



Geant4: A simulation toolkit

O. St zowski



With many thanks to the Geant4 community !!!!

The roadmap of the lecture

W1: installation / running a G4 application

Why?

W2: Primary generator, GPS, physics list

What is It?

W3: Geometries !

W4: Sensitive detectors / user's actions

NOW, HOW does it really work ?



The roadmap of the lecture

W1: installation / running a G4 application
Why?

W2: Primary generator, GPS, physics list

W3: Geometries !
What is It ?

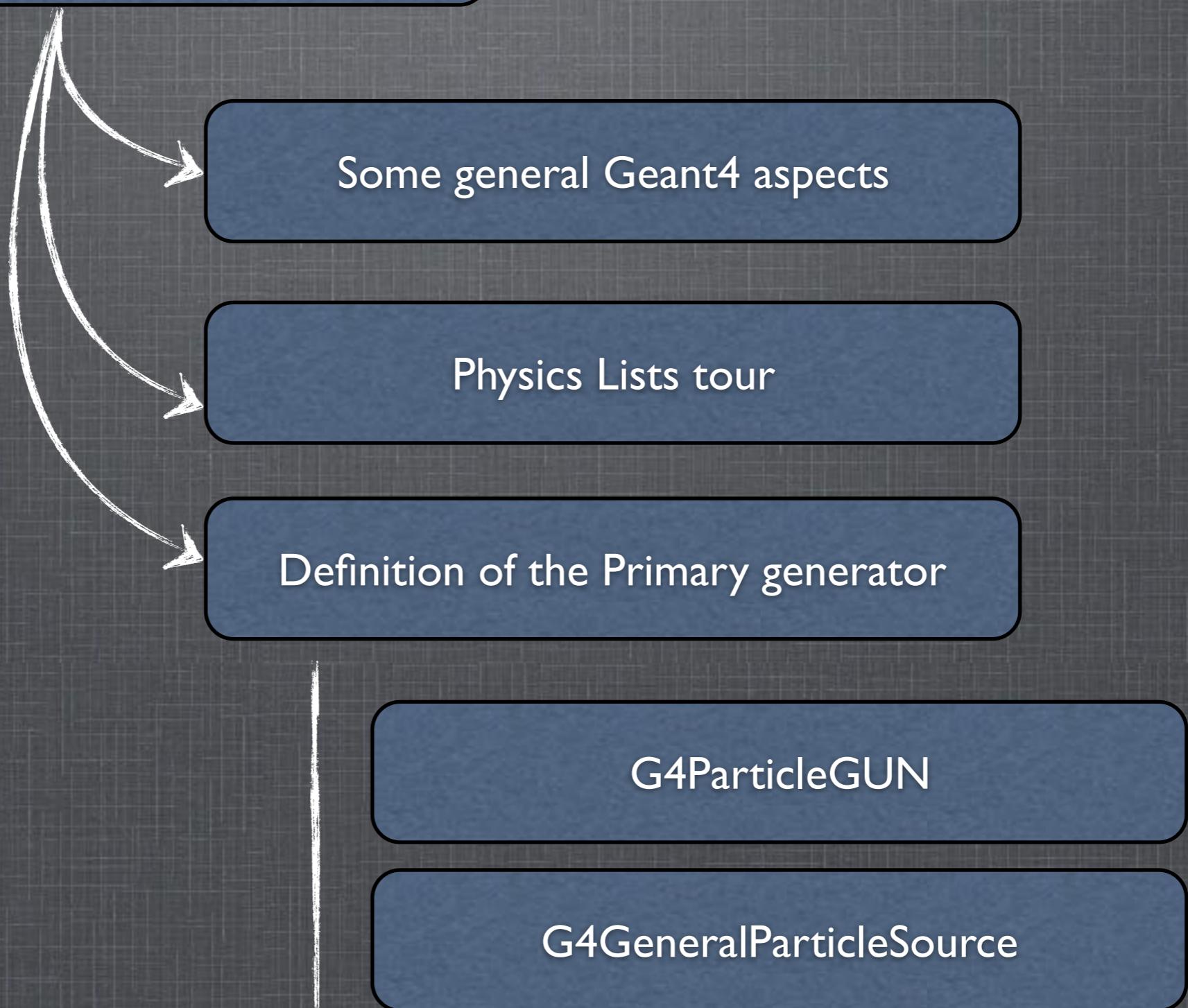
W4: Sensitive detectors / user's actions

More technical view of the content of an application

NOW, HOW does it really work ?



