Zastosowanie pakietu Geant4 w fizyce jądrowej Wykład 6

Aleksandra Fijałkowska

18 listopada 2021

Zadanie

Oszacuj liczbę sygnałów pochodzących z tła w układzie detektorów tworzących PET.
Założenia:

- Tło generuje źródło 60Co zlokalizowanie w jednym miejscu świata (gdzieś w kącie, nie na środku)
- Wygeneruj dwa kwanty γ odpowiadające rozpadwi jądra 60Co
- Stwórz wykres LICZBY zliczeń pochodzących z tła w zależności od numeru detektora
- Zwróć uwagę na to, że liczamy tylko ZDARZENIA w których nastapiła interakcja z którymś z detektorów. Jeśli kilka kroków w ramach jednego zdarzenia miało miejsce w konkretnym detektorze, to zliczamy je tylko jeden raz.

Wyniki symulacji, Prosty licznik

Druga metoda wyciągania danych z symulacji wykorzystuje wbudowane narzędzia biblioteki geant, a konkretnie `G4VPrimitiveScorer` (Proste Liczniki??). Obiekty typu `G4VPrimitiveScorer` służą do zliczania jakiejś jednej, wybranej wielkości.
Licznik przypisany jest do objętości logicznej (**G4LogicalVolume**). Jeśli więc mamy kilka elementów detekcyjnych, każdy z nich musi mieć swój oddzielny licznik. Wyjątek stanowią obiekty, które są kopiami jednej objętości logicznej (tak jak nasze detektory NaI). Tu wystarczy jeden licznik. Nie powinno nas to dziwić, bo przecież wszystkie te kopie powstają z jednego obiektu **G4LogicalVolume**.
Przy okazji zabawy z prostymi licznikami dowiemy się, dlaczego tak ważne jest poprawne numerowanie kopii obiektów (czyli pamiętanie o numerowaniu).

Kilka przykładowych liczników:

- **G4PSTrackLength** – zwraca długość toru, liczoną jako sumę długości kroków, jakie pokonała cząstka wewnątrz detektora
- **G4PSEnergyDeposit** — zwraca całkowitą energię zdeponowaną w detektorze
- **G4PSDoseDeposit** — zwraca całkowitą energię zdeponowaną w detektorze podzieloną przez masę detektora. Masa jest wyznaczana w oparciu o gęstość i objętość określone w **G4VSolid** oraz **G4LogicalVolume**
- Grupa liczników zwracająca prąd i strumień cząstek przelatujących przez powierzchnię detektora
- **G4PSMinKinEAtGeneration** – zwraca minimalną energię kinetyczną cząstki wtórnej wytworzonej w detektorze (tu nie ma sumowania, podawana jest wyłącznie najmniejsza wartość energii)
- **G4PSNofSecondary** – zwraca liczbę cząstek wtórnych, wygenerowanych w detektorze
- **G4PSNofStep** – zwraca liczbę kroków w detektorze
- **G4PSCellCharge** – zwraca całkowity ładunek cząstek zatrzymanych w detektorze

Pełną listę liczników zawiera rozdział 4.4.5. Geant4 User's Guide for Application Developers.

Implementacja

Implementacja jest dziwna. Mam nadzieję, że się nie zgubicie. Przećwiczymy ją jeszcze „na żywo".
W klasie, w której tworzymy geometrię tworzymy metodę `ConstructSDandField()` (nie jest czyto wirtualna, więc też obowiązkowa), a w niej:

```
G4MultiFunctionalDetector* detector =
                          new G4MultiFunctionalDetector("naISensitiveDet");
G4int depth = 2;
G4VPrimitiveScorer* energyDepScorer = new G4PSEnergyDeposit("eDep",depth);
detector->RegisterPrimitive(energyDepScorer);
NaILogic->SetSensitiveDetector(detector);
G4SDManager* SDmanager = G4SDManager::GetSDMpointer();
SDmanager->AddNewDetector(detector);
```

W objętości logicznej NaILogic będzie zliczana energia w niej zdeponowana. Można do G4MultiFunctionalDetector dodać więcej liczników. Możemy zbierać różne statystyki z danej obiętości logicznej.

Implementacja

Co tu się zadarza?

- Utworzyć obiekt klasy **G4MultiFunctionalDetector** (nazwa jest ważna, jednoznacznie identyfikuje detektor i umożliwia dostęp do danych)

```
G4MultiFunctionalDetector* detector =
new G4MultiFunctionalDetector("naISensitiveDet");
```

- Zarejestrować go do Sensitive Detector Manager-a (**G4SDManager**)

```
G4SDManager::GetSDMpointer()->AddNewDetector(detector);
```

lub w dwóch krokach:

```
G4SDManager* SDmanager = G4SDManager::GetSDMpointer();
SDmanager->AddNewDetector(detector);
```

- Przypisać MultiFunctionalDetector do interesujących obszarów logicznych

```
NaILogic->SetSensitiveDetector(detector);
```

- Utworzyć obiekt klasy **G4VPrimitiveScorer**, konstruktor przyjmuje nazwę oraz liczbę całkowitą, oznaczającą rząd przodka, określającego numer kopii

```
G4VPrimitiveScorer* energyDepScorer = new G4PSEnergyDeposit("eDep",depth);
```

- Zarejestrować PrimitiveScorer do MultiFunctionalDetector

```
detector->RegisterPrimitive(energyDepScorer);
```

