Event Biasing

Geant4 Tutorial: version 10.0.p01

Michael Kelsey, Thu 6 Mar 2014

U.S. DEPARTMENT OF ENERGY
Office of Science

SLAC NATIONAL ACCELERATOR LABORATORY

Outline

SLAC

- Motivation
- What Is Event Biasing?
- Geometric Biasing in GEANT4
- Physics Biasing in GEANT4
- User Defined Biasing
- ***NEW*** General Biasing Interface
- Bremsstrahlung Splitting
- Summary

Motivation

If you want a fully realistic simulation, you need to generate, model, and track as much as possible of what would happen in real life.

"Too many events get rejected by my cuts."
"Why track secondaries in my support structure?"
"I don't care about 1 MeV photons!"
"GEANT4 is too slow."

If you need fast results, or high statistics, maybe give up some realism (or low-probability tails) in exchange for getting right answers on average, and worry about tails elsewhere.

What is Event Biasing?

Method of accelerating simulation of useful events at the expense of accurate fluctuations

Analogue simulation uses natural PDFs ***N(x)*** to generate correct mean and correct fluctuations, including far-off tails

- Often includes a significant fraction of events/particles outside final
- acceptance (physical or phase space) or interest
- Poor net efficiency

Biased simulation replaces ***N(x)*** with an artificial PDF ***B(x)***, which enhances production of interesting events/particles

- Increases MC efficiency: more events survive detector model, cuts, etc.
- Distribution and fluctuations are not correct: must apply weight correction, but still won't get tails

Geant4 Built-In Biasing (*Application Guide*, Sec 3.7)

SLAC

GEANT4 does analogue simulation by default

For expert/specialty users, provides methods to manipulate processing stages and apply ***B(x)*** bias early

Geometric or acceptance biasing

- Uses a combination of scoring and biasing functions
- Suppress/enhance events on the basis of coordinates or angles

Physics biasing

- Changes production of primary or secondary particles
- Changes relative branching fractions

User-defined biasing

- Available through the G4WrapperProcess interface
- For situations not covered by the built-in algorithms

Geometric Sampling (3.7.1)

Intended for **non-active** material, such as shielding

- Uses scoring to assign a weight to a generated track, and accept/reject algorithms to evaluate that weight
- **Importance sampling** uses geometrical splitting plus "Russian roulette" to select tracks through each cell
- **Weight roulette** uses windows (upper and lower bounds) or a simple cutoff to keep or remove tracks

Geometric Sampling (3.7.1)

Scoring done with `G4MultiFunctionalDetector`

- Scorers may themselves be biased; see `G4MultiFunctionDetector` (different class!)
- Scoring and importance sampling apply to particle types chosen by the user, not globally.
  - `examples/extended/biasing`
  - `examples/advanced/Tiara`

Geometry Models (3.7.1.1)

Biasing generally requires a parallel geometry equivalent to the detector model (mass geometry) used for simulation

- World volumes for parallel and mass must be identical
- Divide large non-sensitive volumes into cells (`G4GeometryCell`)
- Cells and parameters collected into a store (`G4IStore`, etc.)
- Volume must be fully populated with cells (no holes!)
- Cells must not share boundaries with world volume

Geometry Models (3.7.1.1)

SLAC

```
class B02ImportanceDetectorConstruction : public G4VUserParallelWorld
{ ... };

G4VPhysicalVolume* ghostWorld = pdet->GetWorldVolume();
G4GeometrySampler pgs(ghostWorld,"neutron");
pgs.SetParallel(true);
...
pgs.PrepareImportanceSampling(&aIstore, 0);
pgs.Configure();
```

Importance Sampling (3.7.1.3)

Must have good understanding of problem physics

- Which particle types require importance sampling?
- Define the cells appropriately (size, location)
- Assign importance values to the cells

If not done properly, results cannot be interpreted as describing real experiment

Importance store used to store values related to cells

- User creates an object which inherits from `G4VIstore`: built-in `G4IStore` may be used
- Constructed with reference to the world-volume of the geometry (mass or parallel) used for sampling
- User fills the store with cells and their importance values