W2: Primary generator, GPS, physics list





System of units in Geant4

Geant4 has no default unit imposed to the user but all available

- To give a number, unit must be “multiplied” to the number. For example :

```
G4double width = 12.5*m;  
G4double density = 2.7*g/cm3;
```



- If no unit is specified, the **internal** G4 unit will be used, but **this is discouraged !**
- Almost all commonly used units are available.
- See: [G4SystemOfUnits.hh](#)

- Divide a variable by a unit you want to get.

```
G4cout << dE / MeV << "(MeV)" << G4endl;
```

- To print the list of units:

- From the code

```
G4UnitDefinition::PrintUnitsTable();
```

- At run-time, as UI command:

```
Idle> /units/list
```



System of units in Geant4

- G4 internal system of units :

millimetre (mm), nanosecond (ns), Mega eV (MeV), positron charge (eplus) degree Kelvin (kelvin), the amount of substance (mole), luminous intensity (candela), radian (radian), steradian (steradian)

- All other units are computed from the basic ones.
- In output, Geant4 can choose the most appropriate unit to use.
Just specify the **category** for the data (Length, Time, Energy, etc...):

```
G4cout << G4BestUnit(StepSize, "Length");
```

→ StepSize will be printed in km, m, mm or ... fermi, depending on its value

- New units can be defined using **G4UnitDefinition**



How to write your command

We have seen one can interact with the simulation using UI commands
The technic in play is available for users using **Messengers**

In fact, it helps to modify selected parameters of a given C++ class:

- ➔ No need to re-compile
- ➔ Convenient, to chain simulations in one session

```
# run 1
/myparameter/set value1
/run/beamOn 1000
# run 2
/myparamteter/set value2
/run/beamOn 1000
...
```

- ➔ **It has a cost** ! some C++ code to be written by the user
- ➔ **Do it if the parameter is modified often in your simulation**

Ex: we would like to change MyParameter in the class MyGenerator



How to write your command

```
class MyGenerator: public G4VUserPrimaryGeneratorAction  
{  
private:  
    G4int MyParameter;  
  
...  
public:  
    void SetMyParameter(G4int new_parameter)  
    {  
        MyParameter = new_parameter;  
    }  
...  
};
```

```
MyGenerator::MyGenerator()  
{  
    MyParameter = 0;  
}  
MyGenerator::MyGenerator()  
{  
}
```

This is the starting point to have a
MyGenerator class
with a parameter called
MyParameter ...



How to write your command

... and the requirements for a messenger:
/MyCommands/MyGenerator/MyParameter 10

```
class MyGeneratorMessenger;

class MyGenerator: public G4VUserPrimaryGeneratorAction
{
private:
    G4int MyParameter;
    MyGeneratorMessenger *theMessenger;
...
public:
    void SetMyParameter(G4int new_parameter)
    {
        MyParameter = new_parameter;
    }
...
};
```

```
MyGenerator::MyGenerator()
{
    MyParameter = 0;
    theMessenger = new MyGeneratorMessenger(this);
}
MyGenerator::MyGenerator()
{
    delete theMessenger;
}
```

```
#include "G4UImessenger.hh"

class G4UIDirectory;
class G4UIcmdWithAString;
class G4UIcmdWithAnInteger;

//! Messenger class for MyGenerator
class MyGeneratorMessenger: public G4UImessenger
{
public:
    MyGeneratorMessenger(MyGenerator *);
    ~MyGeneratorMessenger();

    void SetNewValue(G4UIcommand*, G4String);

private:
    MyGenerator *theGenerator;
    G4UIDirectory *theDirectory;
    G4UIcmdWithAnInteger *Cmd;
};
```

```
MyGeneratorMessenger::MyGeneratorMessenger(MyGenerator *agene):
theGenerator(agene)
{
    theDirectory = new G4UIDirectory("/MyCommands/MyGenerator");
    theDirectory->SetGuidance("List of all my commands");

    Cmd = new G4UIcmdWithAnInteger("/MyCommands/MyGenerator/MyParameter", this);
    Cmd->SetGuidance("To change the value of MyParameter");
    Cmd->SetGuidance("Required parameters: an integer");
    Cmd->AvailableForStates(G4State_PreInit,G4State_Idle);
}

MyGeneratorMessenger::~ MyGeneratorMessenger()
{
    delete theDirectory; delete Cmd;
}

void MyGeneratorMessenger::SetNewValue(G4UIcommand* command, G4String newValue)
{
    if( command == Cmd )
        theGenerator->SetMyParameter( Cmd->GetNewIntValue(newValue) );
}
```