Implementacja

Jak dostać się do zabranych danych? Teraz zrobi się jeszcze bardziej dziwnie...
Każdy **G4VPrimitiveScorer** generuje mapę **G4THitsMap<G4double>**.
Indeksem mapy jest numer kopii obszaru logicznego do którego przypisany jest MultiFunctionalDetector (lub numer kopii jego przodka, w zależności od parametru, który wpisaliśmy w konstruktorze klasy **G4VPrimitiveScorer**).
Wartościami w mapie są wielkości, które miał zliczać **G4VPrimitiveScorer** (w naszym przypadku depozyty energii). Dostęp do **G4THitsMap<G4double>** można uzyskać po skończonym zdarzeniu (Event). Są one zalokowane w obiekcie typu **G4HCofThisEvent**, do którego dostęp uzyskuje się ze zmiennej **G4Event**.
W tym celu należy:

1. Znaleźć unikalny numer interesującej mapy. Numery te przetrzymuje G4SDManager (sensitive detector manager), który jest singletonem. Dostęp do niego można uzyskać dzięki publicznej, statycznej metodzie **GetSDMpointer()**. Obiekt ten ma publiczną metodę **G4int GetCollectionID (G4String colName)** zwracającą numer kolekcji. Argumentem wejściowym jest nazwa kolekcji, składająca się z nazwy **G4MultiFunctionalDetector** / nazwy **G4VPrimitiveScorer**. W naszym przykładzie byłoby to „naISensitiveDet/eDep”

Implementacja

2. Wyciągnąć z Eventu (po jego zakończeniu) obiekt typu G4HCofThisEvent (hits collections of an event), odpowiada za to metoda **GetHCofThisEvent()** klasy **G4Event**. Może się zdarzyć, że metoda zwróci pusty wskaźnik (np. jeśli w zdarzeniu nie było żadnego depozytu energii), przed przejściem dalej należy to sprawdzić.
   Zwyczajowo sprawdzenie wygląda tak:

```
void EventAction::EndOfEventAction(const G4Event* event )
{
  G4HCofThisEvent* HCE = event->GetHCofThisEvent();
  if(!HCE) return;

...
}
```

3. Metoda **G4VHitsCollection* GetHC (G4int i)** klasy **GetHCofThisEvent()** zwraca kolekcję odpowiadającą zadanemu numerowi **i**. Typ **G4VHitsCollection** jest typem bazowym dla **G4THitsMap**<**T**> oraz G4THitsCollection < T > (którym zajmiemy się na przyszłych zajęciach), aby otrzymać typ G4THitsMap < G4double > * należy wykonać na niego rzutowanie (**dynamic_ cast** < **G4THitsMap** < **G4double** > * >).

Implementacja

4. Po długich bojach mamy już obiekt **G4THitsMap<G4double>** zawierający interesujące nas wyniki symulacji. Aby wyciągnąć te dane można skorzystać z operatora **[] (T* operator[] (G4int key) const)**. Proszę zwrócić uwagę na fakt, że operator przyjmuje klucz z mapy. Może się okazać, że mapa posiada tylko klucz równy np 4 (tylko 4 kopia detektora ma niezerowy depozyt energii). Nie można z góry zakładać, że klucz 0, 1, 2 itp istnieje. W przykładach, ktre widziałam użytkownicy zazwyczaj sprawdzają, czy mapa jest niezerowa dla kolejnych możliwych kluczy (z góry wiedząc ile może być kopii detektora). Alternatywnie można skorzystać z metody **std::map < G4int, T * > * GetMap ()const**, która zwraca typ **std::map**, na którym można już operować jak na każdej przyzwoitej mapie.
   Drugą trudnością jest fakt, że metoda **GetHC** zwraca wskaźnik do mapy, a operator **[]** wskaźnik do wartości, żeby uzyskać G4double musimy zrobić podwójne odwskaźnikowanie:
   G4double depozytEnergii =*((*mapaDepozytów)[klucz]);

Implementacja

„Klasyczne” (takie rozwiązanie znajdziecie w większości przykładów):

```
G4HCofThisEvent *hitsCollOfThisEvent = anEvent->GetHCofThisEvent();
if(!hitsCollOfThisEvent)
    return;
G4SDManager* SDmanager = G4SDManager::GetSDMpointer();
G4int siliCollId = SDmanager->GetCollectionID("siliSensitiveDet/eDep");

//wyciągamy mapę depozytów energii
G4THitsMap<G4double>* siliEnDep = dynamic_cast <G4THitsMap<G4double>* >
                                  (hitsCollOfThisEvent->GetHC(siliCollId));
for(int i =0; i!= 2; ++i)
{
    if((*siliEnDep)[i] != 0L)
    {
        G4double depozytEnergii =*((*siliEnDep)[i]);
        std::cout << "numer kopii: " << i
                  << " enDep: " << depozytEnergii/keV << std::endl;
    }
}
```

Implementacja

Znalazłam bardzo zgrabne podejście do analizy hitsMap, wykorzystujące dobrodziejstwo c++11:

```
G4double B4dEventAction::GetSum(G4THitsMap<G4double>* hitsMap) const
{
  G4double sumValue = 0.;
  for ( auto it : *hitsMap->GetMap() ) {
    // hitsMap->GetMap() returns the map of std::map<G4int, G4double*>
    sumValue += *(it.second);
  }
  return sumValue;
}
```

Zadanie

Znajdź liczbę cząstek wtórnych wytworzonych w fantomie wykorzystując odpowiedni prosty licznik. Porównaj otrzymane wielkości z wynikami otrzymanymi z kroku (tak, jak to robiliśmy na poprzednich zajęciach). Wypisz na ekran końcową liczbę cząstek wtórnych dla 100 eventów.