```
#include <iostream>
#include <string>
#include <cassert>
#include <stdexcept>

#include "TMath.h"
#include "TGeoManager.h"
#include "TGeoVolume.h"
#include "TGeoBBox.h"
#include "TGeoMatrix.h"
#include "TH1.h"
#include "TCanvas.h"
#include "TRandom.h"
#include "TRint.h"
#include "TBenchmark.h"
#include "TGLViewer.h"
#include "TGLCamera.h"
#include "TGLPerspectiveCamera.h"
#include "TPolyMarker3D.h"
#include "TTree.h"
#include "TBranch.h"

#include "ConfigFile.hpp"
#include "InitialConfig.hpp"
#include "Data.hpp"
#include "Particle.hpp"
#include "Box.hpp"
#include "Tube.hpp"
#include "Volume.hpp"
#include "ParticleManifest.hpp"
#include "Materials.hpp"
#include "Constants.hpp"
#include "Units.hpp"
#include "ValidStates.hpp"
#include "GeomParameters.hpp"
#include "Algorithms.hxx"
#include "DataAnalysis.hpp"

#include <boost/program_options.hpp>


namespace po = boost::program_options;

using namespace std;
using namespace GeomParameters;

// Declarations
void GenerateBeam(TFile& file, const InitialConfig& initialConfig);
Bool_t GenerateParticles(TFile& file, const InitialConfig& initialConfig, const TGeoShape*
beamShape, const TGeoMatrix& beamMatrix);
Bool_t CreateRandomParticle(Particle* particle, const Double_t fillTime, const TGeoShape*
beamShape, const TGeoMatrix& beamMatrix);
```

```
Bool_t DetermineParticleVelocity(Particle* particle, const string vel_dist, const Double_t
maxVelocity, const Double_t minTheta, const Double_t maxTheta, const Double_t minPhi, const
Double_t maxPhi);
void DefinePolarisation(Particle* particle, const Double_t percentPolarised, const TVector3&
spinAxis, const Bool_t spinUp);
void DrawHistograms(TFile& file);

// Namespace for defining titles of histograms
namespace Hist {
  const int nbins = 100;
  const char* initialX       = "initial:X";
  const char* initialXTitle  = "X (m)";
  const char* initialY       = "initial:Y";
  const char* initialYTitle  = "Y (m)";
  const char* initialZ       = "initial:Z";
  const char* initialZTitle  = "Z (m)";
  const char* initialVX      = "initial:VX";
  const char* initialVXTitle = "VX (m/s)";
  const char* initialVY      = "initial:VY";
  const char* initialVYTitle = "VY (m/s)";
  const char* initialVZ      = "initial:VZ";
  const char* initialVZTitle = "VZ (m/s)";
  const char* initialV       = "initial:V";
  const char* initialVTitle  = "V (m/s)";
  const char* initialT       = "initial:T";
  const char* initialTTitle  = "T (s)";

  const char* initialPositions = "initial:Positions";
}

//_____________________________________________________________________________
Int_t main(Int_t argc,Char_t **argv)
{
  try {
    // -- Create a description for all command-line options
    po::options_description description("Allowed options");
    description.add_options()
      ("help", "produce help message")
      ("config", po::value<string>(), "name of configfile to be read in")
      ("plot", po::value<bool>()->default_value(false), "toggle for whether to display plots of the
generated neutrons");
    ;

    // -- Create a description for all command-line options
    po::variables_map variables;
    po::store(po::parse_command_line(argc, argv, description), variables);
    po::notify(variables);

    // -- If user requests help, print the options description
    if (variables.count("help")) {
      cout << description << "\n";
      return EXIT_SUCCESS;
```

```
    }

    // -- Check whether a datafile was given. If not, exit with a warning
    if (variables.count("config")) {
      cout << "Configuration file to be read is: "
           << variables["config"].as<string>() << "\n";
    } else {
      cout << "Error: No configuration file given (eg: config.cfg)\n";
      cout << description << "\n";
      return EXIT_FAILURE;
    }

    // Read in Batch Configuration file to find the Initial Configuration File
    ConfigFile configFile(variables["config"].as<string>());
    // Read in Initial Configuration from file.