The user's application

This is what we are going to see in the next slides
the primary generator - the description of the physics

```
class AGammaGun : public G4VUserPrimaryGeneratorAction
{
    // the virtual method to be implemented by the user
    virtual void GeneratePrimaries(G4Event* anEvent);
};

class AnElectroMagneticPhysicsList: public G4VUserPhysicsList
{
    // the virtual method to be implemented by the user
    void ConstructParticle();
    // the virtual method to be implemented by the user
    void ConstructProcess();
    // the virtual method to be implemented by the user
    void SetCuts();
};

// The User's main program to control / run simulations
int main(int argc, char** argv)
{
    // Construct the run manager, necessary for G4 kernel to control everything
    G4RunManager *theRunManager = new G4RunManager();

    // Then add mandatory initialization G4 classes provided by the USER
    // detector construction
    // physics list
    // initialisation of the generator

    theRunManager->SetUserInitialization( new ARedSphereConstruction() );
    theRunManager->SetUserAction( new AGammaGun() );

    ...

    return 0;
}
```



The user's application

This is what we are going to see in the next slides
the primary generator - the description of the physics

```
class AGammaGun : public G4VUserPrimaryGeneratorAction
{
    // the virtual method to be implemented by the user
    virtual void GeneratePrimaries(G4Event* anEvent);
};

class AnElectroMagneticPhysicsList: public G4VUserPhysicsList
{
    // the virtual method to be implemented by the user
    void ConstructParticle();
    // the virtual method to be implemented by the user
    void ConstructProcess();
    // the virtual method to be implemented by the user
    void SetCuts();
};

// The User's main program to control / run simulations
int main(int argc, char** argv)
{
    // Construct the run manager, necessary for G4 kernel to control everything
    G4RunManager *theRunManager = new G4RunManager();

    // Then add mandatory initialization G4 classes provided by the USER
    // detector construction
    // physics list
    // initialisation of the generator
    theRunManager->SetUserInitialization( new AnElectroMagneticPhysicsList() );

    ...

    return 0;
}
```



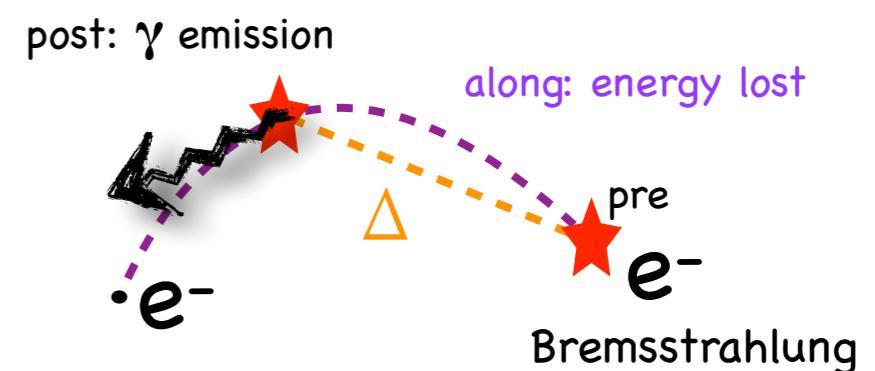
The Physics Lists

The user's class defines **all the particles** and **all the physics** they can see

It includes **Primaries** and **Secondaries**

To each kind of particle a list of processes is attached

- **atomistic approach**: not all the physics into the box!
- models in physics are always evolving
- medical physics not the same than HEP physics !
- for a given interaction, **it allows several models for different energy ranges**
- computation speed is an issue



Process description in Geant4 based on Along, Post and At Rest actions.

Some process belongs to one category while others belong to more. Ex:

- Discrete process: **Compton Scattering, hadronic inelastic, ...**
 - step determined by cross section, interaction at end of step
- Continuous process: **Cerenkov effect**
 - photons created along step, roughly proportional to step length
- At rest process: **radioactive decay of nuclei**
 - interaction at rest

these processes belong to one category

- Rest + discrete: **positron annihilation, decay, ...**
 - both in flight and at rest
- Continuous + discrete: **ionization**
 - energy loss is continuous
 - knock-on electrons (δ -ray) are discrete

these processes belong to two categories



Overview of processes provided

- EM physics
- “standard” processes valid from $\sim 1 \text{ keV}$ to $\sim \text{PeV}$
- “low energy” valid from 250 eV to $\sim \text{PeV}$
- optical photons 
- Weak interaction physics
- decay of subatomic particles
- radioactive decay of nuclei
- Hadronic physics
- pure strong interaction physics valid from 0 to $\sim \text{TeV}$
- electro- and gamma-nuclear valid from 10 MeV to $\sim \text{TeV}$
- Parameterized or “fast simulation” physics

What is to be used for your physics is out of this lecture!



