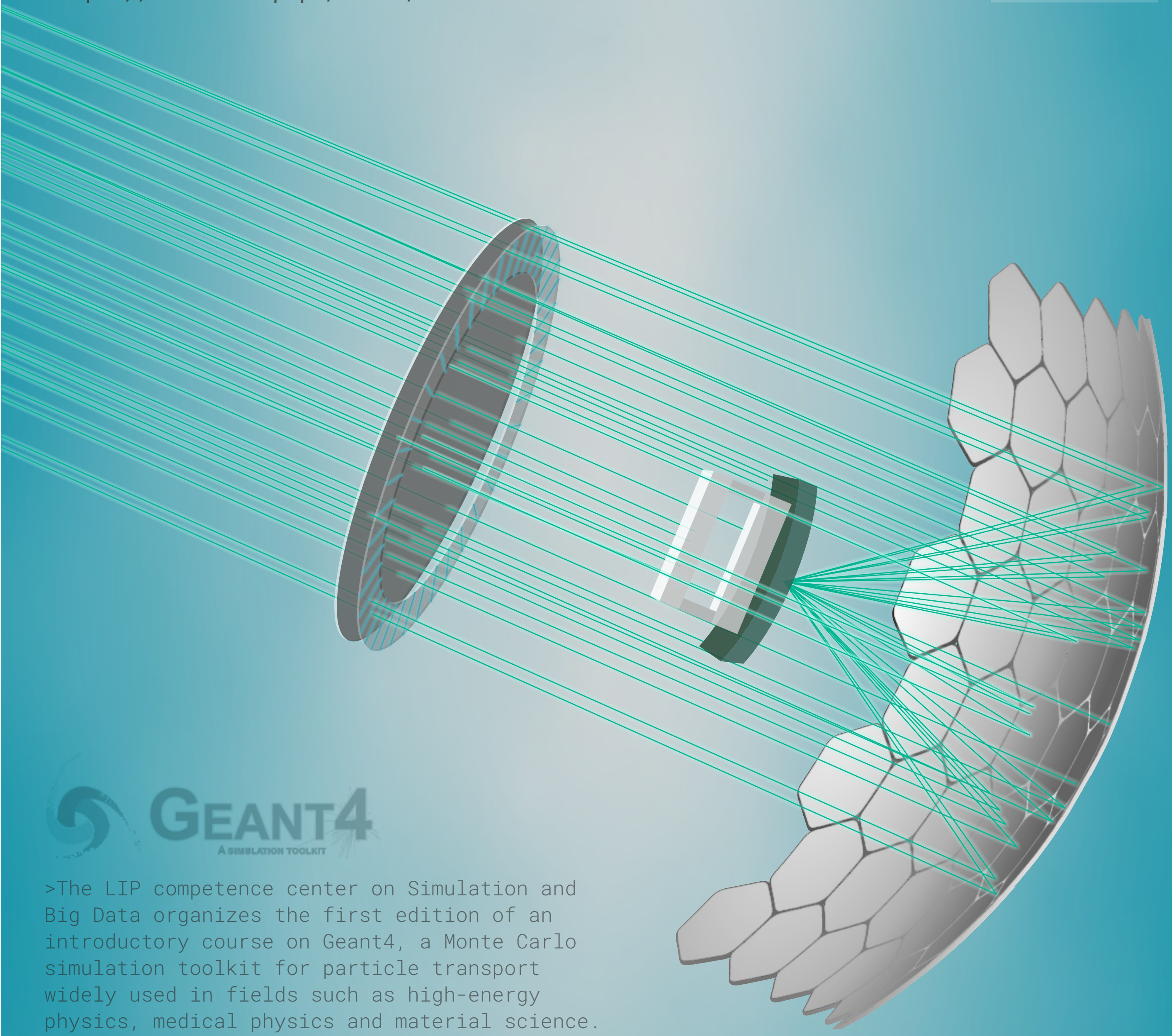
INTRODUCTORY COURSE ON

GEANT4

11-13 February 2020
University of Minho
Gualtar Campus, Braga

https://indico.lip.pt/event/681



>The LIP competence center on Simulation and Big Data organizes the first edition of an introductory course on Geant4, a Monte Carlo simulation toolkit for particle transport widely used in fields such as high-energy physics, medical physics and material science.

