NGen User's Guide

Chris Kachulis
kachulis@bu.edu or cjkachulis@gmail.com

February 9, 2015

Contents

1 Introduction **1**

2 Basic Structure **1**
2.1 The Four Abstract Base Classes 1
2.2 The Two Data Classes 2

3 Writing an Application: The Basics **2**
3.1 EvtVector 3
3.2 FluxGen 3

4 Writing an Application: Beyond the Basics **6**
4.1 User Defined Data 6

1 Introduction

NGen is a software package for neutrino vector generation for large single (of few) volume detecors. This users guide will go over how to create an application to generate neutrino vectors using NGen. Examples referenced in this guide can be found in NGen/examples.

2 Basic Structure

NGen is built around four abstract base classes which the User must inherit to write their application. (Some useful child classes are included in the package and can be used by the User instead of inheriting their own). Additionally two data classes hold information about the current event being produced. This section will briefly introduce these six classes. The basics required to use them to build an application will be in section 3.

2.1 The Four Abstract Base Classes

FluxGen (src/core/FluxGen.[h,cc]): FluxGen performs most of the heavy lifting of the simulation. This is where the user must provide an interface with the chosen nuclear simulation code. The user must also pass instances of child classes derived from the other three abstract base classes to their derived FluxGen child.

FlxTable (src/core/FlxTable.[h,cc]): FlxTable handles the information about the flux model being used.

DetectorGeom (src/core/DetectorGeom.[h,cc]): DetectorGeom handles the detector geometry, and the density of the material of the detector. Some basic geometries are included in src/targets/.

Writer (src/core/Writer.[h,cc]): Writer handles writing the resulting data to output files in whatever format the User chooses.

2.2 The Two Data Classes

These two classes handle the vector currently being produced. They are not intended to be inherited by the user, but the user is responsible for filling instances of them with information about the current event.

EvtVector (src/core/EvtVector.[h,cc]): EvtVector holds the information about the event, such as position, time, interaction code, etc. It also has a field which is a vector of Tracks, which is where the information about particular particles in the event is stored.

Track (src/core/Track.[h,cc]): Track is an individual particle involved in the event. It holds all information about a particular particle.

3 Writing an Application: The Basics

In order to write an application, the User must inherit the four abstract base classes introduced in section 2.1. I will shortly go through the pure virtual methods which must be overwritten by the user in their inherited child classes. First, however, I will go into more detail about the two data classes. Handling these two classes is a large part of the User's job in interfacing with the chosen nuclear simulator and writing out the data.

3.1 EvtVector

EvtVector has five basic data fields. Of these five, four are set by methods in FluxGen without the User doing anything. These are:
vertex: The position of the event in cm.
evtTime: The time of the event as a Time_Struct object (see src/core/-TimeStruct.h), which has a seconds and nanoseconds field.
evtNumber: Incremented with each event that begins the generation process. **evtWrittenNumber**: Incremented every time an event is written out (sometimes you may choose not to write out a particular event while you are producing it, and so the evtNumber will be incremented but not evtWrittenNumber. This is all done automatically, the user doesn't need to worry about this).
In most cases the user will have no reason to change the values of these fields from whatever the methods in FluxGen has set them to. However, there are some cases where a user may actually want to change these values, so they are left public. Just make sure you know what you're doing if you decide to change them in your code. Particularly messing with the evtTime field could lead to bad results if you don't understand exactly what you're doing.
The fifth data field is the users responsibility. It should be set by the User (see FluxGen::ProcessIncomingTrack() in Section 3.2. **intMode:** This holds the interaction mode of the event. Although, unlike the other four fields,

it is only ever touched by the user, so you could put a binary encoding of your shopping list here and the program would happily plug away oblivious to your practical joke.

3.2 FluxGen

IMPORTANT: The information about the current vector is held in the field FluxGen::currentVect which is a pointer to a EvtVector object. The first Track in the currentVect is set to the incoming neutrino information of the base class, and can be accessed by the User with currentVect->GetTrack(0).
Pure Virtual Methods

double GetCrossSection(NEUTRINO::FLAVOR flav, float E): Return cross section of given neutrino flavor and given energy in units of 10^{-38}cm^2/ nucleon. Energy (E) is given in GeV. In the example below (examples/SK_Atm_Neut/AtmFluxGen.cc) fntotpau_ is the external call to the NEUT function which gives the neutrino cross section.

Listing 1: AtmFluxGen::GetCrossSection

```
double AtmFluxGen::GetCrossSection(NEUTRINO::FLAVOR flav,
    float E)
{
  int iPar[6]={12,-12,14,-14,16,-16};
  double ret;

  ret=fntotpau_(iPar[flav],E);

  return ret;
}
```

int ProcessIncomingTrack(): Process the incoming neutrino track (currentVect-¿GetTrack(0)) by running it through your nuclear simulator and recording resulting vector information in currentVect. FluxGen will already have stored the vertex,evtTime, evtVumber, and evtWrittenNumber, as well as the basic information of the incoming neutrino (track 0). Return 0 if successful and you want to write out the event, 1 if successfull but you do not want to write out the event, and -1 if there was an error (this will end the program).

You must only set the interaction type, and add any additional particles to the vector. For clarification I'll walk through List 2 below from examples/SK_Atm_Neut/AtmFluxGen.cc. NB. calls to "SetUserData" which are in the source file have been removed from the example here. These calls set non-basic data specific to the user applications, and are explained in Section 4.1.

Listing 2: AtmFluxGen::ProcessIncomingTrack

```
int AtmFluxGen::ProcessIncomingTrack()
{


  Track * track=currentVect->GetTrack(0);
  float pMom[3];


  for(int i=0;i<3;i++)
    {
      pMom[i]=track->pdir[i]*track->momentum;
    }
  int iErr=0;

  vcclfsicm_();
  std::cerr<<"parID: "<<track->parID<<std::endl;
  float pos[3];
  pos[0]=currentVect->vertex.X();
  pos[1]=currentVect->vertex.Y();
  pos[2]=currentVect->vertex.Z();
  nevect_(track->parID,pos,pMom,&iErr);

  if(iErr>0)
    {

      return -1;
    }
  //set interaction type
  currentVect->intType=nework_.modene;
  //loop through resulting tracks and store in
     currentVect
  Track * tmpTrack;
  //  Track * testTrack=currentVect->AddNewTrack();
  //testTrack->UserData["test"]=4.755;
  for(int i=1;i<vcwork_.nvc;i++)
    {
      tmpTrack=currentVect->AddNewTrack();
      tmpTrack->parID=vcwork_.ipvc[i];
      tmpTrack->mass=vcwork_.amasvc[i];
      float P=sqrt(pow(vcwork_.pvc[i][0],2)+pow(vcwork_.
         pvc[i][1],2)+pow(vcwork_.pvc[i][2],2));
      tmpTrack->momentum=P;
      if(P>0)
  {
    for(int j=0;j<3;j++)
      {
        tmpTrack->pdir[j]=vcwork_.pvc[i][j]/P;
      }
  }
      else
  {
    for(int j=0;j<3;j++)
      {
        tmpTrack->pdir[j]=0;
      }

  }
    }
  return 0;
}
```

4 Writing an Application: Beyond the Basics

4.1 User Defined Data