Overview of processes provided

- EM physics
- “standard” processes
- “low energy” valid from ...
- optical photons
- Weak interaction physics
- decay of subatomic particles
- radioactive decay of nuclei
- Hadronic physics
- pure strong interaction physics
- electro- and gamma-physics
- Parameterized or “fitted”

Two possible approaches

- The **G4VUserPhysicsList** class
 - historically, available from the beginning
 - to do the definition at a fine level (per particle) ...
could be long, hard to maintain ...
- Using **G4VModularPhysicsList***
 - physics organized in modules (EM, optical physics...)

A convenient way to go !



There are pre-packaged physics lists provided:

- based on modular physics lists
- **This is not the truth !!!**



* G4VModularPhysicsLists itself inherits from G4VUserPhysicsList ...

What is to be used for your physics is out of this lecture!

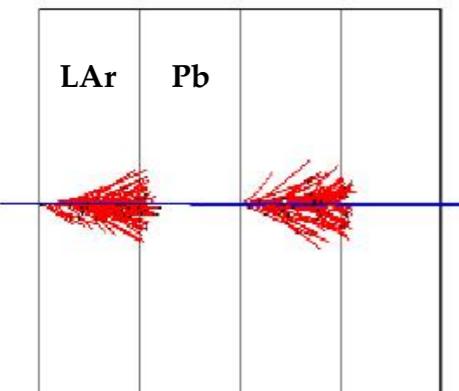


The user's Physics List

```
class AnElectroMagneticPhysicsList: public G4VUserPhysicsList
{
// the virtual method to be implemented by the user
void ConstructParticle();
// the virtual method to be implemented by the user
void ConstructProcess();
// the virtual method to be implemented by the user
void SetCuts();
};
```

```
void AnElectroMagneticPhysicsList::SetCuts()
{
    defaultCutValue = 0.1 * mm;
    // set cut values for gamma at first and for e- second and next for e+,
    // because some processes for e+/e- need cut values for gamma
    //
    SetCutValue(defaultCutValue, "gamma");
    SetCutValue(defaultCutValue, "e-");
    SetCutValue(defaultCutValue, "e+");
    SetCutValue(defaultCutValue, "proton");

    if (verboseLevel>0) DumpCutValuesTable();
}
```



500 MeV proton in LAr/Pb calo.

- 1.5*mm global cut
→ 455 keV in LAr / 2MeV in Pb
- cuts per region is also possible
→ see lecture on geometry

Note - In Geant4, different objects for 'particles'

G4VDynamicParticle: to describe a particle interacting with materials
aggregates information to describe the dynamic of particles
(energy, momentum, polarization, etc...)

G4ParticleDefinition: to define a particle
aggregates information to characterize a particle's properties
(name, mass, spin, etc...) → singleton in C++

```
void AnElectroMagneticPhysicsList::ConstructParticle()
{
    // This ensures that objects of these particle types will be
    // created in the program.
    G4Gamma::GammaDefinition();
    G4Electron::ElectronDefinition();
    G4Positron::PositronDefinition();
    G4MuonPlus::MuonPlusDefinition();
    G4MuonMinus::MuonMinusDefinition();
    ...
}
```

or

```
void AnElectroMagneticPhysicsList::ConstructParticle()
{
    // construction per type of particles
    G4BosonConstructor pBosonConstructor;
    pBosonConstructor.ConstructParticle();

    G4LeptonConstructor pLeptonConstructor;
    pLeptonConstructor.ConstructParticle();

    G4MesonConstructor pMesonConstructor;
    pMesonConstructor.ConstructParticle();
}
```