Importance Store

SLAC

```
class G4IStore : public G4VIStore {
public:
  explicit G4IStore(const G4VPhysicalVolume &worldvolume);
  virtual ~G4IStore();
  virtual G4double GetImportance(const G4GeometryCell &gCell) const;
  virtual G4bool IsKnown(const G4GeometryCell &gCell) const;
  virtual const G4VPhysicalVolume &GetWorldVolume() const;
  void AddImportanceGeometryCell(G4double importance,
  const G4GeometryCell &gCell);
  void AddImportanceGeometryCell(G4double importance,
  const G4VPhysicalVolume &, G4int aRepNum=0);
  void ChangeImportance(G4double importance, const G4GeometryCell &gCell);
  void ChangeImportance(G4double importance, const G4VPhysicalVolume &,
  G4int aRepNum=0);
  G4double GetImportance(const G4VPhysicalVolume &, G4int aRepNum=0) const;
  private: .....
};
```

Importance Sampling Algorithm

Value must be assigned to every cell

Store must be fully occupied (no missing cells)

User creates class inheriting from `G4VImportanceAlgorithm`

- Built-in `G4ImportanceAlgorithm` will be used if none passed to sampler
  - **Cell is not in store**: causes an exception (job terminates)
  - **Importance = 0**: Tracks of the chosen particle type will be killed
  - **Importance > 0**: Normal allowed values
  - **Importance < 0**: Not allowed!
- User-defined algorithm must handle all cases above, perhaps with different behaviour

Weight Windows (3.7.1.5)

Alternative to importance sampling

- Applies splitting and Russian roulette depending on space (cells) and energy
- User defines weight windows in contrast to defining importance values as in importance sampling
- Importance sampling is “weight blind,” this technique uses particle weight in evaluation

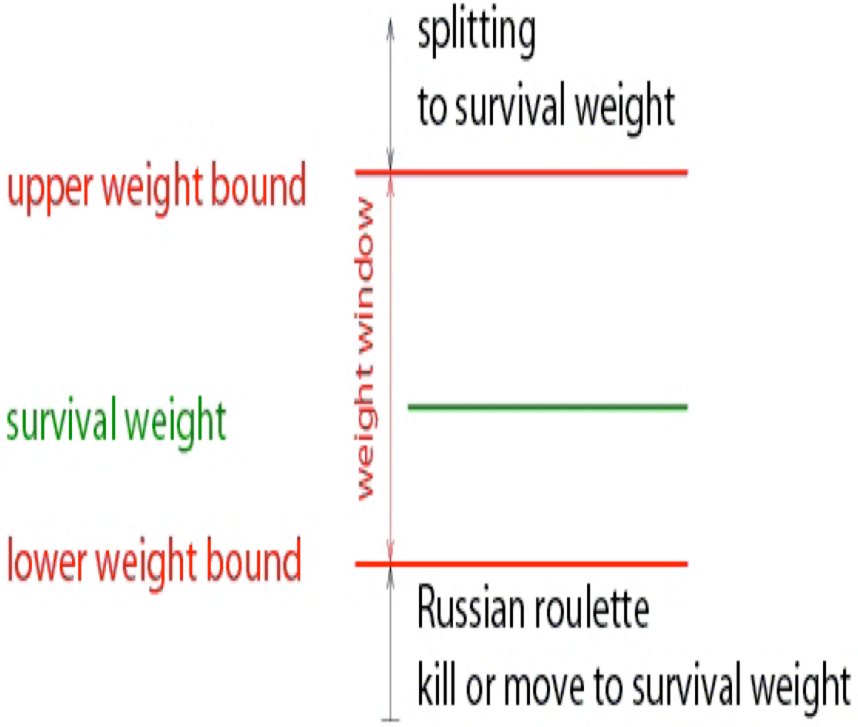
Apply in combination with other variance reduction techniques

- **cross-section biasing**, **implicit capture**
- A weight window may be specified for every geometric cell and for different energy ranges
- **Space-energy cell**, using same `G4GeometryCell` as importance sampling
- Cells with no window will pass all particles through

Weight Window Definition

SLAC

- **Global** lower bound W_L (all cells)
- Scale factors C_S and C_U for each cell, or globally
- Upper bound $W_U = C_U W_L$
- Survival cut $W_S = C_S W_L$
- $C_S = C_U = 1$ equivalent to importance sampling
- User may apply at boundaries, on collisions or both



