Geant4 のプログラムをカスタマイズしよう。

/usr/local/Geant4/ の下に今回の講義で書き換えて使えるようのサンプルが、seminar.tar.gz として置いてあるので、自分のディレクトリにコピーをして tar で解凍して下さい。

展開すると seminar ディレクトリが生成されます。その中に今回カスタマイズして使う Geant4 のプログラム一式が入っています。

GNUmakefile: プログラムを make(コンパイル)する手順が機械語で書かれているファイルです。このファイルがあるところで、make をします。

main.cc: メインプログラムです。

proton.mac: Geant4 を起動して実行結果の絵を描くためのコマンドが書かれています。ビームの種類やエネルギー、入射方向などを変更するときはこのファイルを編集します。

run000*.dat: 実行結果が書かれたファイルです。追加上書きされるので、Geant4 を起動する前には一度削除する必要があります。

bin ディレクトリ:プログラムをコンパイルすると生成され、この中に実行ファイルが収められます。

includ ディレクトリ:メインプログラムに関連してインクルードするヘッダーファイルが入っています。

src ディレクトリ: メインプログラムで参照されている他のプログラムが入っています。

プログラムを書き換えるにあたってまずは、既存のプログラムを実行してみます。

```
$ cd seminar/
```

と打って seminar ディレクトリに移動します。

seminar/以下に移動できたらコマンドラインで

```
$ source(or .) /usr/local/Geant4/geant4.9.1.p03/env.sh
```

と打ち、環境変数を設定します。

新しいウィンドウでコンパイルする前にはおまじないと思って必ず実行しましょう。同じウィンドウであれば一度実行すれば毎回する必要はありません。

コンパイルするにはコマンドラインで

```
$ make
```

と打ちます。

すると、コンパイルが始まるので、終わるまでしばし待ちましょう。

コンパイルが終了したら、Geant4 を実行します。
コマンドラインで

```
$ ./bin/Linux-g++/main
```

と打つと、Geant4 が走ります。

コマンドラインで

```
Idle> /control/execute proton.mac
```

と打ちます。
proton.mac もマクロファイル(ROOT を使うときにも出てきましたね)と呼びます。実行したい命令を任意の mac ファイルに書き込んでおけば、そのマクロファイルを起動するとそれらの命令を順に実行してくれます。
/control/execute とはマクロファイルを実行するコマンドです。

すると、もう一つウインドウが開いて、絵が描かれます。
この絵は、外枠の白線が実験室を、中の 5 つの白枠は鉄板を表しています。
青色の線が+の電荷を持つ粒子、赤色の線が−の電荷を持つ粒子、緑色の線が中性の粒子をそれぞれ表しています。

このウインドウを閉じるには、コマンドラインで

```
Idle> exit
```

と打ちます。

マクロファイルを実行しするとコマンドを入力したウインドウにも何やら数字が書かれます。
たとえば、

Run:0started
Event:0
0 proton (0.00113588,-0.000338649,5) 7999.98MeV
1 proton (-0.000589513,-0.00972099,25) 7986.35MeV
2 proton (-0.00704918,-0.0355289,45) 7974.83MeV
3 proton (-0.0259273,-0.0613517,65) 7964.91MeV
4 proton (-0.0530934,-0.0732295,85) 7955.25MeV
1 e- (9.9207151,10.204554,25) 1.323326MeV

これらの数字は
1 列目の 0,1,2…は鉄板の番号を示していて、粒子の入射する方向から順に 0,1,2…となっています。
2 列目の proton, e^-というのは鉄板を通過した粒子の種類を示しています。
このコマンドライン上には中性の粒子は出力されていません。
3 列目の括弧内の数字はそれら粒子の位置 x,y,z(position)です。
4 列目の 7999.98MeV というのは粒子の運動エネルギー(MeV)です。

