Physics Biasing

Alberto Ribon
CERN EP/SFT

Geant4 Virtual Tutorial, 28 September – 2 October 2020

Outline

- Biasing in Hadronic Physics
  - Second part: the “new”, recommended way

Generic Bias for Hadronics (1/2)

- This is the new and recommended approach for biasing in Geant4 – not only for hadronics !
  - Allow to mix biasing options via “building blocks” (instead of built-in functionalities)
- Examples available in:

  examples/extended/biasing/ and ***examples/extended/hadronic/***

  we discuss here the following ones (relevant for hadronics):

  1. ***GB01/*** : cross-sections biasing (*i.e.* changing the natural cross sections)
  2. ***GB02/*** : force-collision biasing (*i.e.* forcing an interaction in a volume)
  3. ***GB07/*** : leading-particle biasing (*i.e.* keep leading particle and one particle per species)
  4. ***Hadr08/*** : hadronic-model per-region (*i.e.* using a different hadronic model in a region)

Generic Bias for Hadronics (2/2)

- Notes:
  - there is plenty of user code, but nearly all of it can be re-used: only a tiny part needs to be customized per use-case !
  - At any level (*e.g.* stepping action or sensitive detector) the **statistical weight of a track** can be obtained as:

    w = track->GetWeight()

Cross-Section Generic Biasing (GB01)

```
int main( … ) {
   …
   FTFP_BERT* physicsList = new FTFP_BERT;
   G4GenericBiasingPhysics* biasingPhysics = new G4GenericBiasingPhysics;
   biasingPhysics->Bias( "gamma" );
   biasingPhysics->Bias( "neutron" );
   physicsList->RegisterPhysics( biasingPhysics );
   …
}

void GB01DetectorConstruction::ConstructSDandField() {
   …
   GB01BOptrMultiParticleChangeCrossSection* biasingOperator =
         new GB01BOptrMultiParticleChangeCrossSection;
   biasingOperator->AddParticle( "gamma" );
   biasingOperator->AddParticle( "neutron" );
   biasingOperator->AttachTo( logicVolumeToBias );
}
```

Enable biasing only for a subset of particle types

Possible to define xsec biasing for sets of particle types in specified logical volumes

```
class GB01BOptrMultiParticleChangeCrossSection : public G4VBiasingOperator {
  public:
    …
    void AddParticle( G4String particleName );
    void StartTracking( const G4Track* track );
  private:
    virtual G4VBiasingOperation* ProposeOccurenceBiasingOperation(...);
    virtual void OperationApplied(...);
    …
    std::map< const G4ParticleDefinition*, GB01BOptrChangeCrossSection* > fBOptrForParticle;
    std::vector< const G4ParticleDefinition* > fParticlesToBias;
    GB01BOptrChangeCrossSection* fCurrentOperator;
    G4int fnInteractions;
};

void GB01BOptrMultiParticleChangeCrossSection::AddParticle( G4String particleName ) {
  const G4ParticleDefinition* particle = G4ParticleTable::GetParticleTable()->FindParticle( particleName );
  …
  GB01BOptrChangeCrossSection* optr = new GB01BOptrChangeCrossSection( particleName );
  fParticlesToBias.push_back( particle );
  fBOptrForParticle[ particle ] = optr;
}

void GB01BOptrMultiParticleChangeCrossSection::StartTracking( const G4Track* track )
   // Fetch the underneath biasing operator, if any, for the current particle type and store it in fCurrentOperator
}
```

```
G4VBiasingOperation* GB01BOptrMultiParticleChangeCrossSection::
ProposeOccurenceBiasingOperation( const G4Track* track,
                                  const G4BiasingProcessInterface* callingProcess ) {
  // Examples of limitations imposed to apply the biasing:
  if ( track->GetParentID() != 0 )  return 0;  // Limit application of biasing to primary particles only
  if ( fnInteractions > 4 )         return 0;  // Limit to at most 5 biased interactions
  if ( track->GetWeight() < 0.05 ) return 0;  // Limit to a weight of at least 0.05
  if ( fCurrentOperator ) return fCurrentOperator->
                                 GetProposedOccurenceBiasingOperation( track, callingProcess );
  else return 0;
}

void GB01BOptrMultiParticleChangeCrossSection::OperationApplied(...) {
  fnInteractions++;  // Count number of biased interactions
  // Inform the underneath biasing operator that a biased interaction occured:
  if ( fCurrentOperator ) fCurrentOperator->ReportOperationApplied(...);
}
```

```
class GB01BOptrChangeCrossSection : public G4VBiasingOperator {
  public:
    GB01BOptrChangeCrossSection( G4String particleToBias, G4String name = "ChangeXS" );
    ...
    virtual void StartRun();
  private:
    virtual G4VBiasingOperation* ProposeOccurenceBiasingOperation(...);
     ...
    using G4VBiasingOperator::OperationApplied;
    virtual void OperationApplied(...);
    std::map< const G4BiasingProcessInterface*,  G4BOptnChangeCrossSection* > fChangeCrossSectionOperations;
    const G4ParticleDefinition* fParticleToBias;
};

void GB01BOptrChangeCrossSection::StartRun() {
   const G4ProcessManager* processManager = fParticleToBias->GetProcessManager();
   const G4BiasingProcessSharedData* sharedData = G4BiasingProcessInterface::GetSharedData( processManager );
   for ( size_t i = 0 ; i < (sharedData->GetPhysicsBiasingProcessInterfaces()).size(); i++ ) {
     const G4BiasingProcessInterface* wrapperProcess = (sharedData->GetPhysicsBiasingProcessInterfaces())[i];
     G4String operationName = "Xschange-" + wrapperProcess->GetWrappedProcess()->GetProcessName();
     fChangeCrossSectionOperations[ wrapperProcess ] = new G4BOptnChangeCrossSection( operationName );
   }
}
```

```
G4VBiasingOperation* GB01BOptrChangeCrossSection::ProposeOccurenceBiasingOperation(...) {
  if ( track->GetDefinition() != fParticleToBias ) return 0;
  G4double analogInteractionLength =
              callingProcess->GetWrappedProcess()->GetCurrentInteractionLength();
  if ( analogInteractionLength > DBL_MAX/10.0 ) return 0;
  G4double analogXS = 1.0/analogInteractionLength;
  G4BOptnChangeCrossSection* operation = fChangeCrossSectionOperations[ callingProcess ];
  ...
  operation->SetBiasedCrossSection( 2.0 * analogXS );  //<--- Scaling factor for the xsec !
  operation->Sample();
  …
  return operation;
}

void GB01BOptrChangeCrossSection::OperationApplied(...) {
  G4BOptnChangeCrossSection* operation = fChangeCrossSectionOperations[ callingProcess ];
  if ( operation == occurenceOperationApplied ) operation->SetInteractionOccured();
}
```