Cells stored in subclass of `G4VWeightWindowStore`

- Built-in implementation `G4WeightWindowStore` available
- Tracks below W_L may be killed, or weight reset to W_S
- Tracks above W_U are split (replicated n times, $nW_S \leq W_U$)

Weight roulette (3.7.1.6)

Also called **weight cutoff**

- Weight of particle may become so low that no result can change significantly: propagating all tracks wastes computing time.
- Usually applied if importance sampling and implicit capture are used together.
- **Implicit capture** reduces particle's weight at every collision, instead of killing outright with some probability.

Weight roulette scales particle's weight by **importance ratio**

$R = I_s/I_c$ of current cell (I_c) and original source (I_s)

- If weight falls below a lower bound, Russian roulette is applied
- If particle survives, weight is reset to a specified survival weight

Physics Based Biasing (3.7.2)

Replaces the natural distribution of some process with "fake" PDFs that limit events to what is useful for your simulation

- **Primary particle** biasing: e.g., cosmic ray experiments
- **Radioactive decay** biasing: e.g., shielding or underground detectors
- **Hadronic leading-particle** biasing: only the highest-energy secondary(ies) at each step of a shower are kept
- **Hadronic cross-section** biasing: cross-sections or branching ratios can be arbitrarily rescaled

Primary Particle (3.7.2.1.1)

Increases number of primary particles generated

in particular phase space region of interest

- Primary particle's weight is modified as appropriate
- Implemented in `G4GeneralParticleSource`
- Possible to bias position, angular and/or energy distributions

Use in subclass of `G4VUserPrimaryGeneratorAction`

```
MyPrimaryGeneratorAction::MyPrimaryGeneratorAction() {
  generator = new G4GeneralParticleSource;
}

void MyPrimaryGeneratorAction::GeneratePrimaries(G4Event* anEvent) {
  generator->GeneratePrimaryVertex(anEvent);
}
```

Radioactive decay (3.7.2.1.2)

SLAC

`G4RadioactiveDecay` simulates the decay of radioactive nuclei, with optional biasing

- **Increase sampling rate** of radionclides within observation times through user-defined probability distribution function
- **Nuclear splitting**, where parent nuclide is split into a user
- defined number of nuclides
- **Branching ratio biasing** where branching ratios are sampled with equal probability

This is a *process*: register in physics list for `G4GenericIon`

Examples in `examples/extended/radioactivedecay`

Hadronic Leading Particle (3.7.2.1.3)

Keeps only most important part of each event, as well as representative tracks of each given particle type:

- Track with highest energy
- One of each of baryon (p, n), π^0, meson ($\pi^{\pm}$, K), lepton
- Appropriate weights are assigned to the particles

Implemented in `G4HadLeadBias` utility

Environment variable **`SwitchLeadBiasOn`** activates

Hadronic cross-section (3.7.2.1.4)

Artificially enhances/reduces cross section(s) for some process

- Useful for studying thin layer interactions or thick layer shielding
- Photon inelastic, electron nuclear and positron nuclear processes

More details can be found in a talk presented at TRIUMF

http://legacyweb.triumf.ca/geant4-03/talks/03-Wednesday-AM-1/03-J.Wellisch/biasing.hadronics.pdf

Cross-Section Biasing

SLAC

Controlled via

`G4HadronicProcess::BiasCrossSectionByFactor()`

```
void MyPhysicsList::ConstructProcess() {
...
  G4ElectroNuclearReaction *theElectroReaction =
    new G4ElectroNuclearReaction;
  G4ElectronNuclearProcess theElectronNuclearProcess;
  theElectronNuclearProcess.RegisterMe(theElectroReaction);
  theElectronNuclearProcess.BiasCrossSectionByFactor(100);
  pManager->AddDiscreteProcess(&theElectronNuclearProcess);
...
}
```

User Defined Biasing (3.7.2.2)

SLAC

`G4WrapperProcess` can be used to implement user defined event biasing

- `G4WrapperProcess`, which is a process itself, wraps an existing process
- All function calls forwarded to wrapped process
- Non-invasive way to modify behaviour of existing (built-in) process