またこれら出力された数値はファイルにも書き出されています。コマンドラインで

```
$ ls
```

と打ち、seminar/以下にあるファイルを表示して、run0000.dat や run0001.dat というファイルがあることを確認して下さい。

コマンドラインで

```
$ emacs run0000.dat
```

と打って、そのファイルの中身を見ると、以下のように表示されます。

0	proton	1	0	0.0630816	0.0237065	8888.87	0.00113588	−0.000338649	5	8938.25	1	938.272	
1	proton	1	0	−1.70065	−7.5554	8875.16	−0.000589513	−0.00972099	25	8924.62	1	938.272	
2	proton	1	0	−4.07266	−15.3607	8863.57	−0.00704918	−0.0355289	45	8913.1	1	938.272	
3	proton	1	0	−12.6219	−7.54565	8853.59	−0.0259273	−0.0613517	65	8903.18	1	938.272	
4	proton	1	0	−11.4368	−2.94705	8843.89	−0.0530934	−0.0732295	85	8893.53	1	938.272	
1	e^-	4	1	1.0475	0.995078	1.00806	9.92072	10.2046	25	1.83433	−1	0.510999	
0	gamma	10	4	0.099131	−0.0529014	−0.20945	14.9082	8.00913	15	0.237687	0	0	

始めの 2 列はウインドウに出力されていたものと同じで、鉄板の番号と粒子の種類を示しています。
3 列目の数字は鉄板を通過した粒子の ID で、4 列目の数字はその粒子を生成した親粒子の ID を示しています。
0 なのは、元々の粒子だからです。
e^-を例に見ると、1　e^-　4　1 となっています。4 行目の数字が 1 なので、proton から生成されたことが分かります。
そして、3 行目の数字が 4 なので、e^-自身の ID は 4 ということです。
1 行目が 1 なので 1 の鉄板を通ったことが分かります。
gamma は　0　gamma　10　4 となっているので、e^-から生成されて 0 の鉄板を通ったということです。

5,6,7 行目はそれぞれ x,y,z 方向の運動量(MeV/c)を表していて、
8,9,10 行目は x,y,z の位置座標を表しています。
11 行目はその粒子の全エネルギー(MeV)で、12 行目はその粒子の電荷で 0 が中性、1 がプラス、−1 がマイナスを示している。
13 行目は粒子の質量エネルギー(MeV)を表しています。

proton.mac(実行マクロファイル)

コマンドラインで

```
$ emacs proton.mac
```

で proton.mac の中身を見ると、

/vis/open OGLIX
グラフィックを表示するドライバの選択をします。
/vis/drawVolume
実験空間を描画します。
/vis/scene/add/trajectories
/vis/scene/add/hits
生成された粒子の軌跡を表示します。
/vis/viewer/set/viewpointThetaPhi 80 20
サンプルにあったように視点を回転します。
/mydet/generator particleGun
particleGun の場合は次に指定する粒子がビームになります。PYTHIA と指定すると、PYTHIA でシミュレーションした結果を読み込ませます。
/gun/particle proton
入射粒子を指定します。
/gun/energy 8000 MeV
入射粒子の運動エネルギーを決めます。
/gun/position 0 0 -.10 m
入射粒子の入射位置を指定します。
/gun/direction 0 0 1
入射粒子の進行方向を決めます。この場合は z 方向(粒子の進行方向)と並行な成分のみとなります。
/run/beamOn 5
入射させる粒子の数を指定できます。この場合は 5 発です。

以下に、main.cc でインクルードされているプログラムソースの中で、特に書き換えるのに知っているべき内容をあげます。

PrimaryGeneratorAction.cc

```
PrimaryGeneratorAction::PrimaryGeneratorAction()
  : G4VUserPrimaryGeneratorAction(),
    particleGun( 0 )
{
```

```
//const char* filename = "pythia_event.data";
//HEPEvt = new G4HEPEvtInterface(filename);
```

今は//でコメント扱いになっていますが、pythia_event.data を読み込んで絵を描かせることもできます。

```
    G4int n_particle = 1;
    G4ParticleGun* fParticleGun = new G4ParticleGun(n_particle);
    G4ParticleTable* particleTable = G4ParticleTable::GetParticleTable();
    G4String particleName;
    G4ParticleDefinition* particle
       = particleTable->FindParticle(particleName="proton");
```

proton の情報を particleTable から取ります。

```
    fParticleGun->SetParticleDefinition(particle);
```

