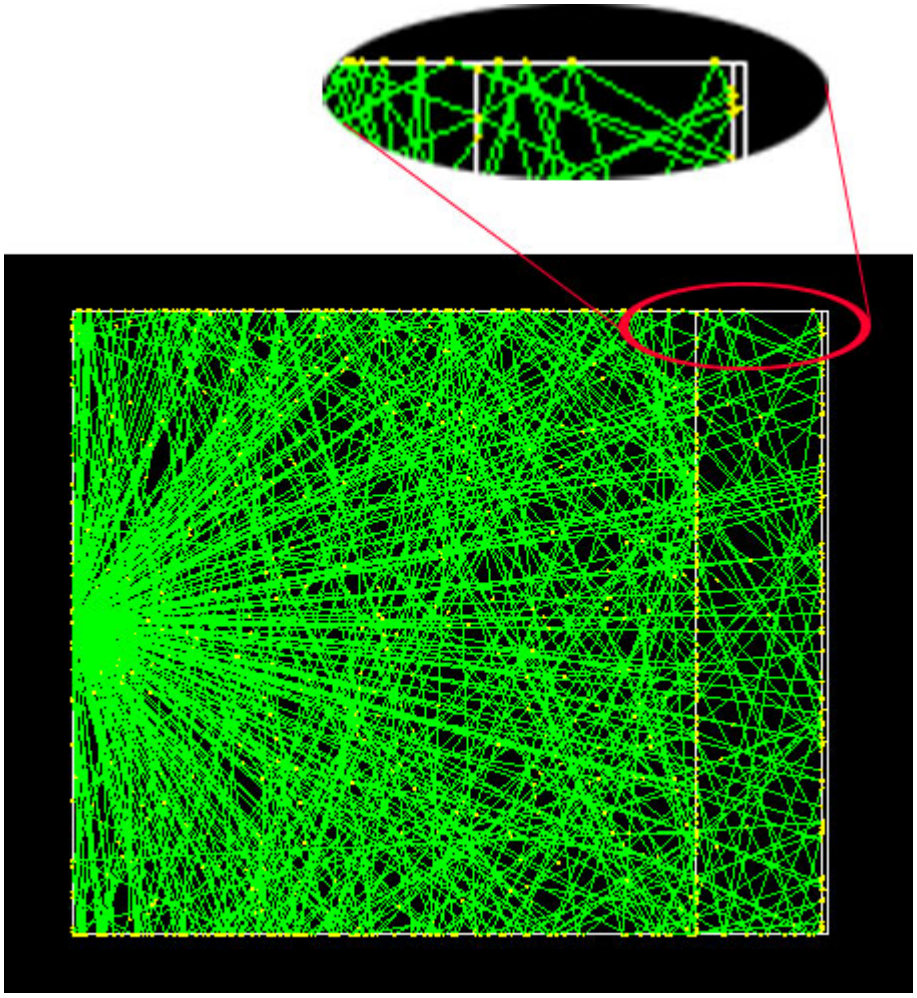


Figura 4.1: Run grafico di test con dettaglio zoommato

Dopo aver testato quindi il corretto funzionamento del software con pochi eventi, ho disabilitato la versione grafica e lanciato la versione testuale, utilizzata da questo momento in poi, in quanto permette un impiego ottimale delle risorse di sistema totalmente mirato verso il vero aspetto simulativo e non verso quello di contorno della visualizzazione grafica.

Il comando che ho utilizzato per dare il via alla simulazione è stato `run/beamOn 10000` per il lancio di ben 10.000 eventi. Ho scelto questo valore dopo aver valutato,

con alcuni *run* di test, la durata della simulazione. Inoltre è buon valore che riesce a fornire una chiare idea statistica della risposta del sensore.

Il *run* di questa simulazione è durato 2 ore circa (macchina dedicata) ed ha generato, come risultato finale, così come illustrato ampiamente nel capitolo precedente, due file di testo, come quelli mostrati in *Figura 4.2*, in cui ogni riga rappresenta una classe dell'istogramma.

I due file, chiamati *CsI_FirstHitHistogram.txt* e *CsI_CountHistogram.txt*, rappresentano rispettivamente l'istogramma dei tempi di impatto del primo fotone e l'istogramma del conteggio del numero di fotoni rivelati. Nel primo istogramma ogni riga del file di testo rappresenta l'intervallo di tempo di 1 ps, mentre nel secondo ogni riga rappresenta il numero di fotoni rivelati.

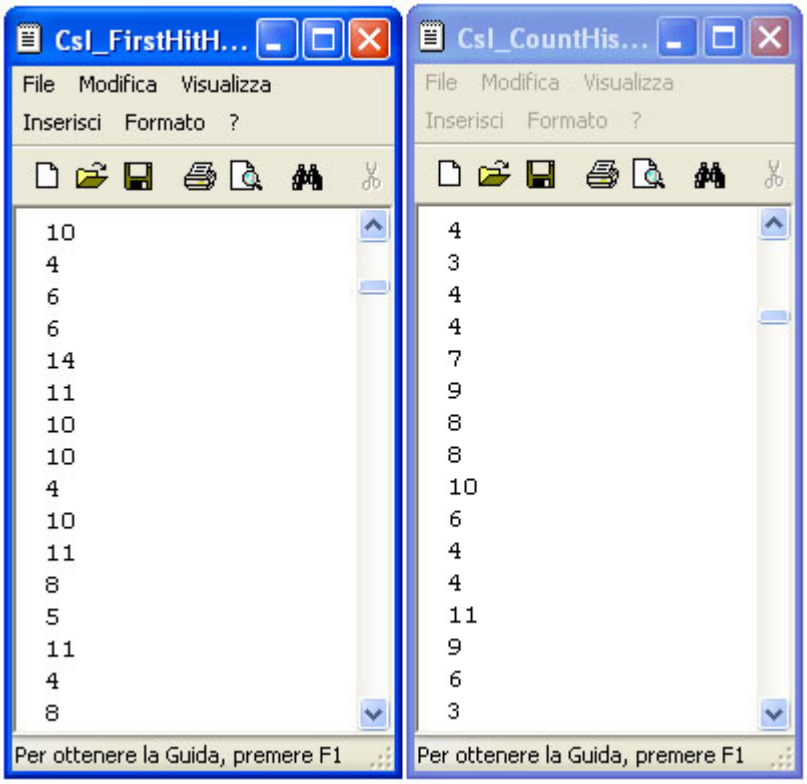


Figura 4.2: File di testo generati dal run della simulazione

Per rendere meglio l'idea concentriamoci innanzitutto su *CsI_FirstHitHistogram.txt*. I numeri 10, 4, 6, 6, 14, etc, stanno a significare che negli istanti (picosecondi) corrispondenti alla riga i-esima si sono verificati 10, 4, 6, 6, 14, etc, impatti con il rivelatore.

I valori presenti, invece, nel file di testo *CsI_CountHistogram.txt*, come ad esempio 4, 3, 4, 4, 7, etc, stanno a significare che sono stati rivelati *n* fotoni per 4, 3, 4, 4, 7, etc, volte, dove *n* è il numero della riga.

In *Figura 4.3* è riportato lo spettro ottenuto attraverso l'importazione ed il plot, tramite Microsoft Excel, del file di testo *CsI_CountHistogram.txt*. I punti blu indicano i dati raccolti dalla simulazione, mentre la linea in rosso rappresenta il *fit*, ovvero la migliore funzione che approssima la distribuzione dei dati.

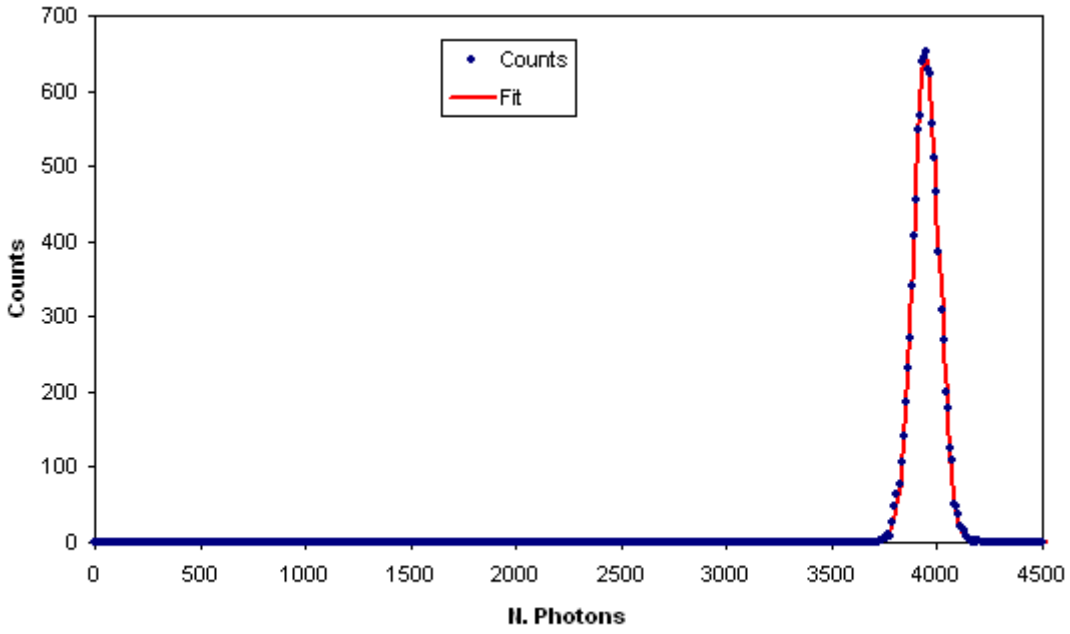


Figura 4.3: Grafico del conteggio di fotoni per evento

In tal caso la curva è una *gaussiana* con equazione:

$$f(x) = a + b \cdot e^{-\frac{1}{2}\left(\frac{x-c}{d}\right)^2}$$

Dove:

- *a* rappresenta l'altezza di riferimento della gaussiana, nel nostro caso a=0 sempre.
- *b* è l'*ampiezza* ed indica l'altezza del picco della gaussiana.
- *c* è il *centroide* della gaussiana.
- *d* è *sigma* ed indica l'apertura della gaussiana. Più è grande sigma, più le code sono lunghe.

Il metodo che ho utilizzato per il ottenere il *fit* e' il seguente:

- Scelgo una forma funzionale per (tentare di) riprodurre i dati, che dipende da alcuni parametri, in questo caso la gaussiana.
- Assegno un valore ai parametri e calcolo la funzione.
- Calcolo il *chi_quadro*, cioè la somma dei quadrati degli scarti tra dati e funzione, punto per punto.
- Modifico i parametri e ripeto la procedura fino a che trovo il set di parametri che minimizza il *chi_quadro*.

Il *Solver* (risolutore) di Microsoft Excel utilizza il metodo di Newton per trovare rapidamente il minimo. Tale metodo altro non e' che una strategia per variare i parametri in maniera furba, in modo da convergere rapidamente verso il valore minimo

della grandezza da minimizzare, attraverso il calcolo di una serie di derivate parziali delle funzioni numeriche in esame.

I valori *b*, *c* e *d* forniscono quindi i parametri per ricreare la gaussiana e sono proprio quelli che ho inserito, in seguito, all'interno di una tabella per permetterne il confronto con quelli di altre simulazioni.

Tale curva ha una importanza fondamentale per la stima della *risoluzione* del sensore e per il calcolo di una valore di efficienza, chiamato *efficienza di fotopicco*.

La *risoluzione* è un parametro standard di tutti gli strumenti di misura. In questo caso è calcolata come rapporto tra sigma e la media e rappresenta la minima variazione apprezzabile del numero di fotoni che possono essere rivelati.

$$res = \frac{d}{c}$$

E' facile intuire che tanto più sigma è piccola e tanto più è grande la media, quanto più si ottiene una risoluzione migliore.

Discorso a parte va effettuato invece per l'*efficienza di fotopicco*. Innanzitutto è necessario precisare che non ha senso parlare di questo parametro nel caso in cui le particelle irradiate siano protoni, per i motivi spiegati più avanti. Per questo consideriamo il *fit* di una tra le altre simulazioni effettuate, in cui le particelle irradiate sono raggi gamma, per riuscire ad apprezzare gli effetti che stiamo introducendo.

Basti osservare il *fit* in *Figura 4.4* per comprendere come esistano due effetti predominanti nella rivelazione dei fotoni: l'effetto Compton e l'effetto fotoelettrico. Invece l'effetto di creazione di coppie, citato in precedenza nel Capitolo 2, non viene

considerato perché l'energia dei fotoni (511 keV) è troppo bassa per scatenarne la produzione.