Alternatively, one could use different cross section scaling factors according to the particle type and/or process type

Force-Collision Generic Biasing (GB02)

- It forces a process to occur in a given volume for a given particle type
- Useful, for instance, when you are interested to study a hadronic inelastic interactions in a thin or light target
- Be careful: in its present form, it forces the occurrence of any of the processes associated to the particle type we want to bias, maintaining the relative probabilities (*e.g.* cross sections) of these processes
  - For instance, for a proton on a thin layer of Silicon, one of the following 4 processes can occur: ionization; bremsstrahlung; proton elastic; proton inelastic
  - In future versions, there might be the feature of forcing only the inelastic process

```
int main( … ) {
   …
   FTFP_BERT* physicsList = new FTFP_BERT;
   G4GenericBiasingPhysics* biasingPhysics = new G4GenericBiasingPhysics;
   biasingPhysics->Bias( "gamma" );
   biasingPhysics->Bias( "neutron" );
   physicsList->RegisterPhysics( biasingPhysics );
   …
}

void GB02DetectorConstruction::ConstructSDandField() {
   …
  GB02BOptrMultiParticleForceCollision* biasingOperator =
      new GB02BOptrMultiParticleForceCollision;
   biasingOperator->AddParticle( "gamma" );
   biasingOperator->AddParticle( "neutron" );
   biasingOperator->AttachTo( logicVolumeToBias );
}
```

```
class GB02BOptrMultiParticleForceCollision : public G4VBiasingOperator {
  public:
    ...
    void AddParticle( G4String particleName );  // Declare particles to be biased
    virtual void StartTracking( const G4Track* track );
  private:
    virtual G4VBiasingOperation* ProposeNonPhysicsBiasingOperation(...);
    virtual G4VBiasingOperation* ProposeOccurenceBiasingOperation(...);
    virtual G4VBiasingOperation* ProposeFinalStateBiasingOperation(...);
    void OperationApplied(...);
    void ExitBiasing(...);
    std::map< const G4ParticleDefinition*, G4BOptrForceCollision* > fBOptrForParticle;
    std::vector< const G4ParticleDefinition* > fParticlesToBias;
    G4BOptrForceCollision* fCurrentOperator;
};


void GB02BOptrMultiParticleForceCollision::AddParticle( G4String particleName ) {
  const G4ParticleDefinition* particle = G4ParticleTable::GetParticleTable()->FindParticle( particleName );
  if ( particle == 0 ) { … }  // just warning exception and return
  G4BOptrForceCollision* optr = new G4BOptrForceCollision( particleName,
                                                           "ForceCollisionFor" + particleName );
  fParticlesToBias.push_back( particle );
  fBOptrForParticle[ particle ] = optr;
}
```

```
G4VBiasingOperation* GB02BOptrMultiParticleForceCollision::ProposeOccurenceBiasingOperation(...) {
  if ( fCurrentOperator ) return fCurrentOperator->GetProposedOccurenceBiasingOperation( track, callingProcess );
  else                    return 0;
}
```