    InitialConfig initialConfig(configFile);
    // Open output file
    const string outputFileName = initialConfig.OutputFileName();
    TFile* file = Analysis::DataFile::OpenRootFile(outputFileName,"RECREATE");
    // Generate particles
    GenerateBeam(*file, initialConfig);

    if (variables["plot"].as<bool>() == true) {
      // Enter ROOT interactive session
      TRint *theApp = new TRint("FittingApp", NULL, NULL);
      DrawHistograms(*file);
      theApp->Run();
    }
    // Close file
    file->Close();

  } catch(exception& e) {
     cerr << "error: " << e.what() << "\n";
     return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}

//_____________________________________________________________________________
void GenerateBeam(TFile& file, const InitialConfig& initialConfig)
{
  // -- Make a 'virtual' beam volume within which we will generate our initial particles
  const std::string beamShapeName = initialConfig.BeamShape();
  const Double_t beamRMin = 0.0;
  const Double_t beamRMax = initialConfig.BeamRadius();
  const Double_t beamHalfLength = initialConfig.BeamLength()/2.0;
  const Double_t beamPhi = initialConfig.BeamPhi();
  const Double_t beamTheta = initialConfig.BeamTheta();
  const Double_t beamPsi = initialConfig.BeamPsi();
  const Double_t beamXPos = initialConfig.BeamDisplacement().X();
  const Double_t beamYPos = initialConfig.BeamDisplacement().Y();
  const Double_t beamZPos = initialConfig.BeamDisplacement().Z();
```

```
  // Create the Beam volume within which we shall generate the particles
  TGeoShape* beamShape = NULL;
  if (beamShapeName == "Tube") {
    beamShape = new Tube("BeamShape", beamRMin, beamRMax, beamHalfLength);
  } else {
    beamShape = new Box("BeamShape", beamRMax, beamRMax, beamHalfLength);
  }
  // Create rotation matrix for beam volume
  TGeoRotation beamRot("BeamRot",beamPhi,beamTheta,beamPsi);
  TGeoTranslation beamTra("BeamTra",beamXPos,beamYPos,beamZPos);
  TGeoCombiTrans beamCom(beamTra,beamRot);
  TGeoHMatrix beamMatrix = beamCom;

  // -- Generate the particles
  GenerateParticles(file, initialConfig, beamShape, beamMatrix);
}

//_____________________________________________________________________________
Bool_t GenerateParticles(TFile& file, const InitialConfig& initialConfig, const TGeoShape*
beamShape, const TGeoMatrix& beamMatrix)
{
  // Generates a uniform distribution of particles with random directions all with
  // the same total energy (kinetic plus potential) defined at z = 0.
  const Int_t particles = TMath::Abs(initialConfig.InitialParticles());
  const string vel_dist = initialConfig.VelocityDistribution();
  const Double_t vmax = TMath::Abs(initialConfig.InitialMaxVelocity())*Units::m/Units::s;
  const Double_t maxEnergy = 0.5*Neutron::mass_eV_c2*TMath::Power(vmax,2.0);
  const Double_t fillTime = TMath::Abs(initialConfig.FillingTime())*Units::s;

  const Double_t percentPolarised = initialConfig.PercentPolarised();