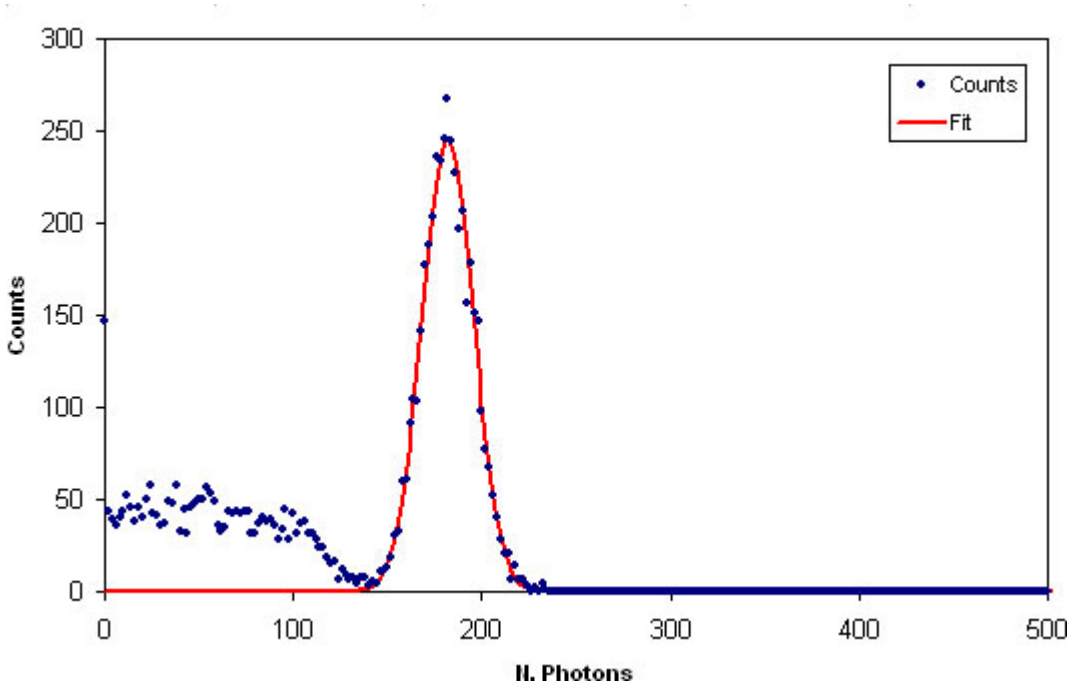


Figura 4.4: Spalla di Compton e picco

L'*effetto Compton* causa la creazione della cosiddetta *spalla di Compton*, visibile nel nostro caso nell'intervallo compreso tra 0 e 150, circa. Questo è un effetto negativo ai fini del rivelamento, poiché l'effetto Compton causa la produzione di elettroni, che successivamente genereranno luce di scintillazione, ad una energia inferiore ai 511 keV. Questo perché l'energia del fotone incidente non viene completamente rilasciata all'elettrone, ma si trasferisce solo in parte, mentre il resto rimane dentro il fotone, che avrà quindi un'energia inferiore ai 511 keV. Questo fotone a sua volta potrà causare un ulteriore scattering Compton. Se il materiale è abbastanza denso ed abbastanza spesso, è piuttosto probabile che da un fotone che ha appena fatto Compton si scaturiscano altri

scattering e che quindi tutta la sua energia venga esaurita all'interno dello scintillatore, provocando un effetto globale di "scomparsa" del fotone incidente e trasferimento di tutta l'energia ad elettroni, i quali, rallentando, genereranno luce di scintillazione.

Se invece il materiale non è abbastanza denso o spesso, c'è un alta probabilità che il fotone, dopo aver fatto Compton, esca dallo scintillatore senza causare ulteriori interazioni. In tal caso il processo di scintillazione produrrà meno fotoni di quelli prodotti nel caso del trasferimento di tutti i 511 keV di energia, in quanto la sua energia sarà inferiore.

Questo porterà ad un conteggio più basso del numero di fotoni catturati in alcuni determinati eventi e quindi di conseguenza la spalla di Compton visibile in figura.

L'*effetto fotoelettrico* causa invece la totale cessione dell'energia del fotone incidente sull'elettrone che, rallentando, darà inizio al processo di scintillazione. I fotoni prodotti sono quindi, in media, tutti quelli che l'efficienza dello scintillatore permette di produrre. Ovviamente negli eventi in cui si verifica l'effetto fotoelettrico ci sarà un conteggio di fotoni che cadrà all'interno della gaussiana.

L'*efficienza di fotopicco*, quindi, indica proprio il numero di eventi per i quali il numero di fotoni raccolti è all'interno della gaussiana.

$$ppe = \frac{a \cdot c \cdot \sqrt{2 \cdot \pi}}{n}$$

Dove n è il numero di eventi, nel nostro caso 10.000.

Nel caso di protoni irradiati su un cubo 5x5x5 cm di materiale scintillante, l'efficienza di fotopicco è circa 1, a meno di leggere perdite. Questo perché tutti i

protoni che entrano all'interno dello scintillatore creeranno luce di scintillazione ad una energia pari a quella del protone stesso, ovvero 500 keV.

Quando invece, al posto del protone, si irradia un fotone, valgono le considerazioni fatte sopra e l'efficienza di fotopicco varia in funzione di un parametro, chiamato *sezione d'urto*. Tale parametro varia approssimativamente in funzione di Z^4 o Z^5, della densità e dello spessore del materiale, dove Z è il numero atomico medio del composto.

Da queste considerazioni e vista la *Figura 4.3* è immediato rendersi conto che il cubo 5x5x5 cm, irradiato con protoni, si comporta, sotto questo punto di vista, nel modo desiderato, in quanto non è presente la spalla di Compton. Questa è già la prima verifica di attendibilità che mostra come il software creato sia attendibile sotto questo punto di vista.

Analizziamo ora invece il plot del file di testo *CsI_FirstHitHistogram.txt*. Anche in questo caso i punti blu rappresentano i dati raccolti mentre la linea in rosso il *fit* che approssima meglio la distribuzione dei punti.

Questo istogramma rappresenta il tempo trascorso tra lo *shoot* del protone ed il primo fotone gamma rilevato dal sensore di silicio.

Per *fittare* questi valori, non è stato possibile utilizzare una gaussiana, così come nel precedente istogramma, ma invece ho adoperato una funzione per metà gaussiana e per metà esponenziale.

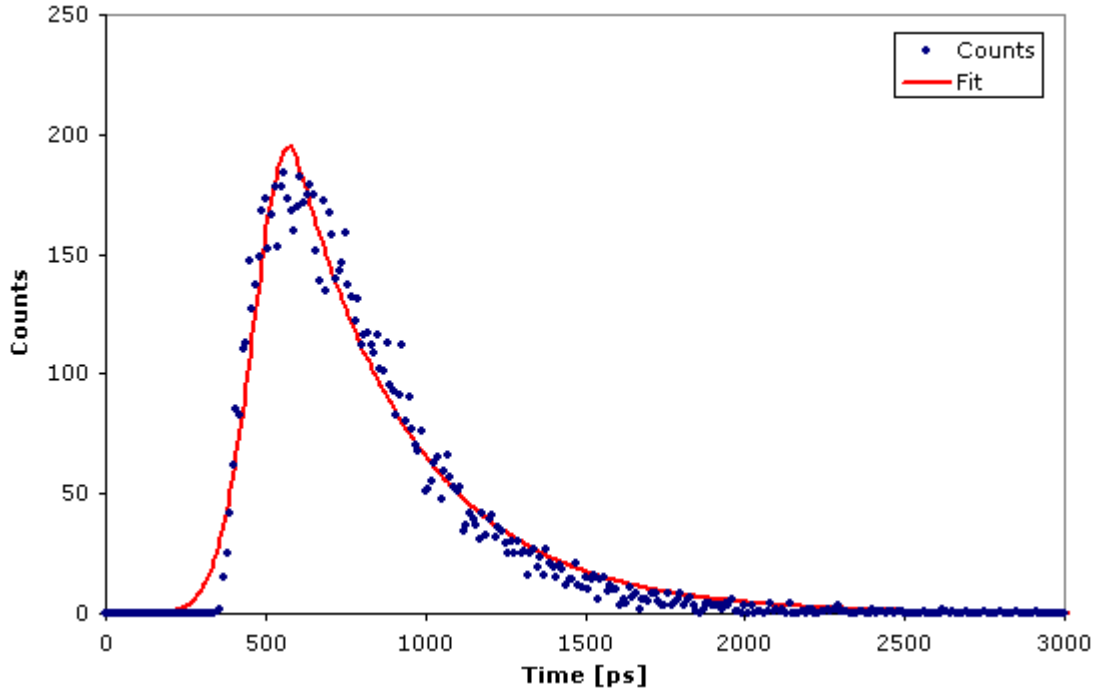


Figura 4.5: Risposta temporale del dispositivo simulato

$$f(x) = \begin{cases} a + b \cdot e^{-\frac{1}{2}\left(\frac{x-c}{d}\right)^2} & \text{se } x < f \\ a + b \cdot e^{-\frac{1}{2}\left(\frac{c-f}{d}\right)^2} \cdot e^{-\frac{x-f}{\tau}} & \text{se } x > f \end{cases}$$

Dove:

- *a* rappresenta l'*altezza di riferimento* della gaussiana, nel nostro caso a=0 sempre.
- *b* è l'*ampiezza* ed indica l'altezza del picco della gaussiana.
- *c* è il *centroide* della gaussiana.
- *d* è *sigma* ed indica l'apertura della gaussiana. Più è grande sigma, più le code sono lunghe.

- *f* è l'*offset* ed indica il punto di giunzione tra la gaussiana e la funzione esponenziale.
- τ è il parametro caratteristico di ogni funzione esponenziale

Da tale istogramma si riesce quindi ad interpretare la risposta nel tempo del dispositivo simulato. In questo caso il picco è intorno ai 600 ps, ma la coda si protrae fin oltre i 2500 ps. I due parametri di maggior interesse per valutare le prestazioni sono sigma e τ, in quanto il primo indica l'apertura della gaussiana, cioè quanto è veloce la risposta in salita, mentre il secondo indica la lunghezza della coda e quindi quanto è veloce la risposta in discesa. E' stato anche utile calcolare una media aritmetica tra i due valori, così da poterla confrontare con i risultati delle altre simulazioni.

Tutti i valori citati finora sono stati riportati in *Tabella 4.1* nel caso delle sei simulazioni di test, di cui le prime tre con irradiamento di protoni, le ultime tre con raggi gamma.

Scintillator		Time						N.photons				
scintillator	particle	offset	ampl.	avg.	sigma	tau	(tau + sigma) / 2	ampl.	avg.	sigma	photo peak eff.	resol.
CsI	p	607,2	195,0	573,7	112,5	377,9	245,2	646,9	3948,4	61,4	0,995	0,016
BC408	p	349,4	492,3	338,7	14,4	19,4	16,9	273,6	848,4	29,2	1,000	0,034
LSO	p	449,2	148,1	429,6	40,1	82,2	61,2	559,4	1313,2	35,5	0,996	0,027
CsI	γ	692,1	119,2	614,5	188,4	457,2	322,8	401,4	4004,2	61,4	0,617	0,015
BC408	γ											
LSO	γ	489,2	104,1	433,0	66,9	79,5	73,2	502,4	1333,6	37,0	0,933	0,028

Tabella 4.1: Risultati delle simulazioni di test

Gli scintillatori testati sono il CsI, il BC408 e l'LSO. Il BC408 è un materiale plastico non utilizzabile nei sensori per rivelazione di raggi gamma da 511 keV e, nonostante la sua veloce risposta nel tempo (solo 16,9 ps di media), ha una bassa densità che lo rende inutilizzabile per i nostri scopi. Infatti, i risultati degli istogrammi non hanno portato ad una generazione di *fit* apprezzabili, a differenza invece degli altri casi. Questo è il motivo per cui la riga relativa al BC408 è vuota. Tale comportamento è ben conosciuto e quindi l'impossibilità della rivelazione, nel caso del BC408, ha dato un'altra conferma sul corretto funzionamento del software. Il motivo per cui però ho deciso di testare il BC408 è anche legato al suo impiego in una configurazione avanzata, illustrata nei prossimi paragrafi, in cui non si fa uso di questo materiale per far scaturire il processo di scintillazione, ma esclusivamente per far avanzare i fotoni luminosi già generati all'interno del dispositivo. Maggiori dettagli in seguito.

E' possibile anche notare che l'efficienza di fotopicco delle simulazioni con i protoni è circa 1 in tutti e tre i casi, per il motivo spiegato in precedenza. Altra conferma della bontà del software.

Nelle pagine seguenti sono mostrati i confronti, tramite grafici, dei risultati ottenuti per queste prime sei simulazioni di test.

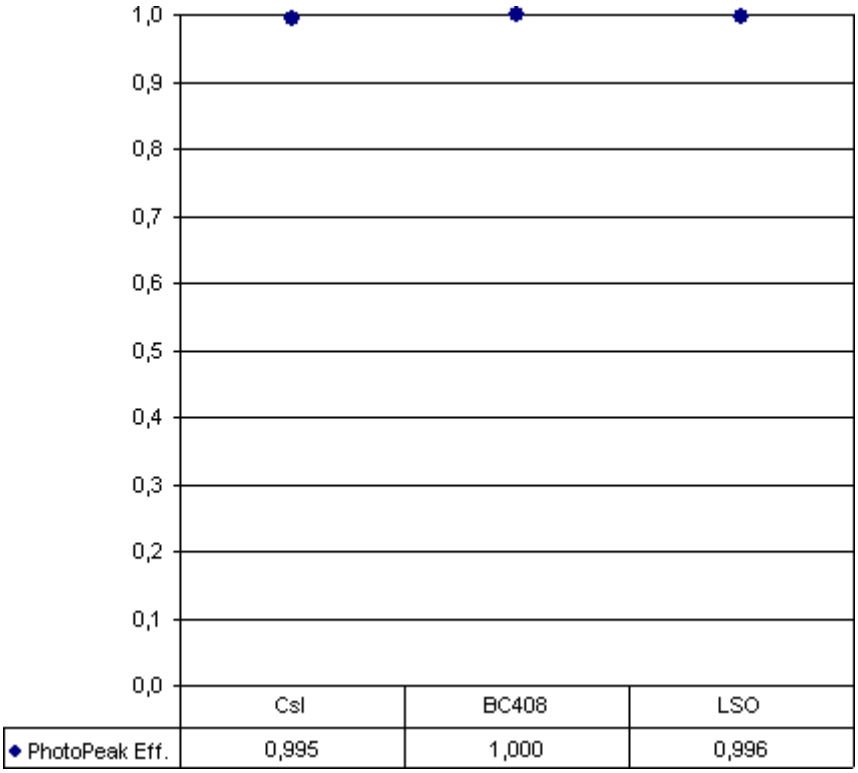


Figura 4.6: Efficienza di fotopicco per la simulazione con i protoni nel cubo 5x5x5 cm