1. Create derived class inheriting from `G4WrapperProcess`
2. Override only the methods to be modified, e.g., `PostStepDoIt()`
3. Register this class in place of the original
4. Finally, register the original (wrapped) process with user class

User Defined Biasing

SLAC

```
class MyWrapperProcess : public G4WrapperProcess {
  ...
  G4VParticleChange* PostStepDoIt(const G4Track& track,
                                  const G4Step& step) {
  // Do something interesting
  }
};

void MyPhysicsList::ConstructProcess() {
  ...
  G4LowEnergyBremsstrahlung* bremProcess =
    new G4LowEnergyBremsstrahlung();
  MyWrapperProcess* wrapper = new MyWrapperProcess();
  wrapper->RegisterProcess(bremProcess);
  processManager->AddProcess(wrapper);
}
```

General Biasing Interface (new for G4 10.0)

More “toolkit-based” approach to user-defined biasing, with pre-written tools using common interface.

`G4BiasingProcessInterface`

- Wrapper class for physics processes

`G4VBiasingOperation`

- Base class to define single biasing action

`G4VBiasingOperator`

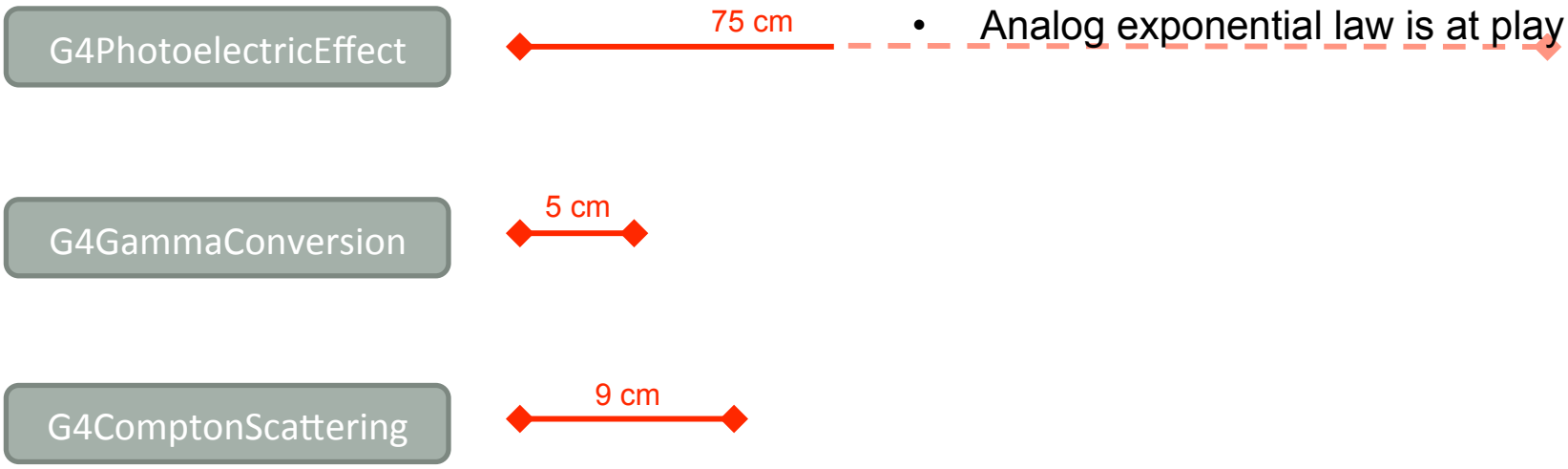
- Configuration to apply one or more operations