  const TVector3 spinAxis = initialConfig.SpinAxis();
  const Bool_t spinUp = initialConfig.SpinUp();

  const Double_t minTheta = initialConfig.DirectionMinTheta();
  const Double_t maxTheta = initialConfig.DirectionMaxTheta();
  const Double_t minPhi = initialConfig.DirectionMinPhi();
  const Double_t maxPhi = initialConfig.DirectionMaxPhi();

  // Calculate the boundaries of the beam volume using the beam matrix
  const Box* boundary = dynamic_cast<const Box*>(beamShape);
  Double_t posBounds[3] = {0.,0.,0.};
  Double_t negBounds[3] = {0.,0.,0.};
  Double_t posLocalBounds[3] = {boundary->GetDX(), boundary->GetDY(), boundary-
>GetDZ()};
  Double_t negLocalBounds[3] = {-boundary->GetDX(), -boundary->GetDY(), -boundary-
>GetDZ()};
  beamMatrix.LocalToMaster(posLocalBounds, posBounds);
  beamMatrix.LocalToMaster(negLocalBounds, negBounds);

  // Create Histograms to view the initial particle distributions
  TH1F* initialXHist = new TH1F(Hist::initialX, Hist::initialXTitle, Hist::nbins, negBounds[0],
posBounds[0]);
```

```
  TH1F* initialYHist = new TH1F(Hist::initialY, Hist::initialYTitle, Hist::nbins, negBounds[1],
posBounds[1]);
  TH1F* initialZHist = new TH1F(Hist::initialZ, Hist::initialZTitle, Hist::nbins, negBounds[2],
posBounds[2]);
  TH1F* initialVXHist = new TH1F(Hist::initialVX, Hist::initialVXTitle, Hist::nbins, -vmax,
vmax);
  TH1F* initialVYHist = new TH1F(Hist::initialVY, Hist::initialVYTitle, Hist::nbins, -vmax,
vmax);
  TH1F* initialVZHist = new TH1F(Hist::initialVZ, Hist::initialVZTitle, Hist::nbins, -vmax,
vmax);
  TH1F* initialVHist = new TH1F(Hist::initialV, Hist::initialVTitle, Hist::nbins, 0.0, vmax);
  TH1F* initialTHist = new TH1F(Hist::initialT, Hist::initialTTitle, Hist::nbins, 0.0, fillTime);
  // Create markers for all the initial particle's positions
  TPolyMarker3D* positions = new TPolyMarker3D(particles, 1); // 1 is marker style

  cout << "-----------------------------------------" << endl;
  cout << "Generating " << particles << " Particles." << endl;
  cout << "Boundary X (m): " << boundary->GetDX() << "\t";
  cout << "Y (m): " << boundary->GetDX() << "\t";
  cout << "Z (m): " << boundary->GetDX() << endl;
  cout << "Max Energy (neV): " << maxEnergy/Units::neV << endl;
  cout << "-----------------------------------------" << endl;

  ////////////////////////////////////////////////////////////////////////////////
  // -- Create storage for the particles
  TTree tree("Particles","Tree of Particle Data");
  Particle* particle = new Particle(0, Point(), TVector3());
  TBranch* initialBranch = tree.Branch(States::initial.c_str(), particle->ClassName(), &particle);
  ParticleManifest manifest;
  ////////////////////////////////////////////////////////////////////////////////
  // -- Initialise random number generator with a unique state
  gRandom->SetSeed(0);
  ////////////////////////////////////////////////////////////////////////////////
  // -- Loop over the total number of particles to be created.