The user's Physics List

```
void AnElectroMagneticPhysicsList::ConstructProcess()
{
    // ##### TRANSPORTATION #####
    AddTransportation();

    // ##### ELECTROMAGNETIC PROCESSES #####
    G4PhysicsListHelper *ph =
    G4PhysicsListHelper::GetPhysicsListHelper();
    theParticleIterator->reset();
    while( (*theParticleIterator)() ){
        G4ParticleDefinition* particle = theParticleIterator->value();
        G4String particleName = particle->GetParticleName();

        // define EM physics for gamma
        if (particleName == "gamma") {
            // activate photoelectric effect
            ph->RegisterProcess(new G4PhotoElectricEffect, particle);
            // activate Compton scattering
            G4ComptonScattering* cs = new G4ComptonScattering;
            cs->SetEmModel(new G4KleinNishinaModel());
            ph->RegisterProcess(cs, particle);
            // activate gamma conversion
            ph->RegisterProcess(new G4GammaConversion, particle);
        }
        ...
    }
}
```



Or

```
AModularElectroMagneticPhysicsList::AModularElectroMagneticPhysicsList()
: G4VModularPhysicsList()
{
    // default cut value (1.0mm)
    defaultCutValue = 1.0*mm;
    RegisterPhysics( new G4EmStandardPhysics(1) );
    // Muon Physics ( Apply related processes to mu and tau
    RegisterPhysics( new MyNeutronPhysics("neutron") );
}

void AModularElectroMagneticPhysicsList::SetCuts()
{
    SetCutsWithDefault();
}
```



```
class MyNeutronPhysics : public G4VPhysicsConstructor
{
    MyNeutronPhysics(const G4String& name="neutron");
    // This method will be invoked in the Construct() method.
    virtual void ConstructProcess();
};
```

here ConstructProcess() contains something similar to the left window !!!!

```
// The User's main program to control / run simulations
int main(int argc, char** argv)
{
    // Construct the run manager, necessary for G4 kernel to control everything
    G4RunManager *theRunManager = new G4RunManager();
    // Then add mandatory initialization G4 classes provided by the USER
    // First method
    theRunManager->SetUserInitialization(new AnElectroMagneticPhysicsList());
    // OR second method
    theRunManager->SetUserInitialization(new AModularElectroMagneticPhysicsList());
    // OR directly a pre-packaged physics list ...
    G4VModularPhysicsList * aphysicsList = new FTFP_BERT();
    theRunManager->SetUserInitialization( aphysicsList );
}
```

Whatever the method ➔

Many pre-packaged **PhysicsLists** available:

http://geant4.cern.ch/support/proc_mod_catalog/physics_lists/physicsLists.shtml



The user's application

This is what we are going to see in the next slides
the primary generator - the description of the physics

```
class AnElectroMagneticPhysicsList: public G4VUserPhysicsList
{
// the virtual method to be implemented by the user
    void ConstructParticle();
// the virtual method to be implemented by the user
    void ConstructProcess();
// the virtual method to be implemented by the user
    void SetCuts();
};

class AGammaGun : public G4VUserPrimaryGeneratorAction
{
// the virtual method to be implemented by the user
    virtual void GeneratePrimaries(G4Event* anEvent);
};

// The User's main program to control / run simulations
int main(int argc, char** argv)
{
// Construct the run manager, necessary for G4 kernel to control everything
    G4RunManager *theRunManager = new G4RunManager();

// Then add mandatory initialization G4 classes provided by the USER
// detector construction
// physics list
// initialisation of the generator
    theRunManager->SetUserAction( new AGammaGun() );

    ...
}

return 0;
```



G4 Primary Generators

- To **provide primary events to the simulation**, the user should provide a class which inherits from `G4VUserPrimaryGeneratorAction`
- This class **controls the generation of primaries**
What kind of particle (how many) what energy, position, direction, polarisation, etc
- This class itself **should NOT generate primaries but invoke `GeneratePrimaryVertex()` method of primary generator(s)** to make primaries (`G4VPrimaryGenerator`)
- Thus, in principle, the constructor is used to
 - Instantiate primary generator(s)
 - Set default values to it (them)
- In the `GeneratePrimaries()` method
 - Randomize particle-by-particle value(s)
 - Set these values to primary generator(s)
 - Invoke `GeneratePrimaryVertex()` method of primary generator(s)