入射粒子の定義。ここでは、proton。

```
    fParticleGun->SetParticleMomentumDirection(G4ThreeVector(0.,0.,1.));
```

運動量の向き(ベクトル)を指定します。この場合は、z 方向(粒子の進行方向)にのみ運動量を持ちます。

```
    fParticleGun->SetParticleEnergy(8000.*MeV);
```

入射粒子の運動エネルギーの指定。

```
    fParticleGun->SetParticlePosition(G4ThreeVector(0.*cm,0.*cm,0.*cm));
```

入射粒子の入射位置。(0,0,0)は、マザーボリュームのど真ん中。

```
    particleGun = fParticleGun;

    messenger = new PrimaryGeneratorMessenger(this);
    useHEPEvt = true;

}
```

proton.mac で指定しないとここに書かれたことが実行されます。

```
void PrimaryGeneratorAction::GeneratePrimaries( G4Event* anEvent )
{

    if(useHEPEvt)
    {
```

```
      double z;
      z = G4UniformRand()*2.;
```
Geant4 の中で使う乱数。
```
      if(z <0) z = z * (-1.);
        //-- Beam position (no correlation in x/y position assumed)
        G4double vx = (G4UniformRand()*2.)*cm;   // vertex-x position
        G4double vy = (G4UniformRand()*2.)*cm;   // vertex-y position
        G4double vz = z*cm;                      // vertex-z position
//          G4double vz = 0.*cm;                   // vertex-z position
         HEPEvt->SetParticlePosition(G4ThreeVector(vx,vy,vz));
```
ビームの入射位置を乱数で振ることもできます。
```
         HEPEvt->GeneratePrimaryVertex(anEvent);
         fileconst(anEvent->GetEventID());
```
Event ナンバーごとにファイルを作ります。
```
   }
   else
   {
particleGun->GeneratePrimaryVertex(anEvent);
//fileconst(int(anEvent->GetEventID()/1000.));
fileconst(int(anEvent->GetEventID()));
}
}
```

---

fileconst.cc

```
void fileconst(int runnum){

sprintf(filenm,"run%04d.dat",runnum);
```
run****.dat というファイルを Event ナンバーごとに生成します。
```
}
```

PhysicsList.cc

```
void PhysicsList::ConstructEM()
```

```
{
   theParticleIterator-> reset();
   while( (*theParticleIterator)() ){
      G4ParticleDefinition* particle = theParticleIterator->value();
      G4ProcessManager* pmanager = particle->GetProcessManager();
      G4String particleName = particle->GetParticleName();
```

粒子の名前を取ります。

```
      if (particleName == "gamma") {
         // gamma
         pmanager->AddDiscreteProcess(new G4GammaConversion());
         pmanager->AddDiscreteProcess(new G4ComptonScattering());
         pmanager->AddDiscreteProcess(new G4PhotoElectricEffect());

      } else if (particleName == "e-") {
         //electron
         G4VProcess* theeminusMultipleScattering = new G4MultipleScattering();
         G4VProcess* theeminusIonisation         = new G4eIonisation();
         G4VProcess* theeminusBremsstrahlung     = new G4eBremsstrahlung();
         //
         // add processes
         pmanager->AddProcess(theeminusMultipleScattering);
         pmanager->AddProcess(theeminusIonisation);
         pmanager->AddProcess(theeminusBremsstrahlung);
```

・・・・

粒子反応を記述したソースは大抵、/usr/local/Geant4/geant4.9.1.p03/source/processes/の下にあります。

DetectorConstruction.cc

```
   G4Material* FilmBaseTAC      = G4Material::GetMaterial("FilmBaseTAC");
   G4Material* EmulsionOpera    = G4Material::GetMaterial("EmulsionOpera");
   G4Material* Acrylic          = G4Material::GetMaterial("Acrylic");
```

G4Material PbPL = G4Material::GetMaterial("PbPL");*
G4Material Air = G4Material::GetMaterial("Air");*
G4Material FePL = G4Material::GetMaterial("FePL");*