Organizing committee:
N. Castro, P. Gonçalves, A. Lindote, R. Sarmento, B. Tomé, M. Vasilevskiy

Optional

(but very useful)

Classes

1st LIP Introductory Course on GEANT4

Universidade do Minho, 11-13 Feb 2020

Basic structure of a GEANT4 simulation

- To create a simulation, the user must define (minimum):
  - ➡ *main()* - main program. Declare classes, initialise managers
  - ➡ *DetectorConstruction()* - geometry definitions (materials, volumes)
  - ➡ *PrimaryGenerator()* - define initial particles (*primaries*)
  - ➡ *PhysicsList()* - particles to use, associated processes and models

Mandatory classes

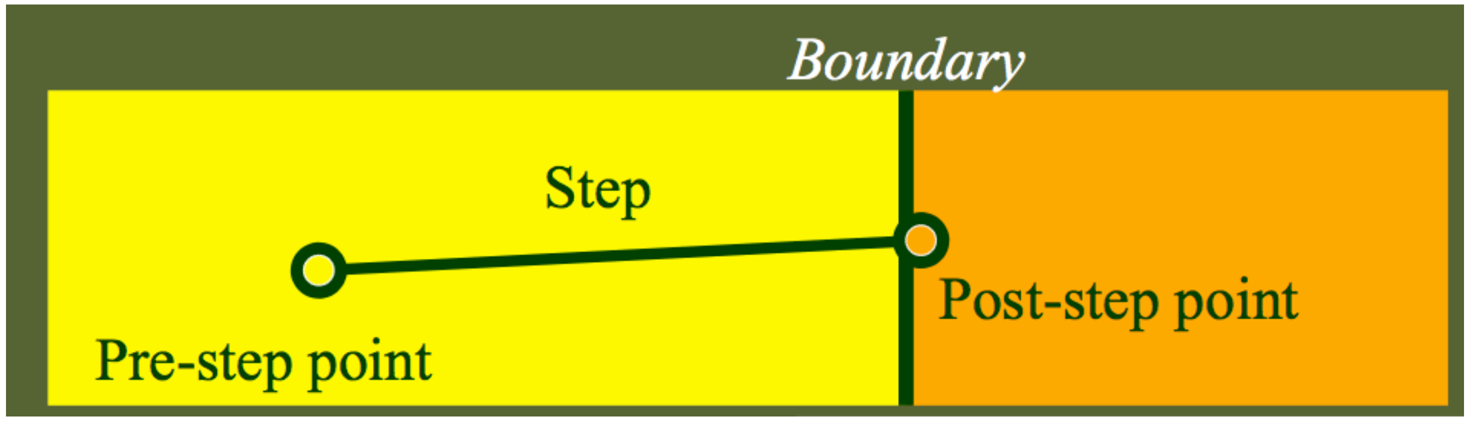
With these classes we have a working simulation

GEANT4 terminology

- **Events** are the basic units of a simulation (G4Event)
- At the beginning of an event, **primary tracks** are generated and stored in a **stack**
- The track is a snapshot of a **particle**, and it keeps its current information: energy, momentum, etc.
- GEANT4 gets a track from the stack and follows it in the geometry, until it exits the world or stops
  - If **secondary tracks** are produced in this process, they are also stored in the stack
  - When the stack is empty the event is completed

GEANT4 terminology

- Tracks are divided in **steps**
- A new step occurs every time the particle crosses a border or has an interaction
- A step has two points (**pre-step** and **post-step**) and G4Step stores information about both and the deltas (e.g. energy loss in the step)
- If a step is limited by a volume boundary, the post-step point is physically at the border, and associated with the next logical volume



GEANT4 terminology

- A **run** is a collection of events all with the same detector setup and physics settings
  - At the beginning of a run, geometry is optimised for navigation and cross section tables are (re)calculated
- In an analogy with real experiments, a run starts with a “beamOn” followed by the number of events to simulate

Optional classes

- Can be used to collect information from the simulation
- Control the simulation flow and behaviour
  - G4User**Run**Action
  - G4User**Event**Action
  - G4User**Tracking**Action
  - G4User**Stacking**Action
  - G4User**Stepping**Action
- **Use only the ones you need**

G4UserEventAction

- Provides 2 methods that are called by GEANT4 right before and right after processing each event
- **BeginOfEventAction()**
    - Called before primary tracks are generated
    - Can be used to initialise variables or event level histograms
- **EndOfEventAction()**
    - Can be used to perform analysis on data that was collected during each event
    - Write event level data to file, fill histograms

G4UserRunAction

- Has three methods the user can implement
- **GenerateRun()**
  - Invoked at the beginning of beamOn. Can be used to override the standard G4Run class (clearly too advanced for this course!)
- **BeginOfRunAction()**
  - Called before starting the event loop, can be used to set run conditions, initialise run level variables and histograms
- **EndOfRunAction()**
  - Can be used to do run level analysis, fill histograms, save data to file