The G4ParticleGun

- **G4ParticleGun** is one Concrete implementations of **G4VPrimaryGenerator**
A good example for experiment-specific primary generator implementation
- It shoots
one primary particle of a **certain energy** from a **certain point** at a **certain time**
to a **certain direction**.
 - Various set methods are available
 - Intercoms commands* are also available for setting initial values
- **G4ParticleGun** is **basic**, but it can be used from inside **UserPrimaryGeneratorAction**
to model **complex source types / distributions**:
 - Generate the desired distributions (by shooting random numbers)
 - Use set methods of **G4ParticleGun**
 - Use **G4ParticleGun** as many times as you want
 - Use other primary generators as many times as you want to make overlapping events



The G4ParticleGun

Example of usage of **G4ParticleGun**

In **MyPrimaryGeneratorAction.cc**

```
MyPrimaryGeneratorAction::MyPrimaryGeneratorAction()
{
    particleGun = new G4ParticleGun();
}

void MyPrimaryGeneratorAction::GeneratePrimaries(G4Event* anEvent)
{
    particleGun->SetParticleDefinition(G4Electron::Definition());
    particleGun->SetParticleMomentum(G4ThreeVector(1.0,0,0));
    particleGun->SetParticleEnergy(100.0*keV);
    particleGun->GeneratePrimaryVertex(anEvent);
}
```

You can repeat this for generating more than one primary particles.
→ several **G4ParticleGun** in action



The G4GeneralParticleSource GPS*

GPS is an advanced concrete implementation of G4VPrimaryGenerator

It offers as pre-defined many common (and not so common) options for particle generation

- Primary vertex can be **randomly positioned** with options
Point, Beam, Plane (Circle, Annulus, Ellipsoid, Square or Rectangle)
Surface or Volume (Sphere, Ellipsoid, Cylinder or Para)
- **Angular emission** can be
isotropic (iso), cosine-law (cos), planar wave (planar), 1-d accelerator beam (beam1d),
2-d accelerator beam (beam2d), focusing to a point (focused) or user-defined (user)
- **Kinetic energy** of the primary particle can also be randomized.
mono-energetic (Mono), linear (Lin), power-law (Pow), exponential (Exp), Gaussian (Gauss),
bremsstrahlung (Brem), black-body (Bbody), cosmic diffuse gamma ray (Cdg), user-defined histogram (User),
arbitrary point-wise spectrum (Arb) and user-defined energy per nucleon histogram (Epn)
- **Multiple sources**
With user defined relative intensity
- **Capability of event biasing (variance reduction).**
By enhancing particle type, distribution of vertex point, energy and/or direction
- All features can be used via C++ or **command line (or macro) UI**

* <http://geant4-userdoc.web.cern.ch/geant4-userdoc/UsersGuides/ForApplicationDeveloper/html/GettingStarted/generalParticleSource.html>



The G4GeneralParticleSource GPS

On the user's side, the **G4VUserPrimaryGeneratorAction** is very simple to implement

In **MyGPSPrimaryGeneratorAction.cc**

```
MyGPSPrimaryGeneratorAction::MyGPSPrimaryGeneratorAction()
{
    m_particleGun = new G4GeneralParticleSource();
}

MyGPSPrimaryGeneratorAction::~MyGPSPrimaryGeneratorAction()
{
    delete m_particleGun;
}

void MyGPSPrimaryGeneratorAction::GeneratePrimaries(G4Event* anEvent)
{
    m_particleGun->GeneratePrimaryVertex(anEvent);
}
```