検出器や標的に使われている物質を指定します。物質そのものの定義は、MaterialConstruction.cc の中でするので、その項を参照してください。
GetMaterial()のカッコの中の文字は MaterialConstruction.cc の中で定義した名前になります。

*G4double feDX = 200.0 *mm; // X-length of the film*
*G4double feDY = 200.0 *mm; // Y-length of the film*

ここでは、検出器の縦横の大きさを feDX と feDY という変数名で与えています。

solFilmElement[0] = new G4Box("FePlate", feDX/2.0, feDY/2.0, 10./2.0);

検出器を形成する個々の物質でできた立体の大きさを指定します。赤字の部分は立体の厚みを入れる場所です。**検出器や標的の厚みを変える時はここを書き換えます。**単位は mm です。
たとえば原子核乾板は、  絵のようにアクリルベース(白)の両端に乳剤(黄色)がある２種３層構造をしています。この場合は、乳剤層を示すボックス(立体)とベースを示すボックスをそれぞれ定義して、あとから組み合わせます。

~~~~~~~~~~~~~~~~Advanced definition!!!!!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

検出器・標的を球状に定義する。

G4Sphere は、空洞の球体を定義できるクラスで、角度を指定することでかまくらのような形を作ることもできます。

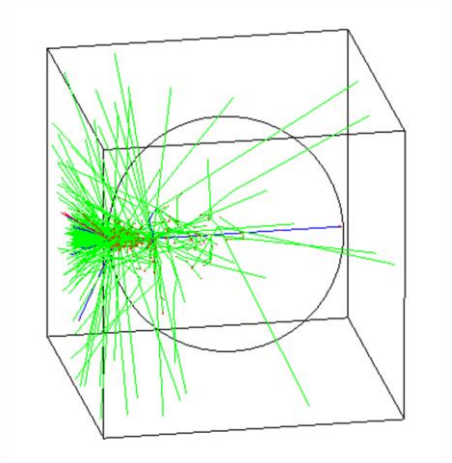
*G4double feRmin = .0 *mm;*
球の内側の半径。
*G4double feRmax = 100.0 *mm; // R-length of the PbBall*
球の外側の半径。
*G4double feSPhi = .0 *rad;*
x 軸と球の直径のなす角。
*G4double feDPhi = 2*acos(-1) *rad; // Phi*
xy 平面上の中心角。
*G4double feSTheta = .0 *rad;*
z 軸と球の直径のなす角。
~~~~~~~~~~~~~~~~

```
G4double feDTheta = acos(-1) *rad;     // Theta
```
xz 平面上の中心角。

```
solElement[0] = new G4Sphere("PbBall", feRmin, feRmax,
                                        feSPhi, feDPhi,
                                        feSTheta, feDTheta);

logElement[0] = new G4LogicalVolume(
                                    solElement[0],        // Solid
                                    PbPL,                 // Material
                                    "PbBall");            // Name

   G4PVPlacement * Pbplate[1];
  G4ThreeVector Pb_POS(0., 0., 0.);
   Pbplate[0] = new G4PVPlacement(0,
                          Pb_POS,
                          "Pb00",
                          logElement[0],
                          worldPV,
                          false,
                          1);
```



このように定義すると、図のような標的が描けます。

~~~~~~

```
    logFilmElement[0] = new G4LogicalVolume(
```
~~~~~~

```
                                solFilmElement[0],      // Solid
                                FePL,                   // Material
                                "FePlate");                 // Name
```

定義したボックスがそれぞれ、どのような物質で形成されているかを指定してロジカルボリュームを定義します。緑字の部分に、上で定義したボックスの情報が収められています。赤字の部分には MaterialConstruction.cc の中で定義した物質の名前を入れます。

```
  const G4ThreeVector WORLD_SIZE(.26*m, .26*m, .26*m);
  const G4ThreeVector WORLD_POS(0., 0., 0.);

  G4Box* worldSolid =
    new G4Box("WORLD",
              WORLD_SIZE.x()/2.,
              WORLD_SIZE.y()/2.,
              WORLD_SIZE.z()/2.);
```