Possible to impose here some limitations on when to apply biasing, *e.g.* only for a certain process type

```
G4VBiasingOperation* GB02BOptrMultiParticleForceCollision::ProposeFinalStateBiasingOperation(...) {
  if ( fCurrentOperator ) return fCurrentOperator->GetProposedFinalStateBiasingOperation( track, callingProcess );
  else                    return 0;
}

void GB02BOptrMultiParticleForceCollision::StartTracking( const G4Track* track ) {
  const G4ParticleDefinition* definition = track->GetParticleDefinition();
  std::map< const G4ParticleDefinition*, G4BOptrForceCollision* >:: iterator it = fBOptrForParticle.find( definition );
  fCurrentOperator = 0;
  if ( it != fBOptrForParticle.end() ) fCurrentOperator = (*it).second;
}

void GB02BOptrMultiParticleForceCollision::OperationApplied(...) {
  if ( fCurrentOperator ) fCurrentOperator->ReportOperationApplied(...);
}
void GB02BOptrMultiParticleForceCollision::OperationApplied(...) {
  if ( fCurrentOperator ) fCurrentOperator->ReportOperationApplied(...);
}
void GB02BOptrMultiParticleForceCollision::ExitBiasing(...) {
  if ( fCurrentOperator ) fCurrentOperator->ExitingBiasing( track, callingProcess );
}
```

Leading-Particle Generic Biasing (GB07)

- Useful technique, in particular for shielding applications
- In an interaction, this biasing scheme returns a reduced set of secondaries: the most energetic secondary (called the “leading” particle, with weight equals to 1) and one secondary randomly chosen for each species (with weight given by the inverse of the number of secondaries of each species)
  - Particles and anti-particles are considered to belong to the same “species”
  - There is a further option – not used in our example – to apply a further reduction of the secondaries via Russian roulette, useful for electron/positron bremmstrahlung emission, positron annihilation, and π° decay

```
int main( … ) {
   …
   FTFP_BERT* physicsList = new FTFP_BERT;
   G4GenericBiasingPhysics* biasingPhysics = new G4GenericBiasingPhysics;
   std::vector< G4String > protonProcessesToBias;
   protonProcessesToBias.push_back( "protonInelastic" );
   biasingPhysics->PhysicsBias( "proton", protonProcessesToBias );
   physicsList->RegisterPhysics( biasingPhysics );
   …
}
```

Specify which particles to bias and, for each of them, which processes to bias

```
void GB02DetectorConstruction::ConstructSDandField() {
   …
   GB07BOptrLeadingParticle* biasingOperator =  new GB07BOptrLeadingParticle;
   biasingOperator->AttachTo( logicVolumeToBias );
   ...
}
```

```
class GB07BOptrLeadingParticle : public G4VBiasingOperator {
  public:
    GB07BOptrLeadingParticle( G4String operatorName = "LeadingParticleBiasingOperator");
    virtual ~GB07BOptrLeadingParticle();
    virtual void StartRun() final;
    virtual void StartTracking( const G4Track* track ) final;

  private:
    // Mandatory from base class: Unused:
    virtual G4VBiasingOperation*  ProposeNonPhysicsBiasingOperation(...) final { return nullptr; }
    virtual G4VBiasingOperation*  ProposeOccurenceBiasingOperation (...) final { return nullptr; }

    // Mandatory from base class: Used:
    // Will return the biasing operation at the final state generation stage
    virtual G4VBiasingOperation*  ProposeFinalStateBiasingOperation( const G4Track* track,
                                                                     const G4BiasingProcessInterface* callingProcess ) final;

    // The leading particle biasing operation that will actually trim the final state generation:
    G4BOptnLeadingParticle* fLeadingParticleBiasingOperation;

};
```

```
GB07BOptrLeadingParticle::GB07BOptrLeadingParticle( G4String operatorName ) :
   G4VBiasingOperator( operatorName ) {
      fLeadingParticleBiasingOperation = new G4BOptnLeadingParticle( "LeadingParticleBiasingOperation" );
}


GB07BOptrLeadingParticle::~GB07BOptrLeadingParticle() {
   delete fLeadingParticleBiasingOperation;
}


GB07BOptrLeadingParticle::
ProposeFinalStateBiasingOperation( const G4Track*, const G4BiasingProcessInterface* callingProcess ) {
   fLeadingParticleBiasingOperation->SetFurtherKillingProbability( -1.0 );
   return fLeadingParticleBiasingOperation;
}
```