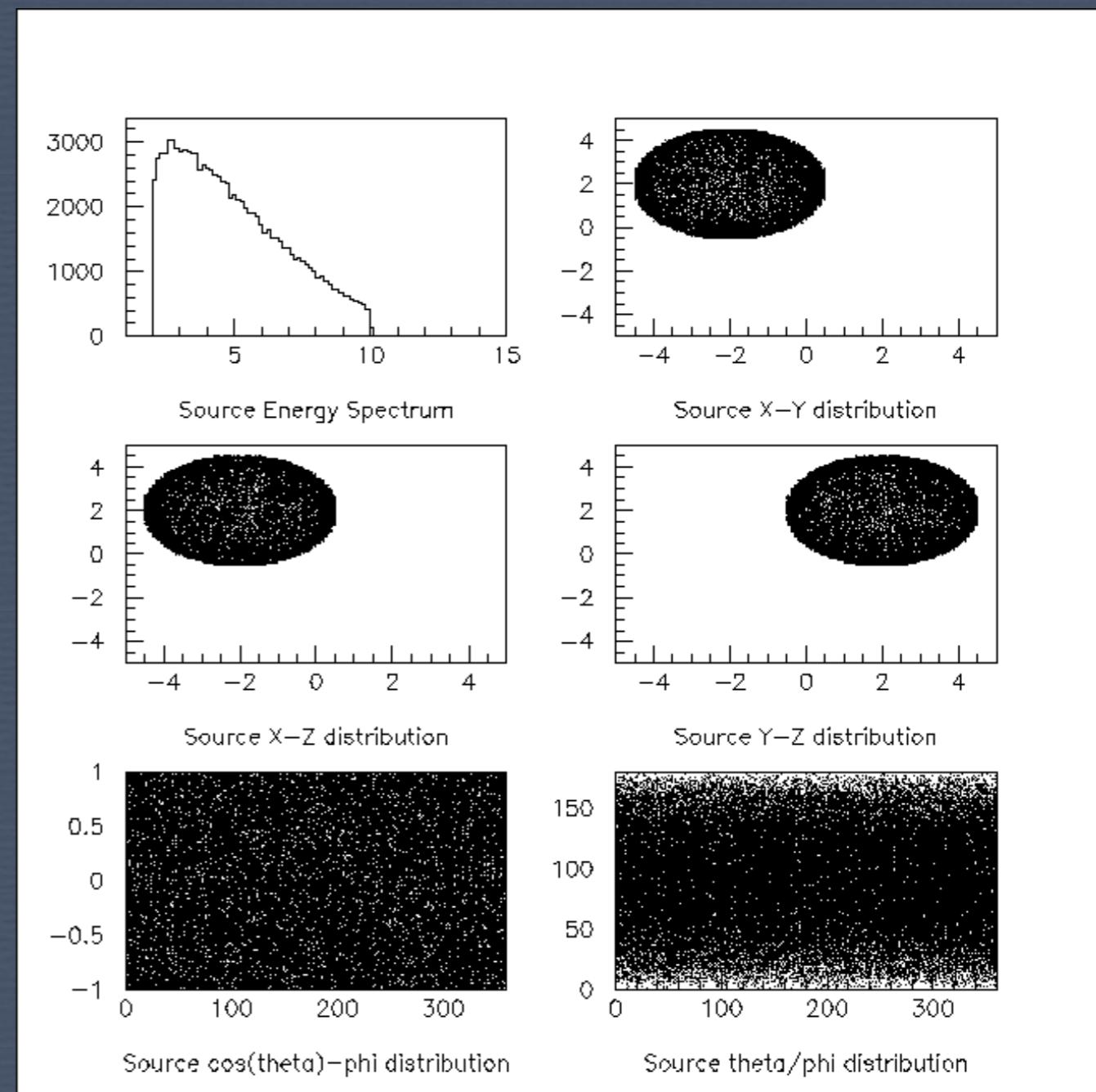


The G4GeneralParticleSource GPS

- Vertex on sphere surface
- Isotropic emission
- Pre-defined spectrum (black-body)

G4 macro to do that

```
/gps/particle geantino  
/gps/pos/type Surface  
/gps/pos/shape Sphere  
/gps/pos/centre -2. 2. 2. cm  
/gps/pos/radius 2.5 cm  
/gps/ang/type iso  
/gps/ene/type Bbody  
/gps/ene/min 2. MeV  
/gps/ene/max 10. MeV  
/gps/ene/temp 2e10  
/gps/ene/calculate
```





The G4GeneralParticleSource GPS

G4 macro to do that

```
# beam #1  
# default intensity is 1 now change to 5.  
/gps/source/intensity 5.
```

```
/gps/particle proton  
/gps/pos/type Beam
```

```
# the incident surface is in the y-z plane  
/gps/pos/rot1 0 1 0  
/gps/pos/rot2 0 0 1
```

```
# the beam spot is centered at the origin and is  
# of 1d gaussian shape with a 1 mm central plateau  
/gps/pos/shape Circle  
/gps/pos/centre 0. 0. 0. mm
```