  for (Int_t i = 1; i <= particles; i++) {
    // -- Create particle at a random position inside beam volume
    particle->SetId(i);
    CreateRandomParticle(particle, fillTime, beamShape, beamMatrix);
    // -- Initialise particle's momentum
    DetermineParticleVelocity(particle, vel_dist, vmax, minTheta, maxTheta, minPhi, maxPhi);
    // -- Setup polarisation
    DefinePolarisation(particle, percentPolarised, spinAxis, spinUp);
    // -- Fill histograms
    initialXHist->Fill(particle->X());
    initialYHist->Fill(particle->Y());
    initialZHist->Fill(particle->Z());
    initialVXHist->Fill(particle->Vx());
    initialVYHist->Fill(particle->Vy());
    initialVZHist->Fill(particle->Vz());
    initialVHist->Fill(particle->V());
    initialTHist->Fill(particle->T());
    positions->SetPoint(i-1, particle->X(), particle->Y(), particle->Z());
```

```
    // -- Add particle to data file
    initialBranch->Fill();
    int branchIndex = initialBranch->GetEntries() - 1;
    manifest.AddEntry(States::initial, particle->Id(), branchIndex);
    // -- Update progress bar
    Algorithms::ProgressBar::PrintProgress(i,particles,1);
  }
  cout << "Successfully generated " << particles << " particles." << endl;
  cout << "----------------------------------------" << endl;
  // -- Write the tree and manifest to file
  manifest.Write();
  tree.Write();
  // -- Navigate to histogram folder
  Analysis::DataFile::NavigateToHistDir(file);
  // -- Save initial state plots to histogram folder
  initialXHist->Write(initialXHist->GetName(),TObject::kOverwrite);
  initialYHist->Write(initialYHist->GetName(),TObject::kOverwrite);
  initialZHist->Write(initialZHist->GetName(),TObject::kOverwrite);
  initialTHist->Write(initialTHist->GetName(),TObject::kOverwrite);
  initialVXHist->Write(initialVXHist->GetName(),TObject::kOverwrite);
  initialVYHist->Write(initialVYHist->GetName(),TObject::kOverwrite);
  initialVZHist->Write(initialVZHist->GetName(),TObject::kOverwrite);
  initialVHist->Write(initialVHist->GetName(),TObject::kOverwrite);
  // -- Write the initial positions out to file
  Analysis::DataFile::NavigateToHistDir(file);
  positions->Write(Hist::initialPositions,TObject::kOverwrite);
  // -- Write the geometry used for visualisation also to the file
  string geomFileName = initialConfig.GeomVisFileName();
  if (geomFileName.empty()) geomFileName = initialConfig.GeomFileName();
  cout << "275" << endl;
  cout << "geomFileName " << geomFileName << endl;
  TGeoManager* geoManager = TGeoManager::Import(geomFileName.c_str());
  cout << "277" << endl;
  if (geoManager == NULL) return EXIT_FAILURE;
  file.cd();
  geoManager->Write("Geometry",TObject::kOverwrite);

  // cleanup
  delete initialXHist;
  delete initialYHist;
  delete initialZHist;
  delete initialTHist;
  delete initialVXHist;
  delete initialVYHist;
  delete initialVZHist;
  delete initialVHist;
  delete geoManager;
  delete particle;
  delete positions;
  return kTRUE;
}
```

```
//_____________________________________________________________________________
Bool_t CreateRandomParticle(Particle* particle, const Double_t fillTime, const TGeoShape*
beamShape, const TGeoMatrix& beamMatrix)
{
  // -- Find a random point inside the Volume
  Double_t point[3] = {0.,0.,0.}, localPoint[3] = {0.,0.,0.};

  // -- Determine the dimensions of the source volume's bounding box.
  const TGeoBBox* boundingBox = static_cast<const TGeoBBox*>(beamShape);
  const Double_t boxWall[3] = {boundingBox->GetDX(), boundingBox->GetDY(), boundingBox-
>GetDZ()};
  const Double_t boxOrigin[3] = {boundingBox->GetOrigin()[0], boundingBox->GetOrigin()[1],
boundingBox->GetOrigin()[2]};

  // Find a random point inside the volume provided
  while(kTRUE) {