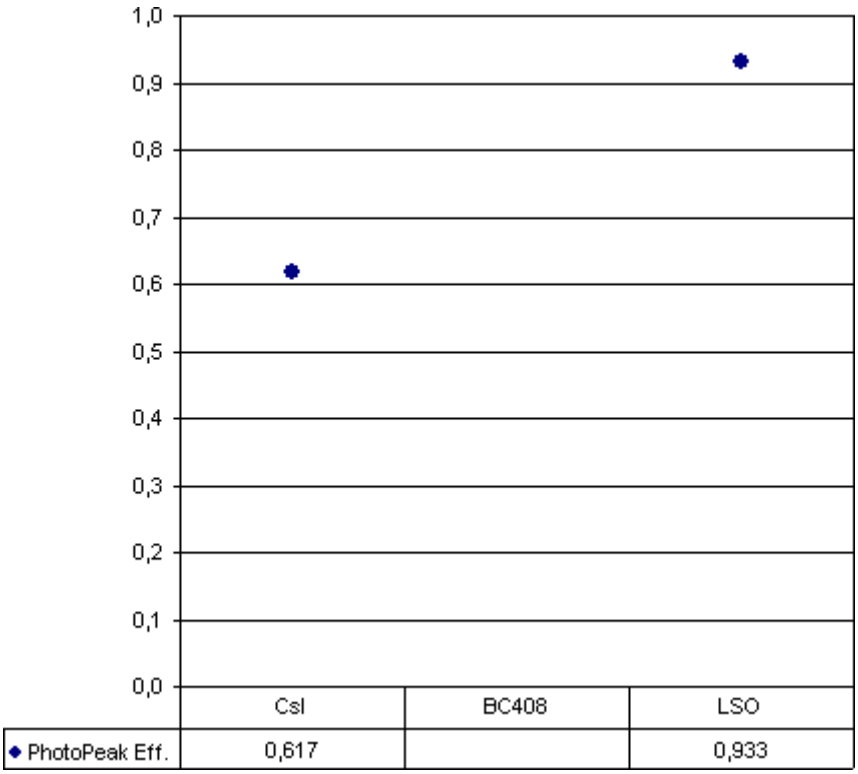


Figura 4.7: Efficienza di fotopicco per la simulazione con i gamma nel cubo 5x5x5 cm

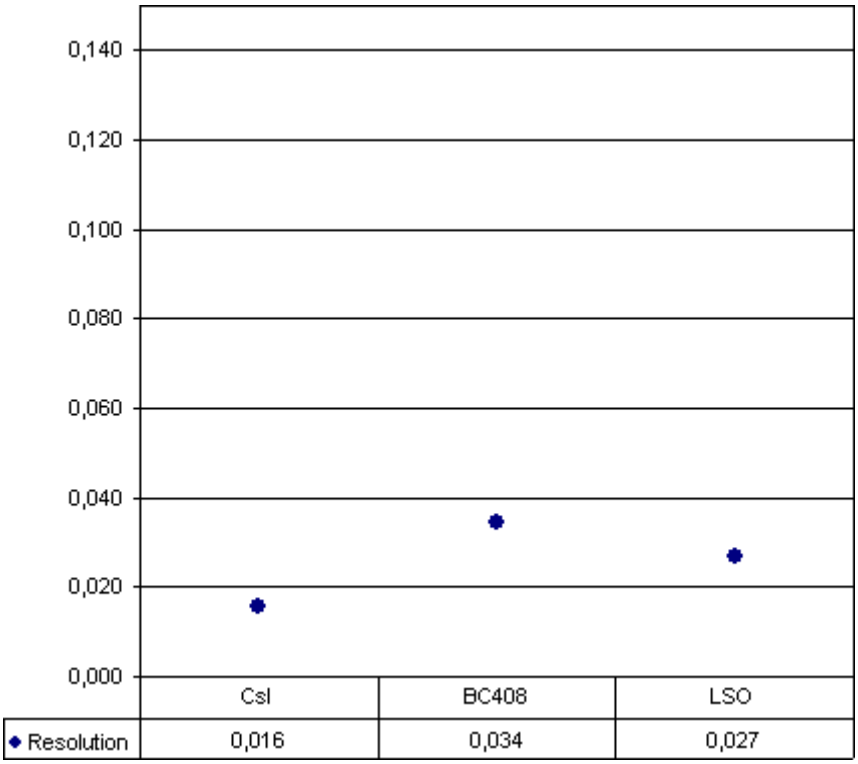


Figura 4.8: Risoluzione energetica per la simulazione con i protoni nel cubo 5x5x5 cm

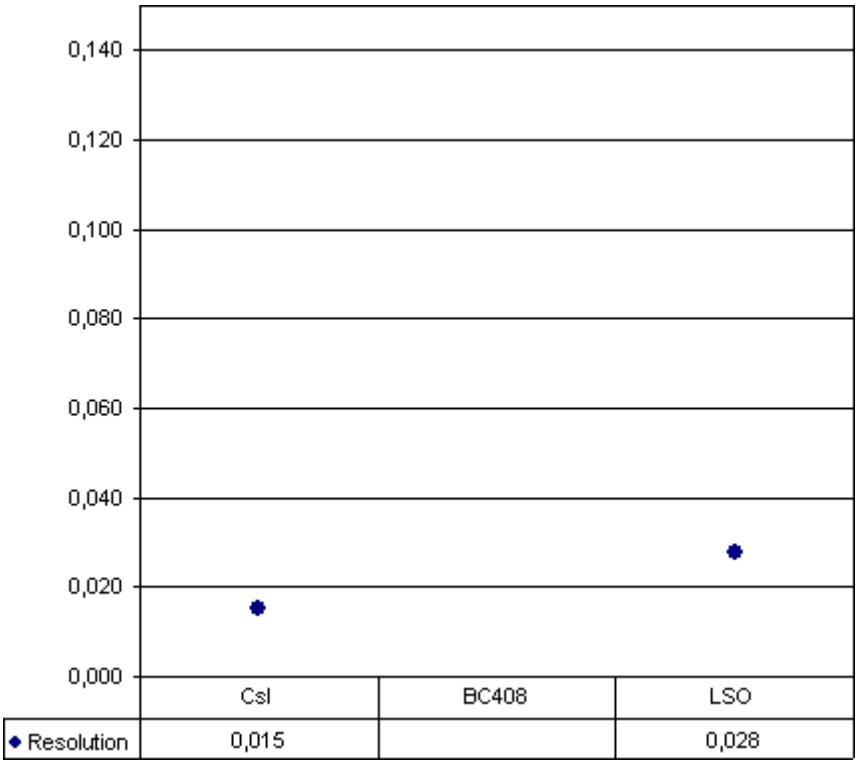
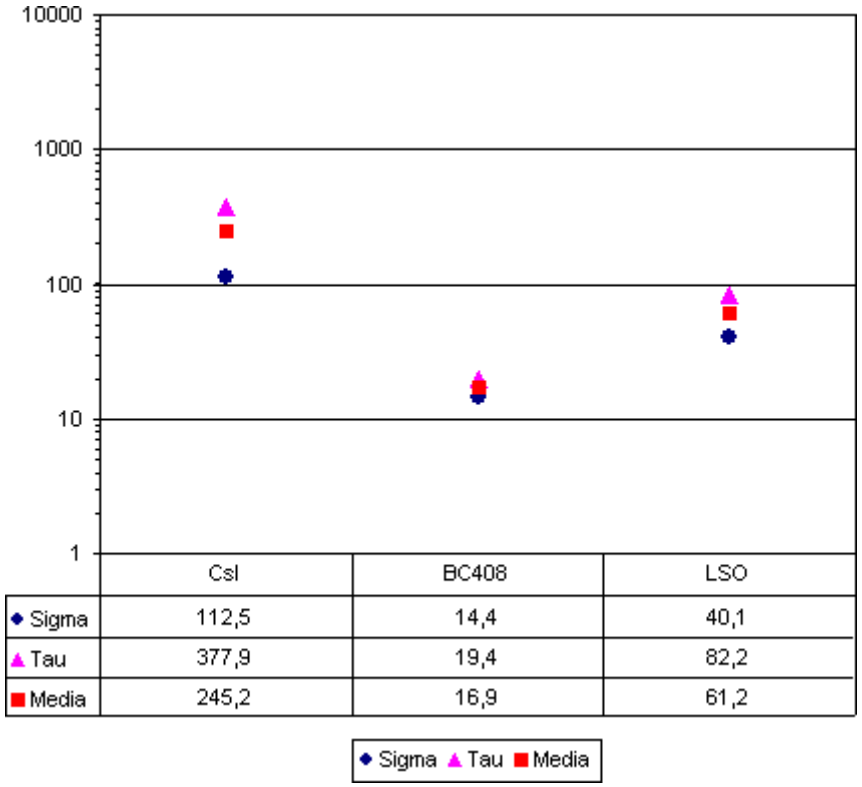
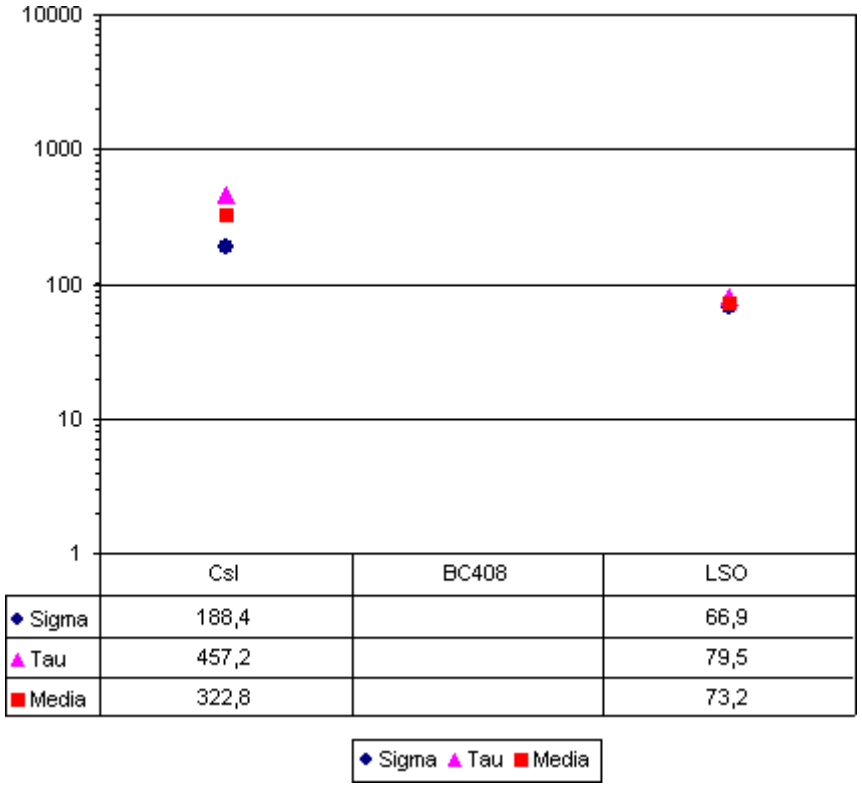


Figura 4.9: Risoluzione energetica per la simulazione con i gamma nel cubo 5x5x5 cm



	CsI	BC408	LSO
Sigma	112,5	14,4	40,1
Tau	377,9	19,4	82,2
Media	245,2	16,9	61,2

Figura 4.10: Risposta nel tempo per la simulazione con i protoni nel cubo 5x5x5 cm



	CsI	BC408	LSO
Sigma	188,4		66,9
Tau	457,2		79,5
Media	322,8		73,2

Figura 4.11: Risposta nel tempo per la simulazione con i gamma nel cubo 5x5x5 cm

4.2 Simulazioni con geometria a bastoncino

Dopo aver testato ed appurato che il software creato simuli correttamente tutti i processi fisici e le interazioni tra particella e materia, ho creato una nuova geometria, più simile a quelle dei sensori realmente utilizzati per le PET, che chiameremo *a bastoncino*. Anch'essa, ovviamente, è composta da uno scintillatore, da uno strato di vetro ed uno di silicio. Quello che cambia sono però le dimensioni che ho impostato a 0,35x0,35x5 cm, cosi come mostrato in *Figura 4.12*. La scelta di una superficie di impatto pari a 0,35x0,35cm è stata motivata nel paragrafo 4.2.5.

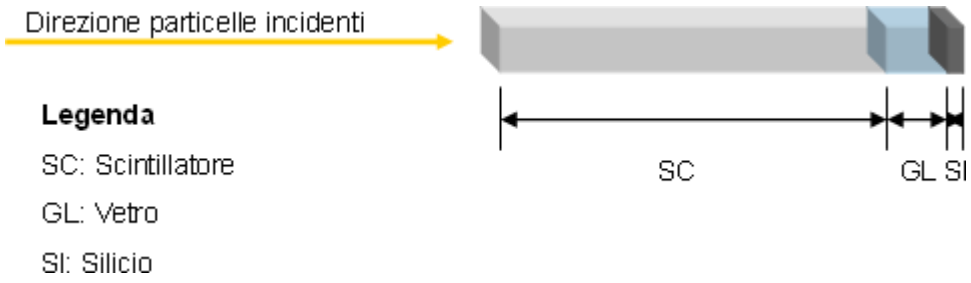


Figura 4.12: Geometria a bastoncino

Per questa geometria ho testato tre diversi scintillatori, il CsI, l'LSO ed il BGO. Da questo momento in poi ho ritenuto opportuno simulare esclusivamente con i raggi gamma da 511 keV in quanto i protoni non mi avrebbero fornito risultati significativi.