Hadronic-model per-region (Hadr08)

- Geant4 does not allow physics lists per region.
  However, it is possible to specify EM models per region.
  But the hadronic framework does not allow hadronic models per region
- This final example shows how it is possible to achieve in practice "HAD models per region" by using "generic biasing"
  - We use the powerful "generic biasing" machinery available in Geant4, but the actual weights of all tracks remain to the usual value (1.0) as in the normal (unbiased) case
  - In our example (inspired by a real application in ALICE) we want to use the physics lists FTFP_BERT everywhere, except in the tracker region, where we want to use the a more accurate but slower intranuclear cascade model, INCLXX, instead of BERT
    - In the case of ALICE, replacing FTFP_BERT with FTFP_INCLXX would slow down the performance of the simulation by a factor of 2, whereas with this trick they can keep practically the same computing performance

```
int main( … ) {
   …
   FTFP_BERT* physicsList = new FTFP_BERT;
   G4GenericBiasingPhysics* biasingPhysics = new G4GenericBiasingPhysics;
   biasingPhysics->Bias( "proton" );
   physicsList->RegisterPhysics( biasingPhysics );
   …
}

void GB02DetectorConstruction::ConstructSDandField() {
   …
   BiasingOperator* biasingOperator = new BiasingOperator;
   biasingOperator->AddParticle( "proton" );
   biasingOperator->AttachTo( trackerLogicalVolume );
}
```

```
class BiasingOperator : public G4VBiasingOperator {
  public:
    BiasingOperator();
    virtual ~BiasingOperator() {}
    void AddParticle( G4String particleName );
    virtual G4VBiasingOperation* ProposeFinalStateBiasingOperation( const G4Track* track,
                                                                    const G4BiasingProcessInterface* callingProcess ) final;
    // Not used:
    virtual G4VBiasingOperation* ProposeNonPhysicsBiasingOperation(...) { return 0; }
    virtual G4VBiasingOperation* ProposeOccurenceBiasingOperation(...) { return 0; }

  private:
    std::vector< const G4ParticleDefinition* > fParticlesToBias;
    BiasingOperation* fBiasingOperation;
};
```

```
BiasingOperator::BiasingOperator() : G4VBiasingOperator( "BiasingOperator" ) {
  fBiasingOperation = new BiasingOperation( "BiasingOperation" );
}


void BiasingOperator::AddParticle( G4String particleName ) {
  const G4ParticleDefinition* particle = G4ParticleTable::GetParticleTable()->FindParticle( particleName );
  …  // check that particle is not nullptr
  fParticlesToBias.push_back( particle );
}


G4VBiasingOperation* BiasingOperator::ProposeFinalStateBiasingOperation( const G4Track* ,
                                                                         const G4BiasingProcessInterface* callingProcess ) {
  // Apply the biasing operation only for proton inelastic process
  if ( callingProcess  &&  callingProcess->GetWrappedProcess()  &&
       callingProcess->GetWrappedProcess()->GetProcessName() == "protonInelastic" ) {
    return fBiasingOperation;
  } else {
    return 0;
  }
}
```

```
class BiasingOperation : public G4VBiasingOperation {
  public:
    ...
    virtual G4VParticleChange* ApplyFinalStateBiasing(...);

    // Unused :
    virtual const G4VBiasingInteractionLaw* ProvideOccurenceBiasingInteractionLaw( ...) { return 0; }
    virtual G4double DistanceToApplyOperation( ...) { return DBL_MAX; }
    virtual G4VParticleChange* GenerateBiasingFinalState(...) { return 0; }

  private:
    G4ProtonInelasticProcess*  fProtonInelasticProcess;
};

BiasingOperation::BiasingOperation( G4String name ) : G4VBiasingOperation( name ) {
  fProtonInelasticProcess = new G4ProtonInelasticProcess;
  // Create hadronic models and cross sections for the alternative "FTFP_INCLXX"
  // to be used only in the TrackerLogicalVolume , and register them in fProtonInelasticProcess
  ...
}

G4VParticleChange* BiasingOperation::ApplyFinalStateBiasing( const G4BiasingProcessInterface* ,
                                                             const G4Track* track, const G4Step* step, G4bool& ) {
  if ( track->GetParticleDefinition() == G4Proton::Definition() ) {
    return fProtonInelasticProcess->PostStepDoIt( *track, *step );
  }
}
```