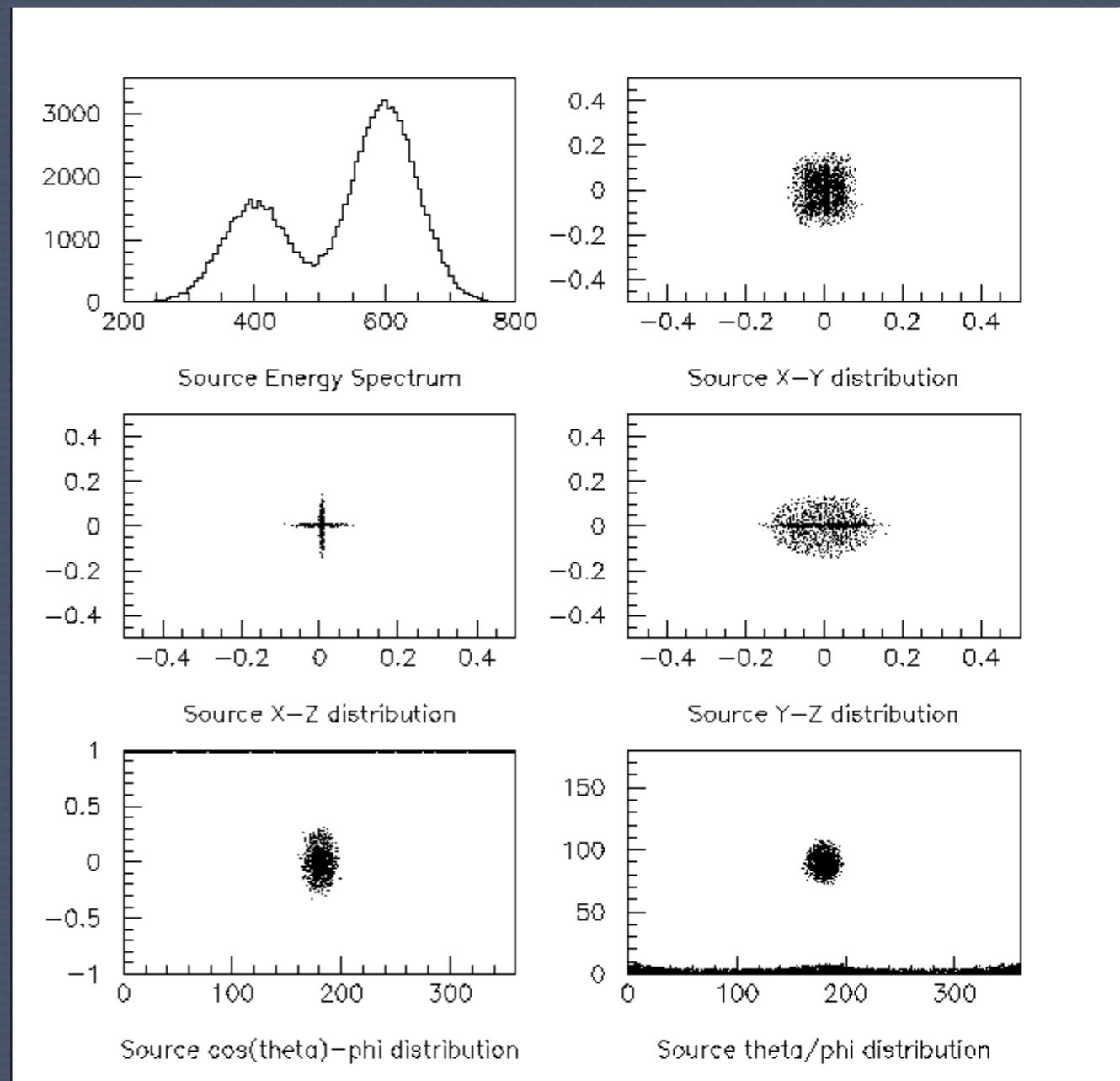
```
/gps/pos/radius 1. mm  
/gps/pos/sigma_r .2 mm  
#
```

```
# the beam is travelling along the X_axis with 5 degrees dispersion  
/gps/ang/rot1 0 0 1  
/gps/ang/rot2 0 1 0  
/gps/ang/type beam1d  
/gps/ang/sigma_r 5. deg  
#
```

```
# the beam energy is in gaussian profile centered at 400 MeV  
/gps/ene/type Gauss  
/gps/ene/mono 400 MeV  
/gps/ene/sigma 50. MeV
```

```
# beam #2  
# 2x the intensity of beam #1  
/gps/source/add 10.  
#  
# this is a electron beam
```

- Two-beam source definition (multiple sources)
- Gaussian profile
- Can be focused / defocused





Conclusions of W2

We have seen:

- how to write messengers
- an overview of the G4 physics list
 - customize if it is required
 - **use pre-defined ones**
- how to write different **generators of primaries**
 - there are also 2 primary generators dedicated to HEP
 - **G4HEPEvtInterface, G4HEPMCInterface**