L'unica modifica che ho apportato al codice già illustrato nel capitolo precedente è stata all'interno della classe `DetectorConstruction` così come mostrato nel *Codice*

4.1. Come è possibile notare, ho modificato le dimensioni della *y* e della *z* portandole a 1,75mm, ovvero la metà dei 3,5mm, sempre per il motivo per cui tutte le definizioni di grandezze devono avvenire simmetricamente rispetto all'origine.

```
G4double Scintillator_dim_x = 2.5*cm;
G4double Scintillator_dim_y = 1.75*mm;
G4double Scintillator_dim_z = 1.75*mm;

G4double Scintillator_pos_x = 0.*cm;
G4double Scintillator_pos_y = 0.*cm;
G4double Scintillator_pos_z = 0.*cm;

G4ThreeVector Scintillator_position = G4ThreeVector
         (Scintillator_pos_x, Scintillator_pos_y, Scintillator_pos_z);

G4Box* Scintillator_box = new G4Box("Scintillator_box",
          Scintillator_dim_x, Scintillator_dim_y, Scintillator_dim_z);

Scintillator_log = new G4LogicalVolume
                          (Scintillator_box, CsI, "Scintillator_log");

Scintillator_phys = new G4PVPlacement(0, Scintillator_position,
    Scintillator_log, "Scintillator", experimentalHall_log, false, 0);
```

Codice 4.1: Costruzione della geometria a bastoncino

La durata dei *run* delle simulazioni è dipesa dal tipo di materiale scintillante, in particolar modo è stata proporzionale alla sua resa di scintillazione ed alla sua densità. Le durate infatti con il CsI e l'LSO sono state circa uguali (12 ore) mentre quella con il BGO leggermente inferiore (8 ore circa).

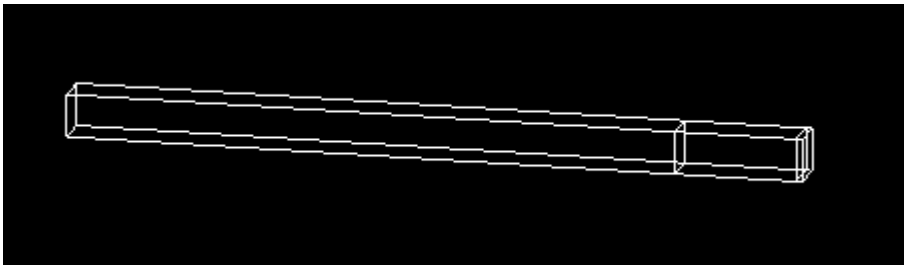


Figura 4.13: Geometria della configurazione a bastoncino

Questo perché, nonostante l'LSO abbia una resa di scintillazione inferiore del CsI, ha una densità circa il doppio.

In *Figura 4.13* è possibile vedere la geometria ottenuta per questa simulazione. Anche in questo caso ho utilizzato la visualizzazione grafica esclusivamente per appurare che la geometria creata da codice coincidesse con quella progettata.

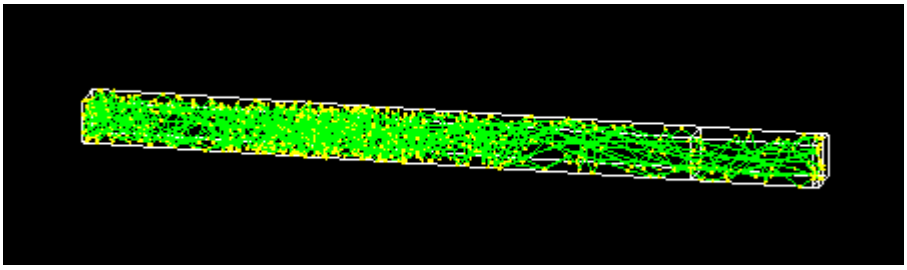


Figura 4.14: Esempio di run con la configurazione a bastoncino

Nella *Figura 4.14* è invece visualizzato un esempio di *run* con pochi eventi, anche qui per verificare che le superfici si comportino come desiderato.

Successivamente ho disabilitato la versione grafica ed eseguito le vere e proprie simulazioni. In *Tabella 4.2* sono mostrati i risultati ottenuti.

Scintillator		Time						N.photons				
scintillator	geom. Eff.	offset	ampl.	avg.	sigma	tau	(tau + sigma) / 2	ampl.	avg.	sigma	photo peak eff.	resol.
CsI	1	580,0	21,6	600,7	169,0	896,6	532,8	33,6	2252,7	75,4	0,212	0,033
LSO	1	486,3	21,5	447,0	75,8	146,2	111,0	93,3	729,3	29,6	0,462	0,041
BGO	1	624,0	3,6	665,6	139,3	1450,0	794,7	83,3	305,2	18,5	0,644	0,061
CsI	0.36	711,3	32,7	685,0	226,2	1948,0	1087,1	210,5	813,0	43,9	0,232	0,054
LSO	0.36	539,1	40,3	503,3	105,7	311,6	208,6	544,2	262,8	18,7	0,509	0,071
BGO	0.36	935,0	4,8	962,2	378,4	3278,8	1828,6	489,4	108,3	10,4	0,641	0,096

Tabella 4.2: Risultati delle simulazioni con geometria a bastoncino

Importante notare che, da questo momento in poi, ho ritenuto opportuno gestire anche l'efficienza geometrica del sensore di silicio, per i motivi già ampiamente discussi nel capitolo precedente. Per questo ho quindi ripetuto le simulazioni, sia con efficienza geometrica 1 che con efficienza geometrica 0,36, in modo da potere confrontarle tra loro. E' facile intuire come i tempi di simulazione si siano abbondantemente ridotti, proprio a causa dell'abbassamento dell'efficienza geometrica.

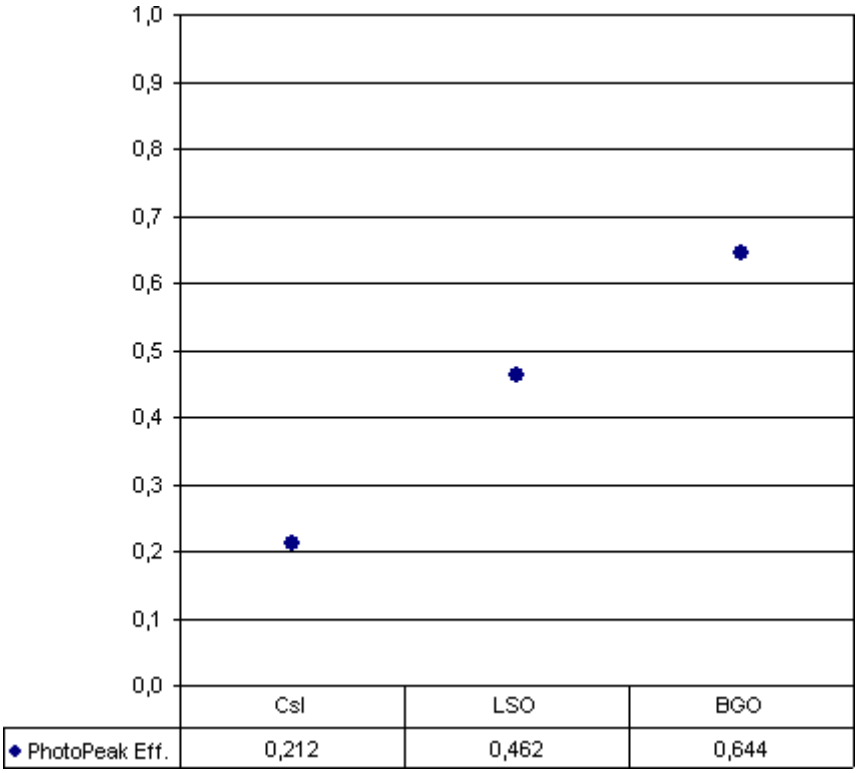


Figura 4.15: Efficienza di fotopicco per la simulazione a bastoncino con efficienza geometrica del sensore 1

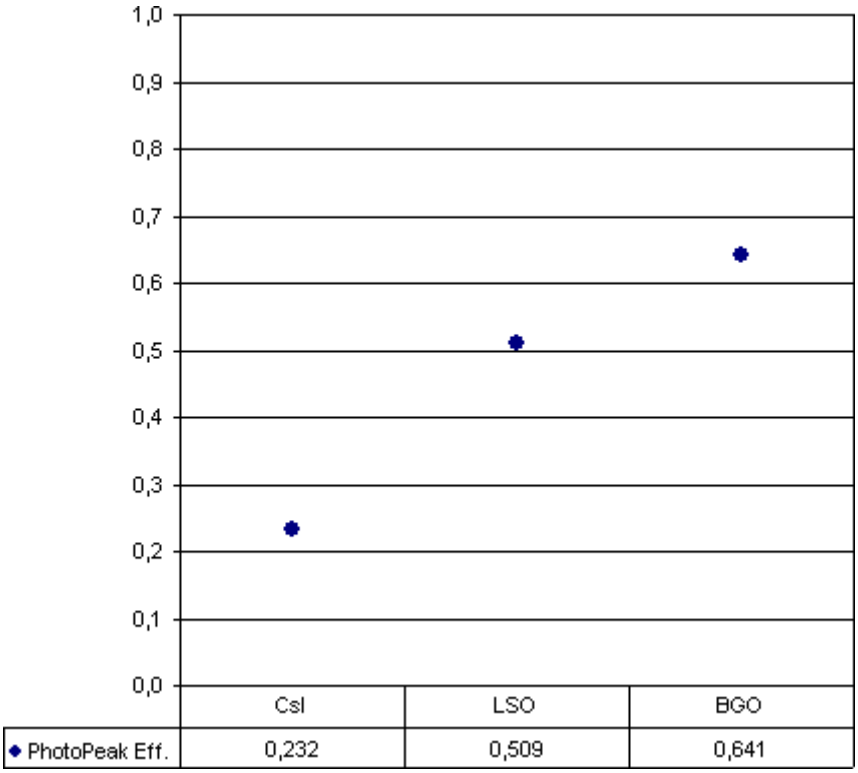


Figura 4.16: Efficienza di fotopicco per la simulazione a bastoncino con efficienza geometrica del sensore 0.36

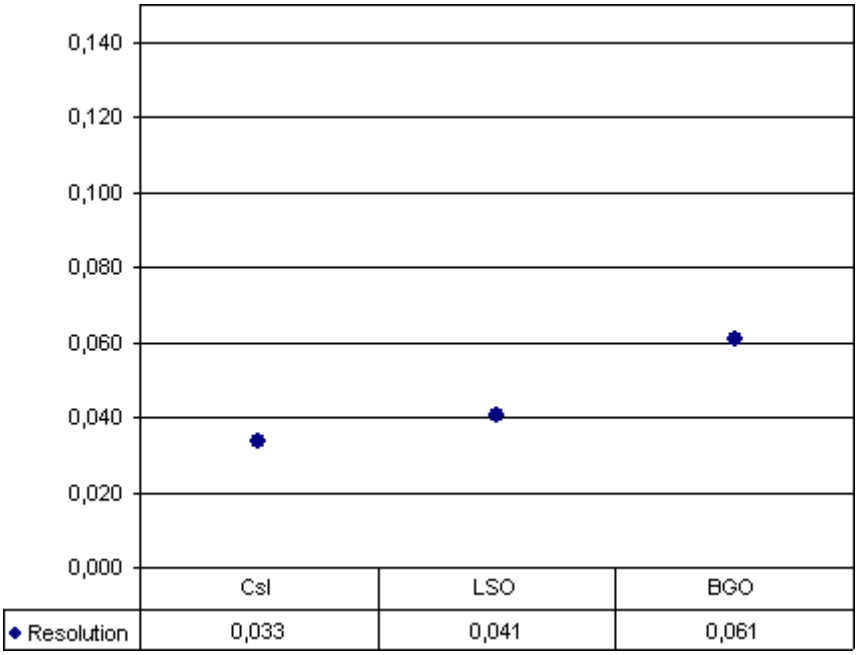


Figura 4.17: Risoluzione energetica per la simulazione a bastoncino con efficienza geometrica del sensore 1

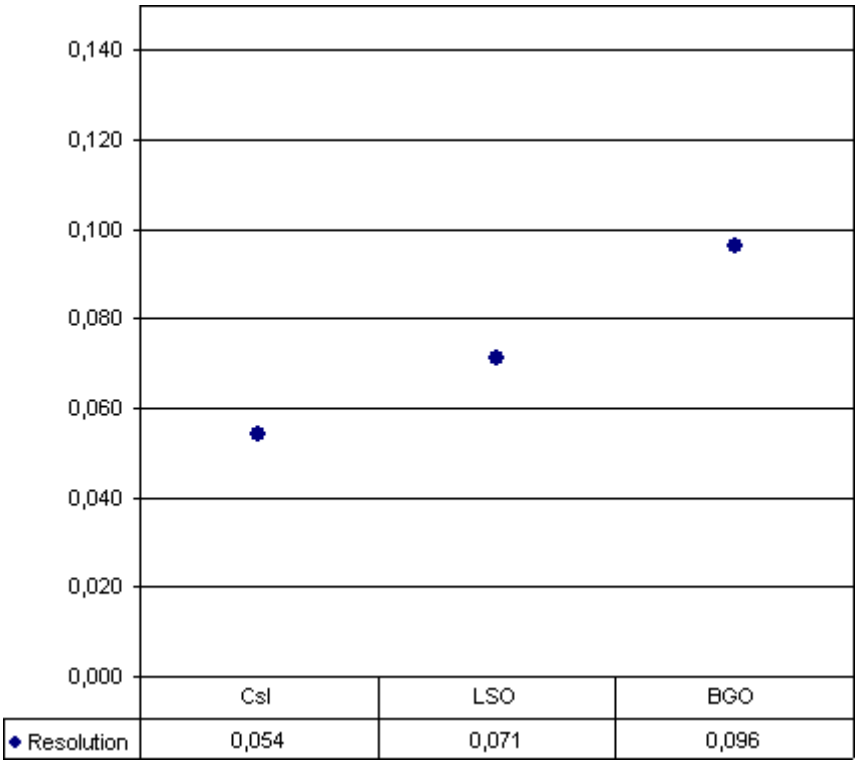


Figura 4.18: Risoluzione energetica per la simulazione a bastoncino con efficienza geometrica del sensore 0.36