General Biasing: Discrete Process Example

SLAC

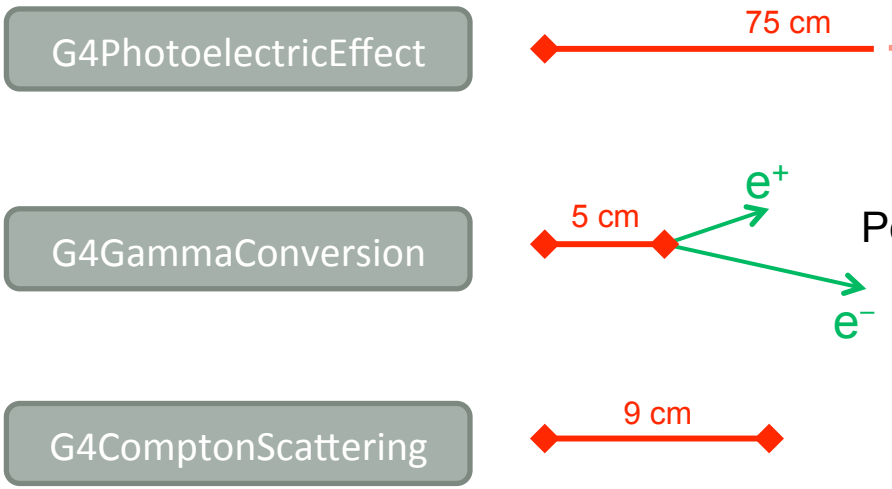
GetPostStepPhysicalInteractionLength()

- Returns the distance at which the process will make an interaction
- Analog exponential law is at play



General Biasing: Discrete Process Example

SLAC



GetPostStepPhysicalInteractionLength()

- Returns the distance at which the process will make an interaction
- Analog exponential law is at play

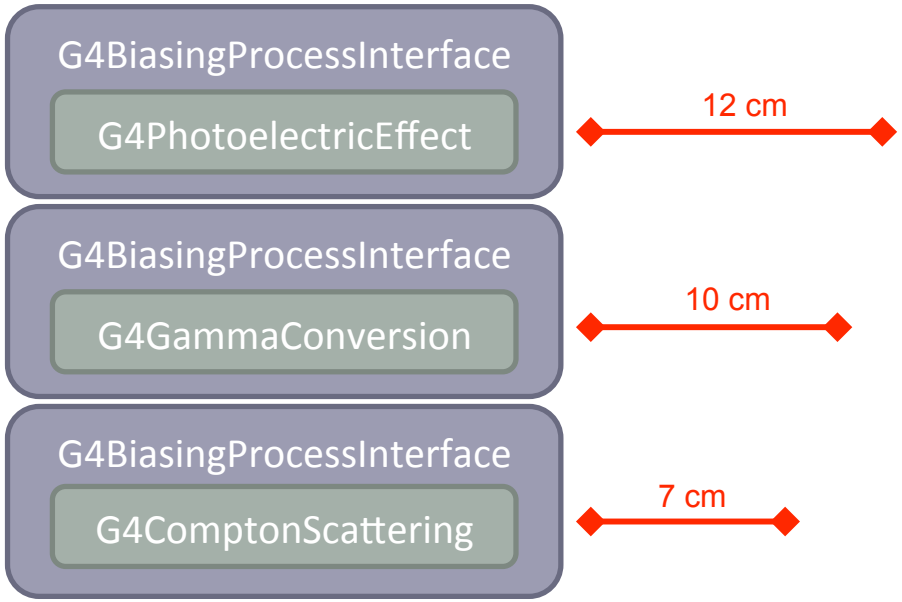
PostStepDoIt()

- Called if the process has responded the shortest of the interaction distances
- Generate final state, according to specific process analog physical law