実験空間の大きさを定義します。

```
  G4LogicalVolume* worldLV =
    new G4LogicalVolume(worldSolid,
                        Air,
                        "WORLD_LV");
```

実験空間の大きさと物質(空気)を与えてロジカルボリュームとして定義します。

```
  G4PVPlacement* worldPV =
    new G4PVPlacement(0,            // 回転しない
                      WORLD_POS,    // 配置する座標
                      "WORLD_PV", // 名前
                      worldLV,      // 配置するロジカルボリューム
                      0,            // 配置するフィジカルボリューム(親)を指定
                      false,
                      0);           // コピーナンバー

  G4PVPlacement* Feplate[10];
```

検出器を設置します。(検出器の情報を格納しておく場所を作ります。)

```
  char Fename[10];
```

```
   for(int i = 0; i < 5; i++){
```
検出器・標的の構造を定義します。(繰り返し構造の場合は、このようにループを使います。)
```
        sprintf(Fename,"Fe%03d",i);
                G4ThreeVector Fe_POS(0., 0., (10+(double(i)*20))*mm);
```
検出器・標的の位置を指定します。マザーボリューム(0,0,0)の位置からの座標を指定します。((0,0,0)は、空間のど真ん中にあたります。)
```
                Feplate[i] = new G4PVPlacement(0,
                        Fe_POS,     // 配置する座標
                        Fename,   // 名前
                        logFilmElement[0],        // 配置するロジカルボリューム
```
物質と大きさを与えたロジカルボリュームを与えます。
```
                        worldPV,      // worldPV に配置する
                        false,
                        i);
```
配置する検出器・標的の番号(ID)になります。
```
   }

   BodySD* detectorSD = new BodySD();
   G4SDManager::GetSDMpointer()->AddNewDetector(detectorSD);
   logFilmElement[0]->SetSensitiveDetector(detectorSD);
```
ロジカルボリュームを定義した物体を情報を取り出せるセンシィティブ(感度のある)検出器に定義します。どのような情報を取り出すかは、BodySD.cc を参照してください。

```
   G4FieldManager* fieldManager =
   G4TransportationManager::GetTransportationManager()->GetFieldManager();
   MagneticField* magField = new MagneticField();
   fieldManager->SetDetectorField(magField);
   fieldManager->CreateChordFinder(magField);
```
検出器・標的に磁場をかけることができます。どの向きのどれくらいの大きさの磁場をかけるかは、MagneticField.cc で定義します。

```
   return worldPV;
}
```

MaterialConstruction.cc

```
G4Element* elPb = new G4Element(name="Lead"      , symbol="Pb", z=
82.,a=207.19*g/mole);
G4Element* elFe = new G4Element(name="Iron"      , symbol="Fe", z=
26.,a=55.85*g/mole);
```

個々の原子を定義する場合は、G4Element を使います。この name で定義した名前が DetectorConstruction.cc のロジカルボリュームの定義で使用する物質の名前になります。

```
G4Material* Al
  = new G4Material(name          = "Aluminium",
                   z             = 13.,
                   a             = 26.98*g/mole,
                   density       = 2.70*g/cm3);
Al->SetChemicalFormula("Al");
```

単体の物質を定義する場合は、G4Material を使います。G4Element で原子を定義してから単体物質を定義しても構いませんん。

```
//--- Water
  G4Material* H2O
    = new G4Material(name          = "Water",
                     density       = 1.000*g/cm3,
                     ncomponents = 2);
  H2O->AddElement(elH, natoms=2);
  H2O->AddElement(elO, natoms=1);
```

1 分子中の原子数

たとえば、物質として水を定義する場合、G4Element で水素(elH)と酸素(elO)を定義しておきます。それから G4Material で物質を構成する成分の数(ncomponents)と、それら成分比(natoms)を定義して水を構成します。

```
//--- FujiFilm ET-7C/7D emulsion for KEK-PS E373
G4Material* EmulsionE373
  = new G4Material(name          = "EmulsionE373",
                   density       = 3.60*g/cm3,
                   ncomponents = 8);
EmulsionE373->AddElement(elI, fractionmass=0.003);
```

```
EmulsionE373->AddElement(elAg,fractionmass=0.454);
EmulsionE373->AddElement(elBr,fractionmass=0.334);
EmulsionE373->AddElement(elS, fractionmass=0.002);
EmulsionE373->AddElement(elO, fractionmass=0.068);
EmulsionE373->AddElement(elN, fractionmass=0.031);
EmulsionE373->AddElement(elC, fractionmass=0.093);
EmulsionE373->AddElement(elH, fractionmass=0.015);
```