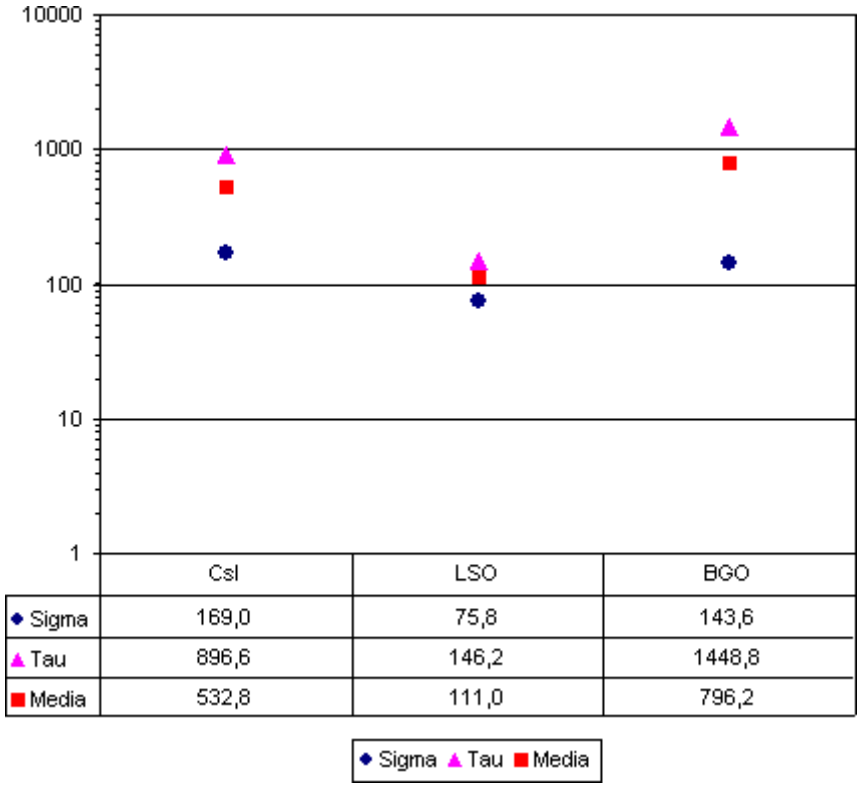


Figura 4.19: Risposta nel tempo per la simulazione a bastoncino con eff. geometrica del sensore 1

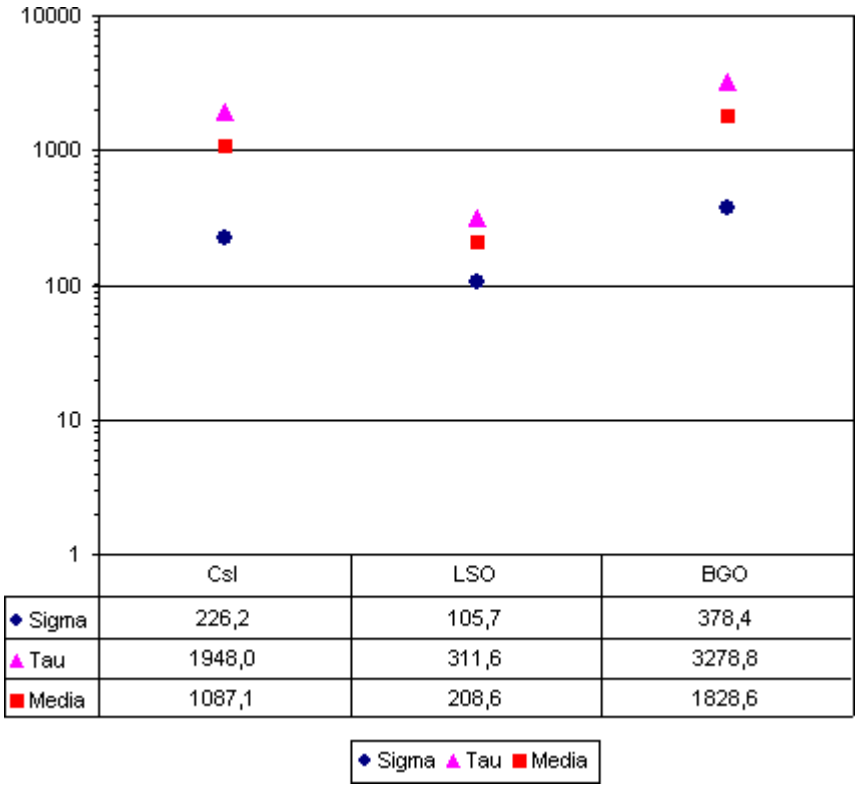


Figura 4.20: Risposta nel tempo per la simulazione a bastoncino con eff. geometrica del sensore 0.36

E’ interessante notare che, tra le due configurazioni con efficienza geometria 1 e 0,36, nonostante l’efficienza di fotopicco sia circa uguale, diminuisce la risoluzione energetica ed aumentano i tempi di risposta. Tale comportamento era immaginabile, in quanto, a parità di eventi lanciati, nel nostro caso 10.000, i fotoni rivelati dal sensore nel caso di efficienza 0,36 sono circa 1/3 rispetto a quelli rivelati nel caso di efficienza 1.

4.3 Simulazioni con geometria a sandwich

Il passo successivo è stato quello di produrre una geometria innovativa e quindi testarne l'efficienza, paragonandola a quella della geometria a bastoncino semplice.

Le dimensioni esterne sono anche qui di 0,35x0,35x5 cm, ma ciò che cambia realmente è la composizione interna della zone scintillante. In particolare sono stati utilizzati degli strati di materiale scintillante alternandone uno pesante (tre casi: CsI, BGO ed LSO) con uno leggero (il BC408). Ecco, quindi, il motivo per cui, in precedenza, ho fatto anche alcuni test sul funzionamento del BC408.

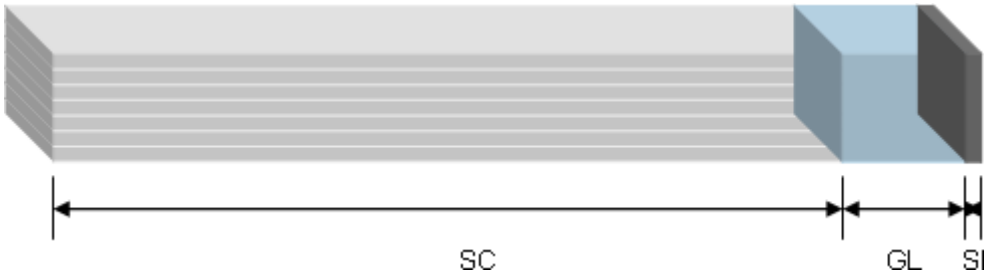


Figura 4.21: Geometria a sandwich

Lo scintillatore pesante è stato scelto delle dimensioni 0.457x3.5x50 mm (7 mattoncini) mentre il BC408 delle dimensioni 0.050x3.5x50 mm (6 foglietti).

L'assemblaggio, per le tre simulazioni, prevede le seguenti configurazioni, così come mostrato in *Figura 4.21*:

- LSO-BC408-LSO-BC408-LSO-BC408-LSO-BC408-LSO-BC408-LSO-BC408-LSO

- CsI-BC408-CsI-BC408-CsI-BC408-CsI-BC408-CsI-BC408-CsI-BC408-CsI

- BGO-BC408-BGO-BC408-BGO-BC408-BGO-BC408-BGO-BC408-BGO-BC408-BGO

E’ stato scelto di intervallare lo scintillatore pesante con quello leggero per tentare di sfruttarne i benefici di entrambi, ovvero una buona resa di scintillazione, per quello pesante, ed un trasferimento veloce dei fotoni generati verso il sensore di silicio, per quello leggero.

Vediamo ora il codice che ho implementato per creare una struttura del genere. Innanzitutto ho deciso di creare gli strati di scintillazione attraverso due array, entrambi di `NUM_LAYERS` elementi, uno per i volumi logici ed uno per quelli fisici, così come mostrato nel *Codice 4.2*.

```
class DetectorConstruction : public G4VUserDetectorConstruction {

  public:

    DetectorConstruction();
    ~DetectorConstruction();

    G4VPhysicalVolume* Construct();

  private:

    static const G4int NUM_LAYERS = 13;

    // Logical volumes
    //
    G4LogicalVolume* experimentalHall_log;
    G4LogicalVolume* Scintillator_log[NUM_LAYERS];
    G4LogicalVolume* Glass_log;
    G4LogicalVolume* Detector_log;

    // Physical volumes
    //
    G4VPhysicalVolume* experimentalHall_phys;
    G4VPhysicalVolume* Scintillator_phys[NUM_LAYERS];
    G4VPhysicalVolume* Glass_phys;
    G4VPhysicalVolume* Detector_phys;

};
```

Codice 4.2: Definizione della classe DetectorConstruction per la configurazione a sandwich

Quindi, all'interno del metodo `Construct()` della classe `DetectorConstruction`, ho implementato il codice visibile nel riquadro *Codice 4.3*, in cui è rappresentata la configurazione a sandwich tra CsI e BC408.

```
G4double Scintillator_dim_x =   25.*mm;
G4double Scintillator_dim_y_1 = 0.2285*mm;
G4double Scintillator_dim_y_2 = 0.025*mm;
G4double Scintillator_dim_z =   1.75*mm;

G4double Scintillator_pos_x =   0.*mm;
G4double Scintillator_pos_y =   0.*mm;
G4double Scintillator_pos_z =   0.*mm;

G4double Start_pos_y = -1.521*mm;

G4ThreeVector Scintillator_position;
G4Box* Scintillator_box;

for(G4int i=0; i < NUM_LAYERS; i++){

  Scintillator_pos_y =
      Start_pos_y + i * (Scintillator_dim_y_1 + Scintillator_dim_y_2);

  Scintillator_position = G4ThreeVector
         (Scintillator_pos_x, Scintillator_pos_y, Scintillator_pos_z);

  // Scintillatore pesante
  //
  if(i%2 == 0) {
     Scintillator_box = new G4Box("Scintillator_box_" + i,
        Scintillator_dim_x, Scintillator_dim_y_1, Scintillator_dim_z);

     Scintillator_log[i] = new G4LogicalVolume
                           (Scintillator_box, CsI, "Scintillator_log");
  }
  // Scintillatore leggero
  else {
     Scintillator_box = new G4Box("Scintillator_box_" + i,
        Scintillator_dim_x, Scintillator_dim_y_2, Scintillator_dim_z);

     Scintillator_log[i] = new G4LogicalVolume
                         (Scintillator_box, BC408, "Scintillator_log");
  }

  Scintillator_phys[i] = new G4PVPlacement(0, Scintillator_position,
                              Scintillator_log[i], "Scintillator_" + i,
                                       experimentalHall_log, false, 0);
}
```

Codice 4.3: Implementazione della geometria a sandwich

Attraverso un ciclo *for* ho popolato l'array dei volumi logici `Scintillator_log[]` inserendo, in alternanza, lo scintillatore pesante e quello leggero. Quindi ho popolato l'array dei volumi fisici `Scintillator_phys[]` impostandone la posizione attraverso l'ausilio della variabile `Scintillator_position,` il cui valore l'ho calcolato all'inizio del ciclo *for*.

Naturalmente, è stato necessario anche settare le proprietà della superficie tra lo scintillatore e l'aria, non più una sola come in precedenza, ma una per ogni strato di materiale.

Sempre all'interno del metodo `Construct()` della classe `DetectorConstruction`, ho quindi impostato le proprietà delle superfici, così come già ampiamente illustrato nel capitolo precedente, e le ho messe in relazione con ogni volume fisico creato, così come mostrato nella porzione di *Codice 4.4*.

```
G4OpticalSurface* ScintillatorOpticalSurface[NUM_LAYERS];

for(G4int i=0; i < NUM_LAYERS; i++) {

  ScintillatorOpticalSurface[i] = new G4OpticalSurface
                                    ("ScintillatorOpticalSurface" + i);

  new G4LogicalBorderSurface("ScintillatorOpticalSurface" + i,
                             Scintillator_phys[i], experimentalHall_phys,
                                        ScintillatorOpticalSurface[i]);

  ScintillatorOpticalSurface[i]->SetType(dielectric_metal);
  ScintillatorOpticalSurface[i]->SetFinish(polishedfrontpainted);
  ScintillatorOpticalSurface[i]->SetModel(unified);
  ScintillatorOpticalSurface[i]->SetSigmaAlpha(0.1);

  ScintillatorOpticalSurface[i]->SetMaterialPropertiesTable
                                   (ScintillatorOpticalSurfaceProperty);

}
```

Codice 4.4: Proprietà delle superfici

La durata dei run delle simulazioni, in questo caso, è stata ancora più lunga del caso precedente con tempi che hanno superato anche le 48 ore per la configurazione CsI-BC408.