G4UserStackingAction

- All the particles (both primary and secondary) created in a given event are stored in a list in the computer memory
(which is called *the* ***stack***)
- GEANT4 gets a particle from this list and follows it until it stops or leaves the geometry (or is killed by the user).
- If it produces secondary particles along the way, they are added to the list
- An event finishes when the stack is empty
- The StackingAction class allows us to manipulate that list
    - Pre-selection of tracks
    - Optimisation of the order of the track execution
    - Split events (e.g. for long decay chains)

G4UserStackingAction

- **ClassifyNewTrack()**
  - Called whenever a new track is placed on the stack
  - The user can change the track classification:
    - **fUrgent** - track is placed in the urgent stack
    - **fWaiting** - track is placed in the waiting stack, and will not be simulated until the urgent stack is empty
    - **fPostpone** - track is postponed to the next event
    - **fKill** - the track is deleted immediately and not stored in any stack.
- **NewStage()**
  - Called when the Urgent stack is empty and the Waiting stack has at least one track
- **PrepareNewEvent()**
  - Invoked at the beginning of each event, when the stacks are still empty

G4UserSteppingAction

- This class is called every time a particle has a step
- Independently of what particle is being tracked or the volume it is in
- Can make the simulation significantly slower
  - especially with complicated geometries
  - and events with many tracks
- But it’s the easiest and most straight-forward way of collecting information
- We will use it as a first approach to get data from our hands-on simulation *(more advanced/elegant strategies tomorrow)*

G4UserSteppingAction

- **UserSteppingAction()**
  - Provides a pointer to the current G4Step object
    - Get StepPoints and delta information, e.g.

```
G4double edep_inStep = thisStep->GetTotalEnergyDeposit();
```

  - From G4Step you can access the G4Track info
    - Get info on the particle, parent, current volume, etc.

```
G4Track* thisTrack = thisStep->GetTrack();
G4VPhysicalVolume* theVolume = thisTrack->GetVolume();
const G4ParticleDefinition* thisParticle = thisTrack->GetParticleDefinition();
```

    - You can kill the track if you're no longer interested in following it

```
thisTrack->SetTrackStatus(fStopAndKill);
```

```
thisTrack->SetTrackStatus(fKillTrackAndSecondaries);
```

Hands-On Session

Optional classes and information output

- In this hands-on exercise we will use the **SteppingAction** and **EventAction** classes to collect information from our simulation, do some basic event level analysis, and write to a file
- The goal is to write out the total energy deposited per event in the NaI crystal for various radioactive sources (^{22}Na, ^{60}Co, ^{137}Cs)
- As an example of a simple event level analysis, we will apply a Gaussian to the deposited energy in each event
  (use $\boldsymbol{\sigma} = \sqrt{(\mathrm{Edep})}$, with Edep in keV)
- Once you have an output, analyse the data using your favourite tool
  (the docker image includes ROOT, but feel free to use Matlab, GNUplot, etc)

Hands-On Session

- General instructions

  1. Open a file for writing in the EventAction class

  2. Define a public variable in the EventAction class which will be used to keep tally of the deposited energy
     (or be more elegant and create Set/Get methods, keeping the variable private)

  3. In the SteppingAction class, check if the interaction occurred inside your volume of interest

  4. If so, add the energy deposited in the step to the total (stored in the EventAction public variable)

  5. At the end of the event, write the total energy to the file
     (both the true and smeared energies)

Hands-On Session

- Run a simple analysis in ROOT
  - Download this script to your local folder
  - Open ROOT by typing `root` (or `root -l`)
  - Execute the script: `.x DoAnalysis.C`
  - To exit ROOT: `.q`

Detailed instructions

(use only if you get stuck!)

Hands-On Session

- Changes needed in the **EventAction** class
  - look in the `include/EventAction.hh` file, you already have
    - a file object declared (**bout**)
    - a public variable (**edep**) which you can use to accumulate the energy deposited in each step
  - what you need to do in `src/EventAction.cc`
    - associate the file object with a file. Do this in the constructor

```
bout.open("EnergyDeposition.txt");
```

    - close the file in the destructor

```
bout.close();
```

    - initialise the edep variable at the start of each event

```
edep = 0.;
```

    - write the edep variable to the file at the end of each event

```
bout << edep /keV << G4endl;
```

Hands-On Session

- Changes needed in the **SteppingAction** class
  - look inside the `include/SteppingAction.hh` file
    - you'll notice that we have a pointer to an EventAction object
  - what you need to change in the `src/SteppingAction.cc` file
    - note that the constructor receives a pointer to the EventAction, which is copied to the local pointer variable
    - there is already a lot of information collected in the method UserSteppingAction(). Use it to make a simple volume check (we're only interested in energy depositions inside the NaI crystal

```
if(thisVolumeName == "Crystal")
```

    - and add the deposited energy in the step to our EventAction edep variable

```
theEventAction->edep += edep_inStep;
```