    // First generate random point inside bounding box, in the local coordinate frame of the box
    for (Int_t i=0; i<3; i++) localPoint[i] = boxOrigin[i] + boxWall[i]*gRandom->Uniform(-1.0,
1.0);
    // Then test to see if this point is inside the volume
    if (beamShape->Contains(&localPoint[0]) == kFALSE) {
      continue;
    } else {
      break;
    }
  }
  // Next transform point to the global coordinate frame
  beamMatrix.LocalToMaster(localPoint, point);
  // -- Set Starting time
  // Pick a time uniformly between start, and the filling time of the experiment
  // Thus a particle can be generated at any time in the filling time process when the beam is on
  Double_t startTime = gRandom->Uniform(0.0,fillTime);
  // -- Create Particle
  particle->SetPosition(point[0], point[1], point[2], startTime);

  return kTRUE;
}

//_____________________________________________________________________________
Bool_t DetermineParticleVelocity(Particle* particle, const string vel_dist, const Double_t
maxVelocity, const Double_t minTheta, const Double_t maxTheta, const Double_t minPhi, const
Double_t maxPhi)
{
  // -- Determine a random direction vector on the unit sphere dOmega = sin(theta).dTheta.dPhi
  // Convert limits to radians
  Double_t minThetaRad = minTheta*TMath::Pi()/180.0;
  Double_t maxThetaRad = maxTheta*TMath::Pi()/180.0;
  Double_t minPhiRad = minPhi*TMath::Pi()/180.0;
  Double_t maxPhiRad = maxPhi*TMath::Pi()/180.0;
  Double_t minU = -TMath::Cos(minThetaRad);
  Double_t maxU = -TMath::Cos(maxThetaRad);
  // Get Random angle between limits
```

```
  Double_t phi = gRandom->Uniform(minPhiRad,maxPhiRad);
  Double_t u   = gRandom->Uniform(minU,maxU);
  Double_t theta = TMath::ACos(-u);
  // Convert spherical polars into cartesian nx,ny,nz
  Double_t dir[3];
  dir[0] = TMath::Cos(phi)*TMath::Sin(theta);
  dir[1] = TMath::Sin(phi)*TMath::Sin(theta);
  dir[2] = TMath::Cos(theta);
  // Check that it is a normalised direction vector
  assert(TMath::Abs(TMath::Sqrt(dir[0]*dir[0] + dir[1]*dir[1] + dir[2]*dir[2]) - 1.0) < 1.E-10);

  // -- Determine Particle Velocity
  Double_t velocity = 0.0;
  if (vel_dist == VelocityDistributions::mono) {
    // -- Mono-energetic distribution. All particles have SAME TOTAL energy (Kinetic + potential)
    double maxTotalEnergy = 0.5*Neutron::mass_eV_c2*TMath::Power(maxVelocity,2.0);
    double gravPotE = Neutron::mass_eV_c2*Constants::grav_acceleration*particle->Z();
    double kineticE = maxTotalEnergy - gravPotE;
    velocity = TMath::Sqrt(2.0*kineticE/Neutron::mass_eV_c2);
    if (velocity < 0.0 || velocity > maxVelocity) {
      cout << "Error, maxVelocity is set too low for height of beam volume : " << velocity << "\t"
<< particle->Z() << endl;
      cout << "root(2*g*h) " << TMath::Sqrt(2.0*Constants::grav_acceleration*particle->Z()) <<
"\t" << maxVelocity << endl;
      throw runtime_error("Error, maxVelocity is set too low for height of beam volume");
    }
  } else if (vel_dist == VelocityDistributions::uniform) {
    // -- Uniform distribution
    // Define velocity as just the max velocity
    velocity = maxVelocity;
  } else if (vel_dist == VelocityDistributions::v) {
    // -- V distribution
    // Define velocity as distributed along linear v distribution
    Double_t normalisation = (2.)/(TMath::Power(maxVelocity,2.0));
    Double_t prob = gRandom->Uniform(0.0,1.0);
    velocity = TMath::Power(((2.0*prob)/normalisation),(1.0/2.0));
  } else if (vel_dist == VelocityDistributions::v_squared) {
    // -- V^2 distribution
    // Define velocity as distributed along v^2 curve
    Double_t normalisation = (3.)/(TMath::Power(maxVelocity,3.0));
    Double_t prob = gRandom->Uniform(0.0,1.0);
    velocity = TMath::Power(((3.0*prob)/normalisation),(1.0/3.0));