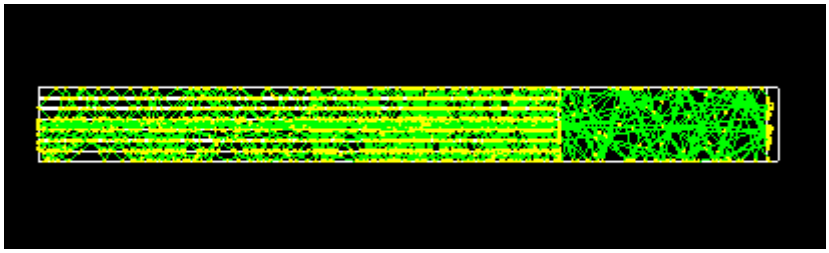


Figura 4.22: Run di esempio con geometria a sandwich

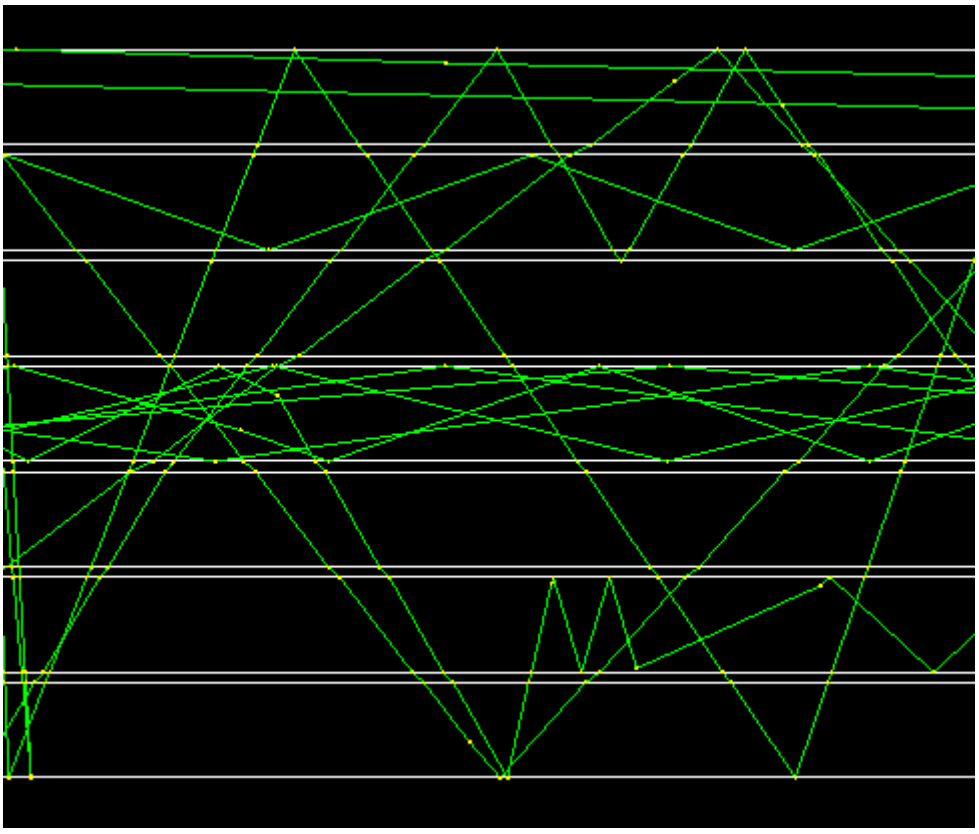


Figura 4.23: Zoom del rum con geometria a sandwich

In *Tabella 4.3* sono proposti i risultati delle simulazioni, mentre nelle pagine seguenti tali risultati sono stati riportati in grafico per una più facile interpretazione.

Le prime tre righe della tabella sono le stesse già riportate in *Tabella 4.2*. Ho preferito riscriverle nuovamente per permetterne un confronto diretto con la geometria a sandwich, in modo da valutarne vantaggi e svantaggi.

Da tali risultati è possibile apprezzare come, l'utilizzo della geometria a sandwich, ha portato ad un miglioramento delle prestazioni.

	Time						N.photons				
scintillator	**offset**	**ampl.**	**avg.**	**sigma**	**tau**	**(tau + sigma) / 2**	**ampl.**	**avg.**	**sigma**	**photo peak eff.**	**resol.**
CsI	711,3	32,7	685,0	226,2	1948,0	1087,1	210,5	813,0	43,9	0,232	0,054
LSO	539,1	40,3	503,3	105,7	311,6	208,6	544,2	262,8	18,7	0,509	0,071
BGO	935,0	4,8	962,2	378,4	3278,8	1828,6	489,4	108,3	10,4	0,641	0,096
CsI-BC408	644,0	34,8	625,2	188,8	1895,5	1042,2	185,8	879,3	45,5	0,212	0,052
LSO-BC408	517,7	43,4	487,0	95,1	293,9	194,5	498,5	284,6	19,4	0,484	0,068
BGO-BC408	886,0	5,8	808,4	244,8	2764,8	1504,8	410,6	133,1	11,9	0,611	0,089

Tabella 4.3: Risultati delle simulazioni con geometria a sandwich

Prendendo in esame ad esempio il caso con BGO, la risoluzione è salita da 9,6% all'8,9%, l'efficienza di fotopicco è invece leggermente peggiorata, passando da 0,641 a 0,611. Il tempo di risposta è invece migliorato abbastanza passando da una media di 1828,6 a 1495,7. Questo dimostra come il tentativo effettuato nel testare una geometria innovativa ha dato dei risultati interessanti.

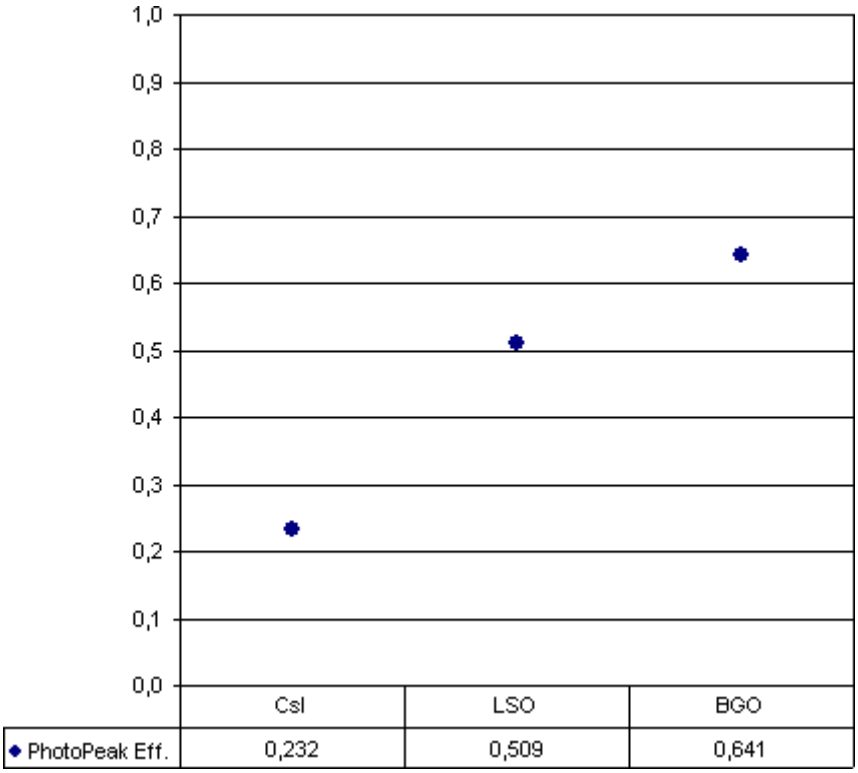


Figura 4.24: Efficienza di fotopicco per la simulazione a bastoncino con efficienza geometrica del sensore 0.36

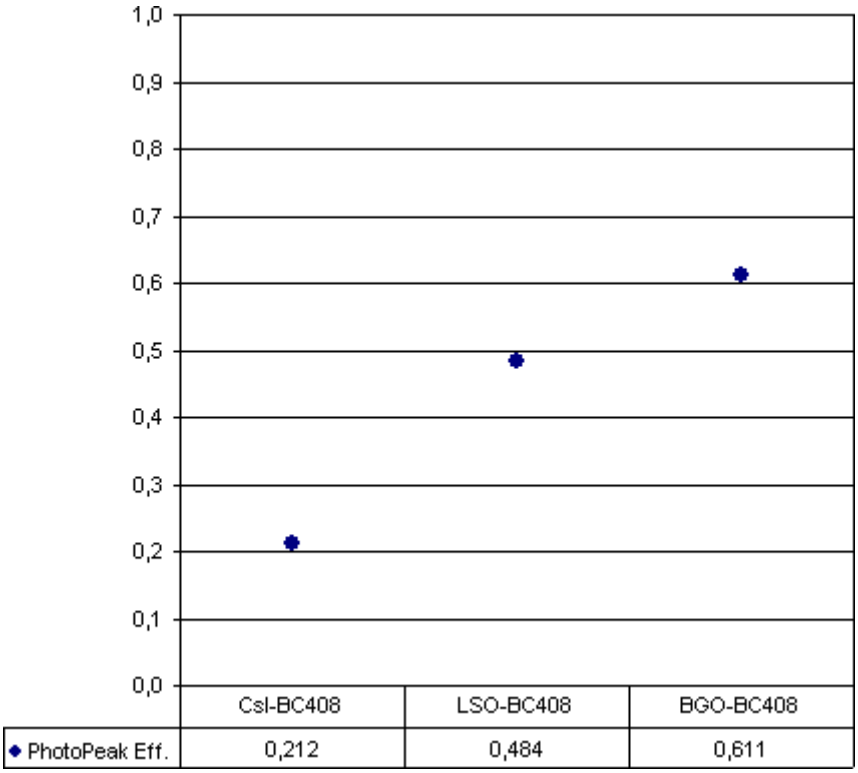


Figura 4.25: Efficienza di fotopicco per la simulazione a sandwich con efficienza geometrica del sensore 0.36

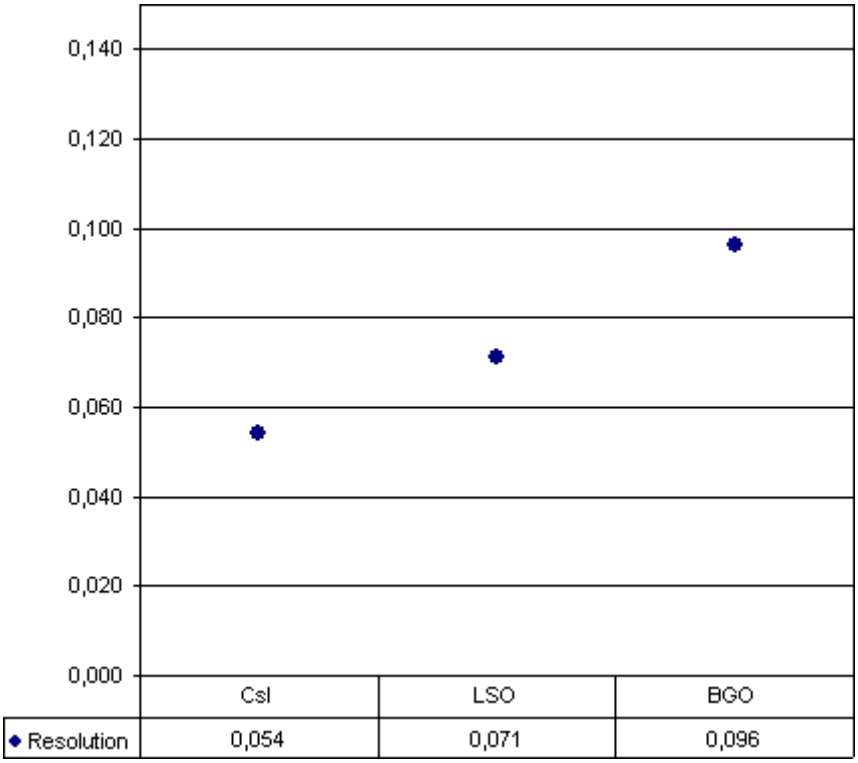


Figura 4.26: Risoluzione energetica per la simulazione a bastoncino con efficienza geometrica del sensore 0.36

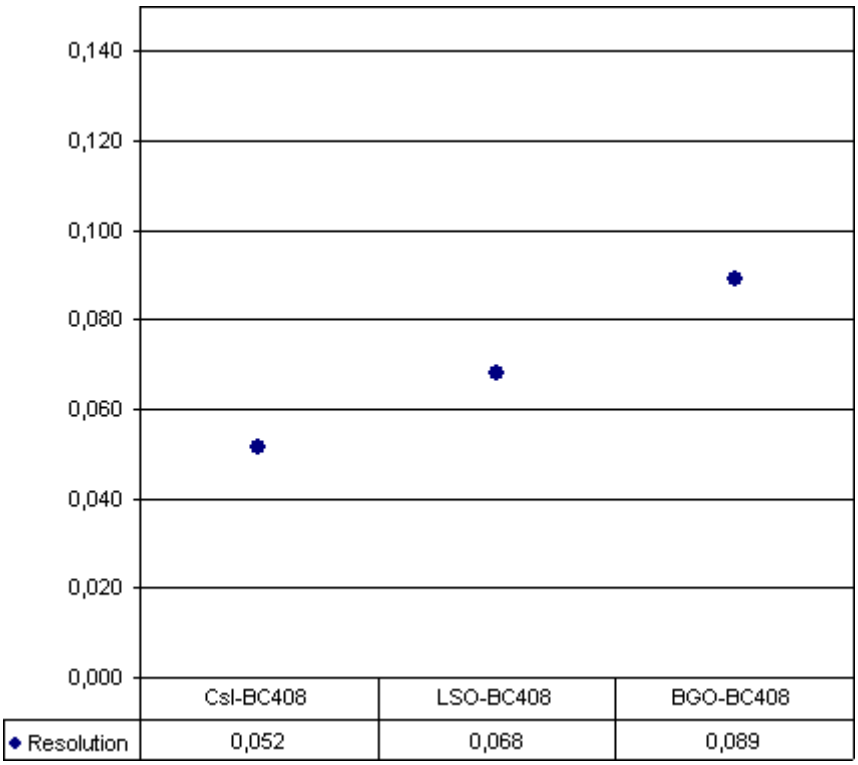
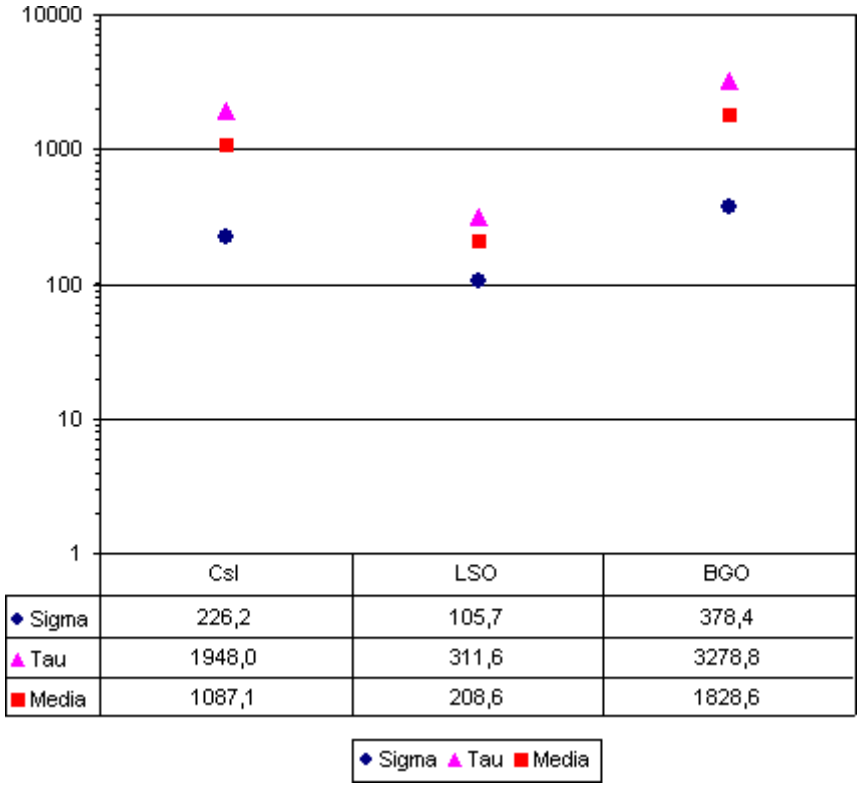
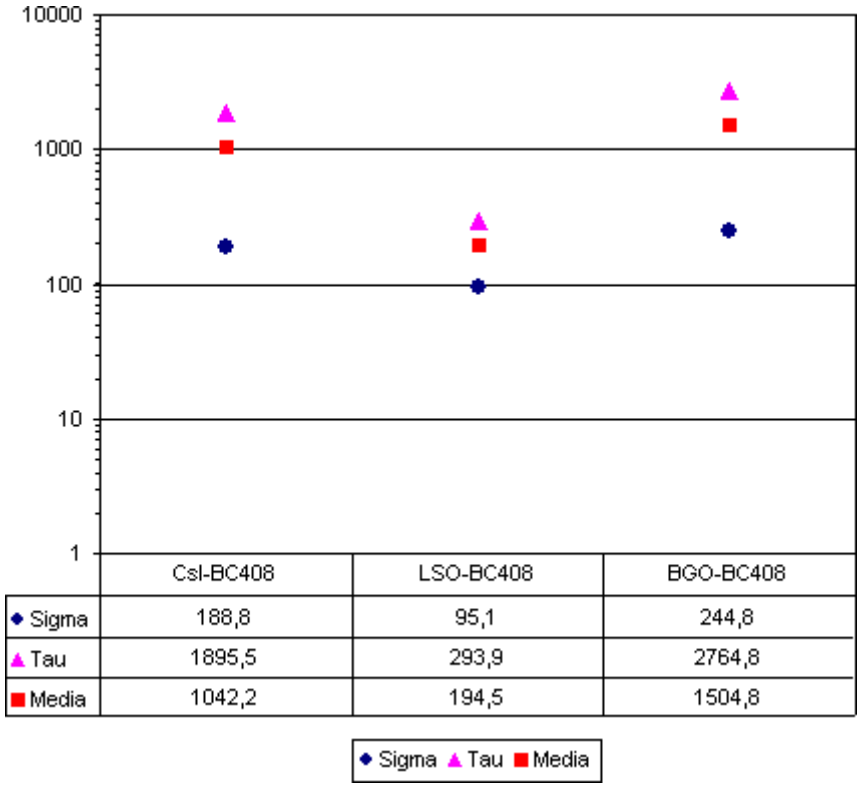