原子核乾板の乳剤を定義する場合は、それぞれの原子を重量比(fractionmass)で定義して構成します。

BodySD.cc

```
std::ofstream f(filenm,std::ios::out | std::ios::app);
```

一粒子ずつシミュレートされていくので一つの Event で生成されたすべての粒子が同じファイルに書き込まれるように追加上書きでファイルを開きます。

```
const G4StepPoint* preStepPoint = aStep->GetPreStepPoint();
const G4StepStatus status = preStepPoint->GetStepStatus();
G4ParticleDefinition* pd = aStep->GetTrack()->GetDefinition();

G4ThreeVector pos = preStepPoint->GetPosition();
double posx = pos.x();
double posy = pos.y();
double posz = pos.z();
```

GetPosition()が返すのはベクトル表示(1.0,2.0,3.0)になった位置座標です。そこから、x 座標のみ、y 座標のみ、z 座標のみを取り出して、それぞれ posx、posy、posz に入れます。

```
G4ThreeVector mom = preStepPoint->GetMomentum();
double momx = mom.x();
double momy = mom.y();
double momz = mom.z();

G4double kE = preStepPoint->GetKineticEnergy();
G4double tE = preStepPoint->GetTotalEnergy();
```

```
G4double mss = preStepPoint->GetMass();
G4Track* track = aStep->GetTrack();
G4String particlename = pd->GetParticleName();
```
/usr/local/Geant4/geant4.9.1.p03/source/track/の下に粒子の情報を呼び出す関数が多くあります。その include の中にある G4Track.icc, G4StepPoint.icc, G4Step.icc などが参考になります。

```
   if ( status == fGeomBoundary && charge != 0.) {
      G4cout << channel << " " << particlename << " " << pos << " " << kE/MeV << "MeV" << G4endl;   }
```
ボリューム(立体物質)の境界の時に、電荷を持つ粒子の情報だけを画面出力します。

```
    if ( status == fGeomBoundary) {
       f << setw(4) << channel << " " << setw(8) << particlename << " " << setw(4) << tid << " " << setw(4) << paid << " "<< setw(8) << momx << " " << setw(8) << momy <<" " << setw(8) << momz << " "<< setw(8) << posx << " " << setw(8) << posy << " " << setw(8) << posz << " " << setw(8) << tE/MeV << " " << setw(2) << charge << " " << setw(6) << mss << std::endl;
            }
```
ファイルには、すべての粒子の情報を書き出します。

```
f.close();
   return true;
}
```

MagneticField.cc

```
void MagneticField::GetFieldValue( const G4double Point[3], G4double* Bfield ) const
{
```
Point[0],Point[1],Point[2]はそれぞれ x,y,z 座標です。

```
   const G4double zup = .35*m;
   const G4double zlow = -.15*m;
```
磁場をかける最上流と最下流の z 座標(ビーム方向)を指定します。

```
   const G4double By = 1.0*tesla;
```

y 方向に 1 テスラの磁場をかけます。

```
   if ( Point[2] <= zup && Point[2] >= zlow ) {
```

z 座標が指定した最上流・最下流の間にあるときだけ、磁場をかけます。

```
      Bfield[0]= 0.0;
      Bfield[1]= By;
      Bfield[2]= 0.0;
   } else {
      Bfield[0]= 0.0;
      Bfield[1]= 0.0;
      Bfield[2]= 0.0;
   }
return;
}
```

以上が、Geant4 のプログラムを書き換えるのに最低限知っていると良い知識です。
自由に書き換えて粒子の振る舞いをシミュレートしてみてください。