  } else {
    // Default case
    throw runtime_error("Error, unrecognised velocity distribution set");
  }
  // Set velocity
  Double_t vel[3];
  vel[0] = velocity*dir[0];
  vel[1] = velocity*dir[1];
  vel[2] = velocity*dir[2];
  particle->SetVelocity(vel[0], vel[1], vel[2]);
```

```
  return kTRUE;
}

//_____________________________________________________________________________
void DefinePolarisation(Particle* particle, const Double_t percentPolarised, const TVector3&
spinAxis, const Bool_t spinUp)
{
  Double_t percentage;
  // Check if percent polarised falls outside allowed bounds
  if (percentPolarised < 0.0 || percentPolarised > 100.0) {
    throw runtime_error("Error, unrecognised polarisation percentage");
  } else {
    percentage = percentPolarised;
  }
  // For the polarised ones define along provided axis
  if (gRandom->Uniform(0.0, 100.0) <= percentage) {
    particle->Polarise(spinAxis, spinUp);
  } else {
    particle->PolariseRandomly();
  }
}

//_____________________________________________________________________________
void DrawHistograms(TFile& file)
{
  // Read all histograms and geometry into memory
  TDirectory* histdir = Analysis::DataFile::NavigateToHistDir(file);
  TH1F* initialXHist   = static_cast<TH1F*>(histdir->Get(Hist::initialX));
  TH1F* initialYHist   = static_cast<TH1F*>(histdir->Get(Hist::initialY));
  TH1F* initialZHist   = static_cast<TH1F*>(histdir->Get(Hist::initialZ));
  TH1F* initialVXHist  = static_cast<TH1F*>(histdir->Get(Hist::initialVX));
  TH1F* initialVYHist  = static_cast<TH1F*>(histdir->Get(Hist::initialVY));
  TH1F* initialVZHist  = static_cast<TH1F*>(histdir->Get(Hist::initialVZ));
  TH1F* initialVHist   = static_cast<TH1F*>(histdir->Get(Hist::initialV));
  TH1F* initialTHist   = static_cast<TH1F*>(histdir->Get(Hist::initialT));
  TPolyMarker3D* positions = static_cast<TPolyMarker3D*>(histdir->Get(Hist::initialPositions));

  // Draw the histograms describing the initial particles' states
  initialXHist->SetLineColor(kBlue);
  initialXHist->SetFillStyle(3001);
  initialXHist->SetFillColor(kBlue);
  initialYHist->SetLineColor(kBlue);
  initialYHist->SetFillStyle(3001);
  initialYHist->SetFillColor(kBlue);
  initialZHist->SetLineColor(kBlue);
  initialZHist->SetFillStyle(3001);
  initialZHist->SetFillColor(kBlue);
  initialTHist->SetLineColor(kBlue);
  initialTHist->SetFillStyle(3001);
  initialTHist->SetFillColor(kBlue);

  initialVXHist->SetLineColor(kRed);
```

```
  initialVXHist->SetFillStyle(3001);
  initialVXHist->SetFillColor(kRed);
  initialVYHist->SetLineColor(kRed);
  initialVYHist->SetFillStyle(3001);
  initialVYHist->SetFillColor(kRed);
  initialVZHist->SetLineColor(kRed);
  initialVZHist->SetFillStyle(3001);
  initialVZHist->SetFillColor(kRed);
  initialVHist->SetLineColor(kRed);
  initialVHist->SetFillStyle(3001);
  initialVHist->SetFillColor(kRed);

  TCanvas *canvas1 = new TCanvas("InitialPhaseSpace","Initial Phase Space",60,0,1000,800);
  canvas1->Divide(4,2);
  canvas1->cd(1);
  initialXHist->Draw();
  canvas1->cd(2);
  initialYHist->Draw();
  canvas1->cd(3);
  initialZHist->Draw();
  canvas1->cd(4);
  initialTHist->Draw();
  canvas1->cd(5);
  initialVXHist->Draw();
  canvas1->cd(6);
  initialVYHist->Draw();
  canvas1->cd(7);
  initialVZHist->Draw();
  canvas1->cd(8);
  initialVHist->Draw();

  // Draw the Geometry and initial particle postions
  TGeoManager& geoManager = Analysis::DataFile::LoadGeometry(file);
  TCanvas* canvas2 = new TCanvas("initial:Positions","Positions",60,0,100,100);
  double cameraCentre[3] = {0.,0.,0.};
  Analysis::Geometry::DrawGeometry(*canvas2, geoManager, cameraCentre);
  positions->SetMarkerColor(2);
  positions->SetMarkerStyle(6);
  positions->Draw();
}
```