Figura 4.27: Risoluzione energetica per la simulazione a sandwich con efficienza geometrica del sensore 0.36



	CsI	LSO	BGO
Sigma	226,2	105,7	378,4
Tau	1948,0	311,6	3278,8
Media	1087,1	208,6	1828,6

Figura 4.28: Risposta nel tempo per la simulazione a bastoncino con eff. geometrica del sensore 0.36



	CsI-BC408	LSO-BC408	BGO-BC408
Sigma	188,8	95,1	244,8
Tau	1895,5	293,9	2764,8
Media	1042,2	194,5	1504,8

Figura 4.29: Risposta nel tempo per la simulazione a sandwich con eff. geometrica del sensore 0.36

4.4 Simulazioni con geometria a bastoncino ed a sandwich corto

Gli ultimi e più importanti *step* di questo progetto sono consistiti nell'accorciare la lunghezza dei bastoncini e dei sandwich passando da 5 cm ad una misura pari circa alla lunghezza di assorbimento media $\lambda=1/\mu$ dei fotoni gamma all'interno del materiale specifico, già ampiamente descritta nel paragrafo 3.2.

Per questo motivo ho scelto la lunghezza di 25 mm nel caso dello scintillatore con CsI, che ha λ=22,9mm, mentre 12 mm nel caso di LSO e BGO, che hanno rispettivamente λ=11,4mm e λ=10,4mm.

Questa è una scelta effettuata per trovare un compromesso tra una buona efficienza di fotopicco e un piccolo errore di parallasse. L'ideale per avere una ottima efficienza di fotopicco sarebbe avere il sensore il più grande possibile, non solo in lunghezza ma in tutte e tre le dimensioni, in modo tale che la probabilità di assorbimento dei gamma all'interno del dispositivo sia elevata.

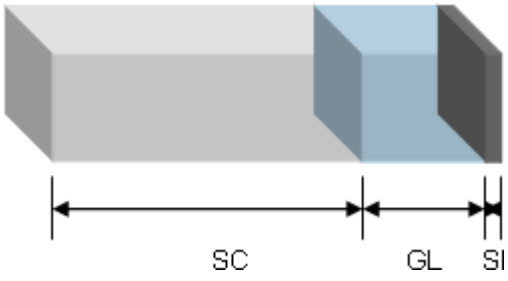


Figura 4.30: Geometria a bastoncino corto

Per ridurre l'errore di parallasse, invece, l'ideale sarebbe avere il sensore quanto più sottile e corto possibile, così da essere sensibile anche al singolo "pixel". Scegliere quindi una lunghezza pari a λ vuol dire stabilire un compromesso che permette di rivelare circa i 2/3 dei gamma irradiati.

La maggior parte dei sensori in genere utilizzati, ha proprio le dimensioni sopra illustrate, appunto per assolvere a tale scopo.

Non mi soffermerò ulteriormente ad illustrare il codice di queste simulazioni, dato che è uguale alle precedenti con geometria a bastoncino ed a sandwich, con l'unica differenza nella coordinata *x*, che varia come spiegato sopra.

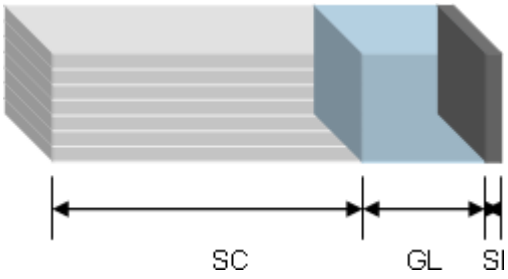


Figura 4.31: Geometria a sandwich corto

In *Figura 4.30* ed in *Figura 4.31* sono rappresentati gli schemi logici delle due configurazioni a bastoncino ed a sandwich.

La durata di queste simulazioni è stata inferiore a quelle con lunghezza 5 cm. Nel caso del bastoncino sono bastate circa 6 ore, mentre nel caso del sandwich circa 12 ore per simulazione. I risultati, anch'essi ottenuti con il lancio di 10.000 eventi, sono riportati in *Tabella 4.4*.

	Time						Nphotons				
scintillator	offset	ampl.	avg.	sigma	tau	(tau + sigma) / 2	ampl.	avg.	sigma	photo peak eff.	resol.
CsI	530,0	32,7	503,4	221,8	1432,5	827,2	145,2	1283,5	55,4	0,202	0,043
LSO	240,3	42,9	217,6	65,4	188,7	127,0	291,9	518,9	22,6	0,331	0,044
BGO	604,0	4,8	558,8	310,3	2133,6	1222,0	245,3	181,9	13,6	0,418	0,075
CsI-BC408	469,7	35,2	450,9	161,5	1385,7	773,6	125,9	1350,8	62,1	0,196	0,046
LSO-BC408	215,6	47,4	202,4	49,4	184,8	117,1	265,9	542,6	24,2	0,323	0,045
BGO-BC408	448,0	5,3	446,1	175,1	2008,9	1092,0	215,9	199,3	15,1	0,408	0,076

Tabella 4.4: Risultati per le simulazioni a bastoncino e sandwich corti

Nelle pagine seguenti sono riportati, in grafici, i risultati ottenuti per queste simulazioni, per permettere un confronto intuitivo tra loro.