General Biasing: Discrete Process Example

SLAC

`G4BiasingProcessInterface` wrapper intercepts physics-related function calls



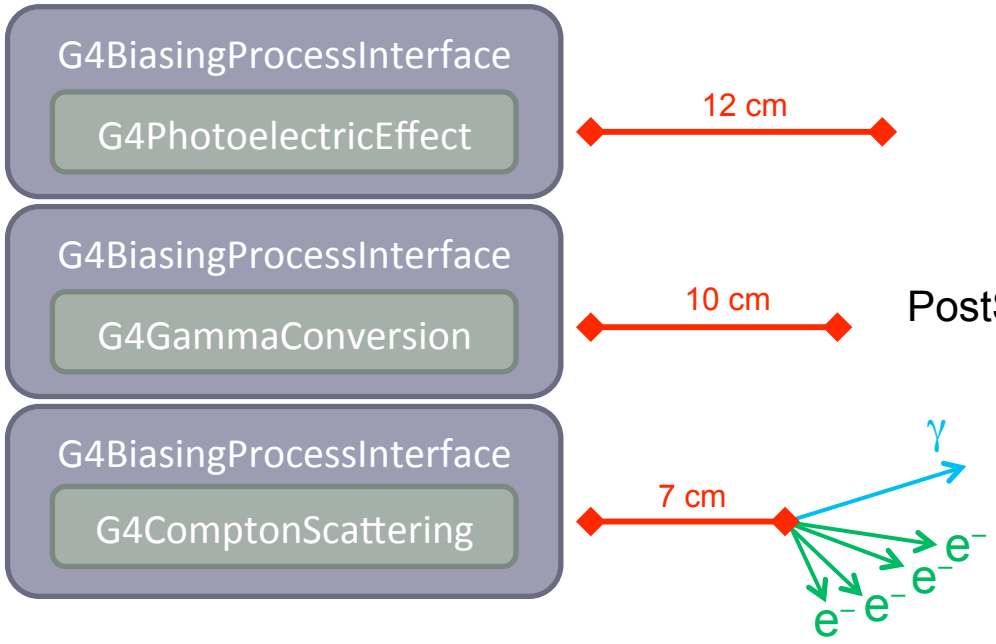
GetPostStepPhysicalInteractionLength()

- Returns the distance at which the process will make an interaction
- ~~Analog exponential law is at play~~
- Above analog behavior superseded by biased behavior (if wished)

General Biasing: Discrete Process Example

SLAC

`G4BiasingProcessInterface` wrapper intercepts physics-related function calls



GetPostStepPhysicalInteractionLength()

- Returns the distance at which the process will make an interaction
- ~~Analog exponential law is at play~~
- Above analog behavior superseded by biased behavior (if wished)

PostStepDoIt()

- Called if the process has responded the shortest of the interaction distances
- ~~Generate final state, according to specific process analog physical law~~
- Above analog behavior superseded by biased behavior (if wished)

Example: Bremsstrahlung Splitting

- Implemented via `G4WrapperProcess` (details shown)
- Assume only interested in scoring photon hits
- Increase MC performance by reducing tracking of secondary electrons
- Demonstrates biasing through enhanced production of secondaries

Brem Splitting Algorithm

Sample photon energy, angular distributions *N* times

Generate *N unique* secondaries (vs. once per interaction)

- This is called “splitting”
- Don’t confuse with importance sample splitting, where *N* *identical* copies are created

Reduce electron energy by just one of the chosen photons

Remove bias introduced in photon energy and angular distributions

- Assign each secondary a statistical weight
- $w_{sec} = w_{parent} / N$

Brem Splitting Implementation

User process inherits from `G4WrapperProcess`

```
class BremSplittingProcess : public G4WrapperProcess {
public:
  BremSplittingProcess();
  virtual ~BremSplittingProcess();

  // Override only this method
  G4VParticleChange* PostStepDoIt(const G4Track& track,
                                  const G4Step& step);
private:
  G4int fNSplit;       // Number of secondaries per split
};
```

Brem Splitting Implementation

SLAC

```
G4VParticleChange*
BremSplittingProcess::PostStepDoIt(const G4Track& track,
                                   const G4Step& step) {
  G4double weight = track.GetWeight()/fNSplit;
  ...
  std::vector<G4Track*> secondaries(fNSplit); // Secondary store

  // Loop over wrapped PSDI method to generate multiple secondaries
  for (G4int i=0; i<fNSplit; i++) {
    particleChange = pRegProcess->PostStepDoIt(track, step);
    assert (0 != particleChange);
    particleChange->SetVerboseLevel(0);

    // Save the secondaries generated on this cycle
    for (G4int j=0; j<particleChange->GetNumberOfSecondaries(); j++) {
      secondaries.push_back(new G4Track(*(particleChange->GetSecondary(j))));
    }
  }
  ...
}
```

Brem Splitting Implementation

Register wrapped process and wrapper with process manager

```
G4LowEnergyBremsstrahlung* bremProcess =
  new G4LowEnergyBremsstrahlung();

BremSplittingProcess* bremSlitting =
  new BremSplittingProcess;

bremSplitting->RegisterProcess(bremProcess);
pmanager->AddProcess(bremSplitting);
```

Uniform Bremsstrahlung Splitting

SLAC