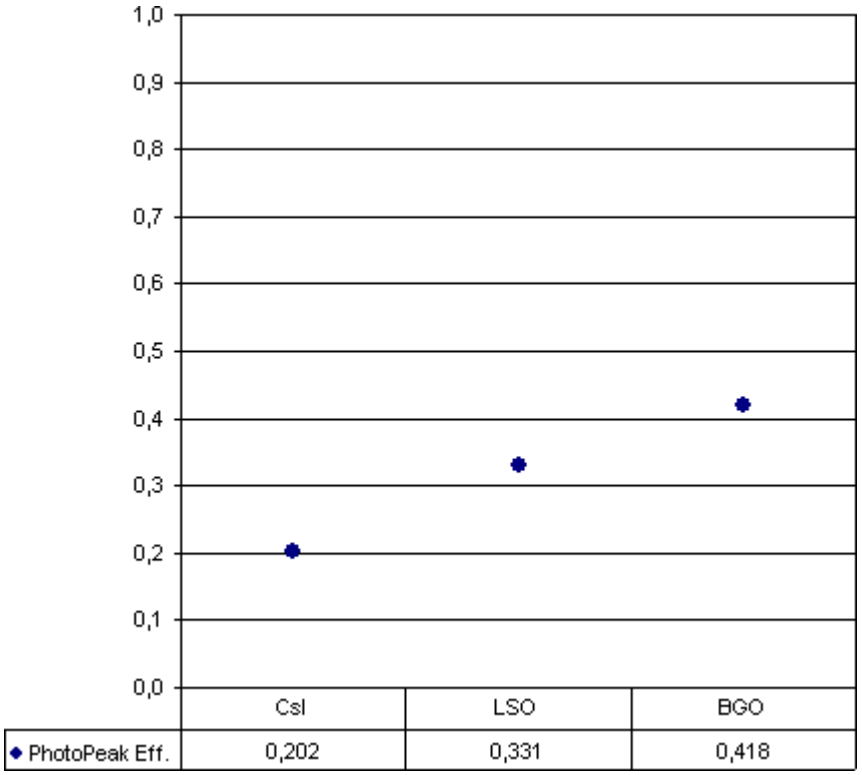


Figura 4.32: Efficienza di fotopicco per la simulazione a bastoncino corto

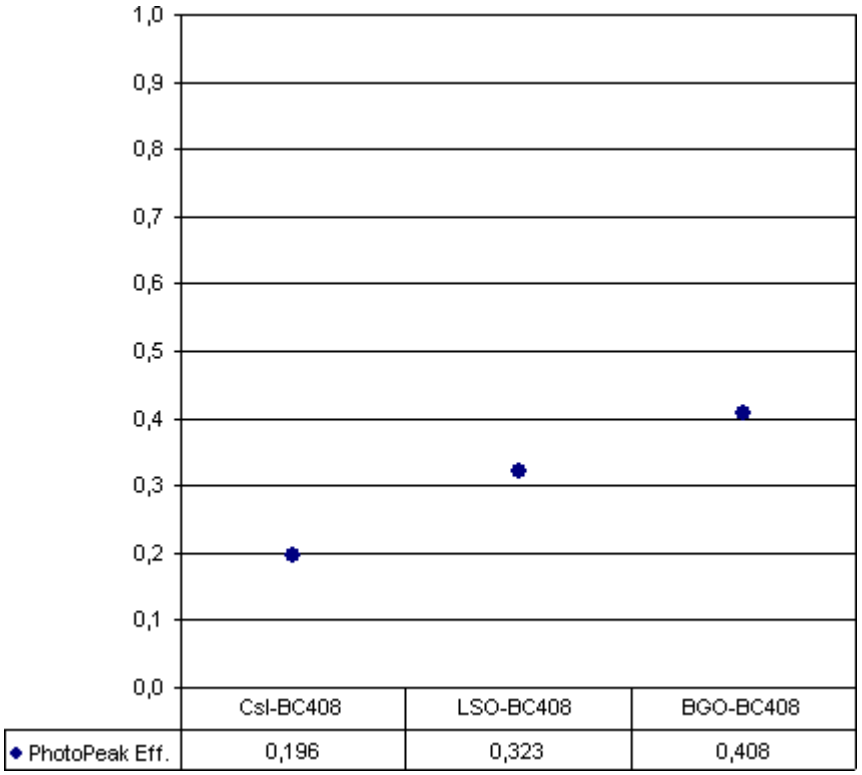


Figura 4.33: Efficienza di fotopicco per la simulazione a sandwich corto

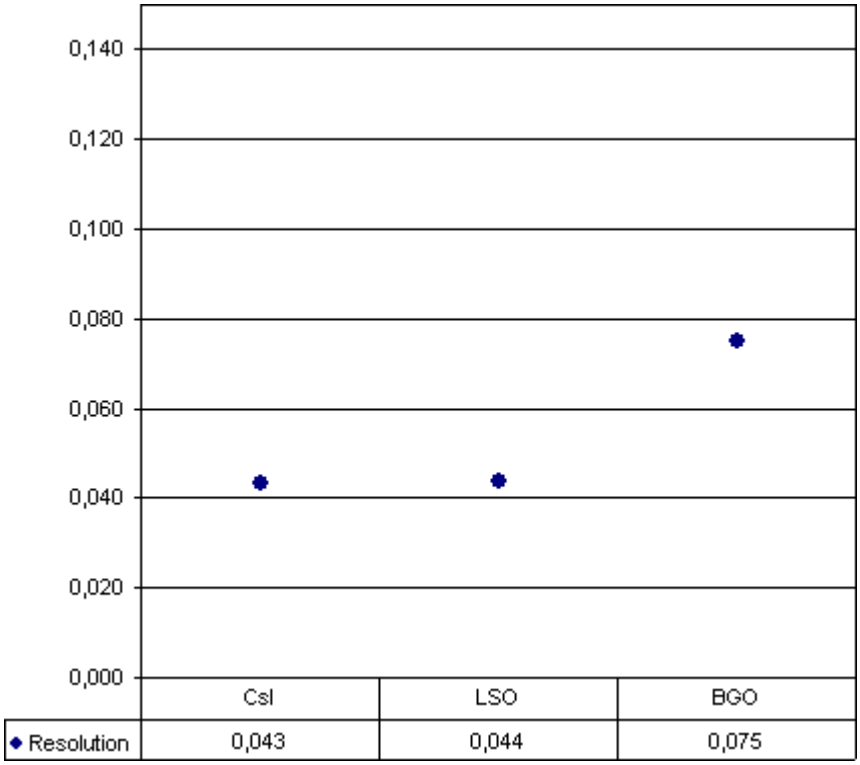


Figura 4.34: Risoluzione energetica per la simulazione a bastoncino corto

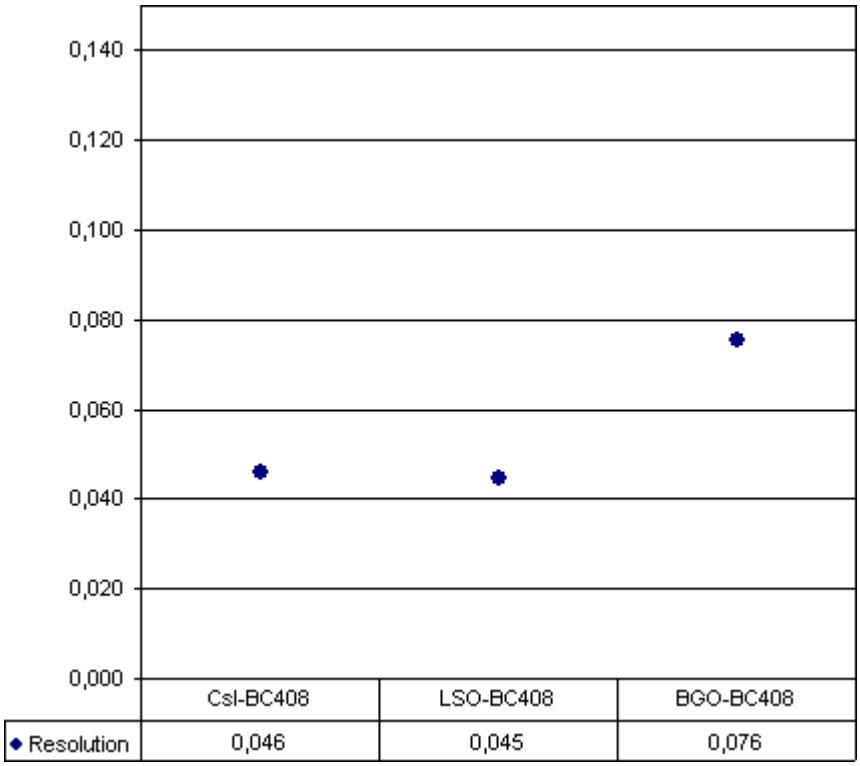


Figura 4.35: : Risoluzione energetica per la simulazione a sandwich corto

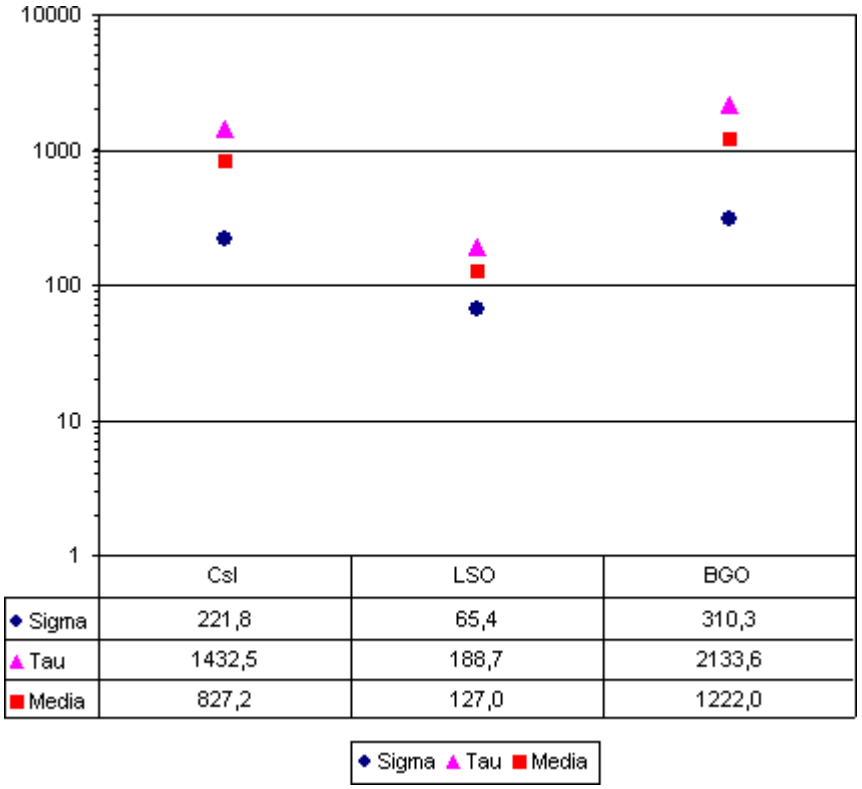


Figura 4.36: Risposta nel tempo per la simulazione a bastoncino corto

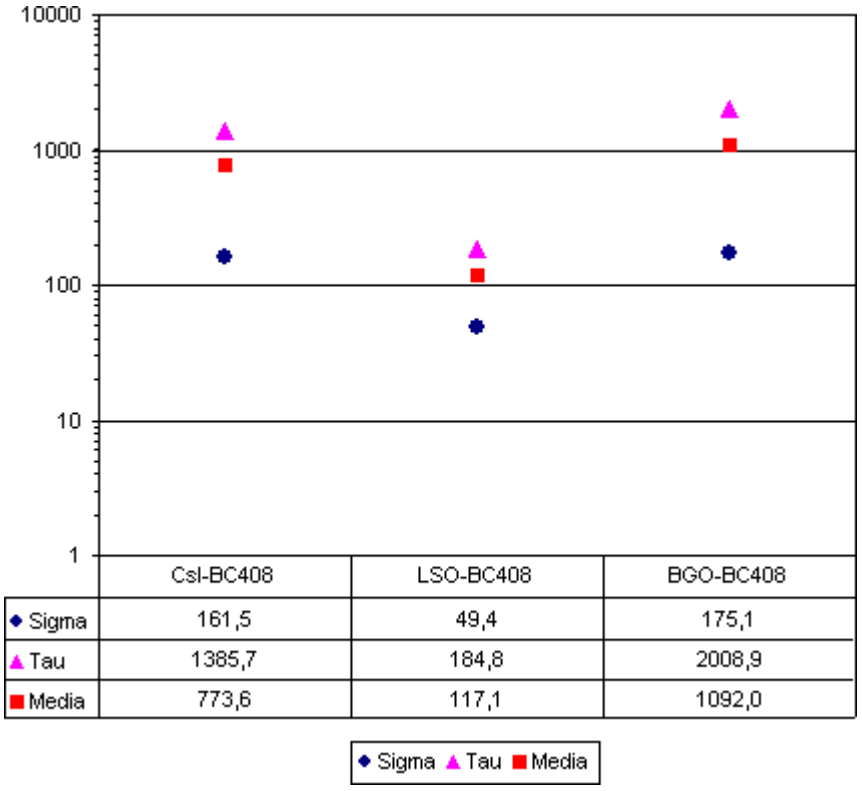


Figura 4.37: Risposta nel tempo per la simulazione a sandwich corto

Conclusioni

Nella presente tesi si è affrontato il problema dello studio dei sensori di raggi gamma utilizzati nello strumento di diagnostica oncologia per eccellenza, la tomografia ad emissione di positroni (PET).

Inoltre sono stato effettuati test simulativi su un nuovo modello di sensore, con una geometria innovativa, che ha dato risultati molto interessanti, i quali potrebbero aprire nuove strade per sviluppi futuri.

Tale studio è stato effettuato attraverso lo sviluppo di una simulatore scritto in C++, con l'ausilio del toolkit Geant4, e strutturando il codice secondo modelli avanzati di ingegneria del software, in modo da sfruttare a pieno le potenzialità del paradigma ad oggetti di cui il C++ è basato, grazie alle sue proprietà di ereditarietà e polimorfismo.

Questo lavoro è nato dalla necessita di interfacciare il campo fisico-applicativo con quello informatico ed è stato possibile verificare come, attraverso il software simulativo sviluppato, si possono ottenere risultati fisici di rilievo, facilmente riproducibili e modificabili con introduzione di nuove geometrie e nuovi materiali.

I risultati più interessanti e soddisfacenti sono venuti dal confronto tra le diverse geometrie testate ed i materiali utilizzati. Si è visto come la geometria a sandwich corto, illustrata nel Capitolo 4, sia la configurazione più promettente rispetto alle altre.

I risultati di tali simulazioni aprono la strada, adesso, al test diretto presso i LNS-INFN di alcune delle configurazioni simulate evitando quelle che, grazie alle

simulazioni, non sembrano di interesse. Il vantaggio è che ci si limiterà alla verifica delle più interessanti con un notevole risparmio di risorse umane ed economiche.

E' stato possibile imparare, anche, come si comportano e come scalano i risultati delle simulazioni in funzione delle proprietà ottiche, fisiche e geometriche dei materiali.

Ringraziamenti

Ringraziamenti particolari vanno a tutti coloro che mi sono stati vicini e che mi hanno sostenuto.

Un ringraziamento particolare va al Dott. Paolo Finocchiaro che, con le sue eccellenti conoscenze di fisico, ha suscitato in me un interesse tale da invogliarmi nello sviluppo di questo elaborato, contribuendo in maniera fondamentale alla produzione di questa tesi ed alla mia crescita professionale.

Al Prof. Michele Malgeri, per la sua totale disponibilità ed assistenza nella produzione, sviluppo e revisione della tesi.

A tutto il Centro di Calcolo dei LNS-INFN, per la disponibilità e l'assistenza nei momenti di necessità.

Al Dott. Massimiliano Berretta ed a tutto il Centro di Riferimento Oncologico di Aviano, per avermi dato la possibilità di essere qui in questo momento.

Alla mia famiglia, per essermi stata vicina nei momenti più difficili della mia vita e per avermi sostenuto in questa esperienza con orgoglio e soddisfazione.

Ed infine, ma non certo ultima per importanza, ad Ornella, la ragazza più importante della mia vita, per aver condiviso con me ogni momento bello e difficile degli ultimi anni.

Bibliografia

[Rif. 1] Immagine tratta da http://www.brachiterapia.it

[Rif. 2] Immagine tratta da http://www.irmet.com

[Rif. 3] Database materiali disponibile sul sito ufficiale del Geant4 all'indirizzo: http://geant4.web.cern.ch/geant4/G4UsersDocuments/UsersGuides/ForApplicationDeveloper/html/Detector/materialNames.html

[Rif. 4] M. Mazzillo et al., *Silicon Geiger mode avalanche photodiodes*, Optoelectronics Letters, Vol.3, N.3, (2007)177

[Rif. 5] Paulo Alexandre Vieira Crespo – *Optimization of In-Beam Positron Emission Tomography for Monitoring Heavy Ion Tumor Therapy* – 2005

[Rif. 6] *Tomografia ad emissione di positroni su Wikipedia* http://it.wikipedia.org/wiki/Tomografia_ad_emissione_di_positroni

[Rif. 7] Ben-Hui. Liu – *Statistical Genomics: Linkage, Mapping, and QTL Analysis* – 1997

[Rif. 8] Gerhard Arminger, Clifford C. Clogg, Michael E. Sobel – *Handbook of statistical modeling for the social and behavioral sciences* – 1995

[Rif. 9] Carlo Chiesa e Maria Carla Gilardi - S.C. Medicina Nucleare – Istituto Nazionale Tumori – Milano - IBFM CNR, Università di Milano Bicocca, Istituto scientifico H S.Raffaele, Milano

[Rif. 10] http://www.ct.infn.it/~rivel/Tipi/radiazioni/fotoni.html

[Rif. 11] http://en.wikipedia.org/wiki/Isotope

[Rif. 12] Ralph Dollan – *Simulation of optical processes in Geant4*

[Rif. 13] Saint-Gobain Crystals – *BGO Datasheet*

[Rif. 14] Ispoptics – *Cesium Iodide*

[Rif. 15] Saint-Gobain Crystals – *CsI Datasheet*

[Rif. 16] Catherine Michelle Pepin, Philippe Bérard, Anne-Laure Perrot, Claude Pépin, Daniel Houde, Roger LecomtCharles L. Melcher,Henri Dautet - *Properties of LYSO and Recent LSO Scintillators for Phoswich PET Detectors* – 2004

[Rif. 17] Simone Weber, Daniela Christ, Marcel Kurzeja, Ralf Engels, Guenter Kemmerling, Horst Halling - Comparison of LuYAP, LSO and BGO as scintillators for high resolution PET detectors

[Rif. 18] P.Finocchiaro et al., *SPAD arrays and micro-optics: towards a real single photon spectrometer*, Journal of Modern Optics, Vol. 54, Nos. 2–3 (2007)199

A.Campisi, L.Cosentino, P.Finocchiaro, et al., *Multipixel geiger-mode photon detectors for ultra-weak light sources*, Nuclear Instruments and Methods in Physics Research A571 (2007)350

P.Finocchiaro, et al., *A new generation of low-voltage single photon micro-sensors with timing capability*, Nuclear Instruments and Methods in Physics Research A567 (2006)83

P.Finocchiaro, et al., *Test of scintillator readout with single photon avalanche photodiodes*, IEEE Transactions on Nuclear Science, Vol.52, N.6, (2006)3040