Readme for Simulations

Jeremy Lopez

July 9, 2009

Contents

1 Introduction **3**
1.1 Downloading and Compiling the Code 3

2 Classes **3**
2.1 ParticleGenerator 4
2.2 EventGenerator 5
2.3 RunGenerator 6
2.4 SimCamera 6
2.5 SimChamber 6
2.6 SimTools 7

3 Scripts **7**
3.1 makeRunParameterFiles.py 7
3.2 readFromFile and readFromFile.cxx 8
3.3 multipleRunsFromFile 9
3.4 matchFiles and matchFiles.awk 9

4 Running the Monte Carlo **9**
4.1 Creating Parameter Files 9
4.2 Making a Run 10
4.3 Description of Output 10

5 Examples **11**

6 Future Work **11**

1 Introduction

This is the readme file for the simulation software I have created in summer 2009. It has several important advantages over earlier versions. Among these are:

- Designed to use DmtpcDataset data type
- Separate camera class will eventually allow for use of multiple cameras
- Will allow for simulation of electronics readout
- Should be easier to use and modify
- Memory problems have been solved

This Monte Carlo is set up to run neutron, alpha and WIMP events and currently can add multiple tracks to a single image, but only saves one readout and the true MC event data for a single track.

1.1 Downloading and Compiling the Code

To run the code, it is necessary to have the DMTPC MaxCam package and all of the software that this requires (C++, ROOT, etc). The current version of the Monte Carlo (version 1) may be found in CVS in the folder `projects/DarkMatter/MaxCam/Simulations/v1/`.

There are several new classes that are used in the simulation, and these must be compiled. To do this, go to the `v1/code/` folder and type

```
% make clean
% make
```

This will run a make file and create a shared library file in the `v1/lib/` folder. Make sure that in the `rootlogon.C` file in the `v1/` folder that you load this file when you start ROOT.

2 Classes

The simulation uses a series of new classes and a new namespace that were created for the simulation. Making this simulation much more object-oriented than the last should allow it to be more flexible and easier to use and modify.

2.1 ParticleGenerator

As the name might suggest, the `ParticleGenerator` class holds all of information needed to generate a projectile and its corresponding recoil. Randomization for all of the classes is done with `TRandom3` objects. Currently, the following options are supported, although several options such as the special options, may supersede others:

- Recoil Type
  - Alpha: Projectile and recoil particle are identical
  - Neutron: Elastic scattering of a particle off of a gas molecule
  - Wimp: Like neutron but allows for use of wimp distribution
- Projectile Particle
- Energy Option
  - Fixed: Only a single energy used
  - Random: Randomized between given limits
- Position Option
  - Fixed: Only a single position used
  - Random: Randomized between given limits
- Direction Option
  - Fixed: Only a single direction used
  - Isotropic: Evenly distributed over all angles
  - Source: Find vector between given source position and recoil position
  - Collimated: Collimated to a given number of degrees from a source direction
- Time Option
  - Fixed: Only a single time used
  - Random: Randomized between given limits

  - Current: Use current time

- Special Option
  - WIMP: Use WIMP distribution (wimp type only)
  - ENDF: Use ENDF files to generate neutron energy spectrum and recoil distribution

Besides these options, there are a number of other parameters which may be set and which are used to recover the information about the generated event.

The latitude and longitude of the event, which is important for WIMP runs, do not need to be entered directly, as the positions of various locations are entered into the code and may be set using the `ParticleGenerator::setLocation()` function. Supported arguments include "dusel", "mit", "wipp", "kamioka", "fermilab", "soudan", "gransasso", "snolab" and "boulby."

2.2 EventGenerator

Currently, much of this class is just a copy of the image generating code of the `MaxCamMC` class. Straggling is currently turned off, as I would like to find new code to simulate this that has no effect on the energy of the recoil, as was the case previously. This class also includes functions to create histograms of a hypothetical charge readout. The main difference in the code is that while `MaxCamMC` saved tracks onto a histogram and then simulated diffusion and other effects to create a new histograms, the `EventGenerator` class uses vectors. The advantage of this is that the z-position of each point in the track is not lost. This will be important when simulating charge readouts and eventually when we can simulate one track covering more than one image.

Currently, this just generates a single image and a single charge readout, but in the future it should allow for several readouts and images related to the same region in the detector. It is likely that dual TPC mode datasets will not be generated, as what happens in one TPC should not affect the other.

This class also controls spacer and radial effects. To generate a recoil event and simulate the recoil effects in the detector, use the `EventGenerator::runEvent()` function. If the input is `true`, any existing tracks present on the image will be erased prior to image creation for this track.

2.3 RunGenerator

The `RunGenerator` class is the class that controls input/output and, as its name suggests, generating a series of events that will make up a run. It primarily manages looping over the event generating code, saving data files and outputting run summary text files.

The classes are designed so that this is the only class that needs to be used in a ROOT macro used to generate runs. Thus, the macro may only need to be ten to twenty lines long, which is much more reasonable than the 500+ line macro in the older MC.

One important thing that has been changed in this simulation is the way that parameters are set. While all of the parameters may be hard coded into the macro, but if we want to make the parameters inputs rather than hard coded numbers, we would need functions with dozens of arguments. To get around this, the `RunGenerator` class has the function `RunGenerator::readParametersFromFile()`. The contents of the file, whose path is the input to this function, will be discussed later. However, it is important to note that the change is basically that this class will read in text files containing all of the parameters and automatically set them. Thus, we can change any of the parameters without needing to alter the code in our macros.

2.4 SimCamera

This class holds all of the information about a CCD camera, such as its gain, read noise, pixel counts, bin counts, and imaging region size. It also includes methods to convert between pixels, bins, and detector position, since this information cannot be contained in a single `TH2F` image.

2.5 SimChamber

This class holds all of the information about the detector chamber, such as voltages, pressure, spacer size and positions, and diffusion constants.

Transverse diffusion here is modeled to be Gaussian with a z-dependent variance given by

$$\sigma_T^2(z) = A + Bz \tag{1}$$

where A and B are the diffusion constant and dz terms, respectively. While you may set the dz term explicitly if you wish, this class also includes the data

included in the diffusion measurement paper, where the constant is obtained from the pressure, temperature, drift length and drift voltage.

2.6 SimTools

`SimTools` is a namespace that contains several methods used by the other classes to do things such as convert between `TTimeStamp` and `TDatime` time-keeping objects. This is meant for useful methods that don't need to be part of any of the classes.

3 Scripts

I have also included several scripts to allow users to run the simulation without needing to write large amounts of code for their own personal directories. These scripts are described in this section and may be found in the `/scripts/` directory.

3.1 makeRunParameterFiles.py

This file is a Python 2 script used to automatically create the text files that are to be read in by the `RunGenerator` object. It will not work with Python 3, so be sure that you use the correct path to Python 2 in the top line of the file or just explicitly call Python when you want to run the program. If you must use Python 3, then the only thing that you need to do to make this script work is change all instances of `raw_input` to `input`. This can be done in VI or VIM like this:

```
% vi scripts/makeRunParameterFiles.py
[> = command in vi normal mode]
> :%s/raw_input/input/g
> ZZ
```

When you run the script, it will ask several questions about the run and then display some of the parameters and allow you to edit them. It should be very straightforward, and if you need to add a parameter, modifying the code should be fairly easy as well. You just need to modify this script and the corresponding input function in `RunGenerator.cc`.

The script should be run like this:

```
% ./scripts/makeRunParameterFiles.py
```

The output, using the default filenames, is as follows (all in the `/runParameters/` directory):

- `fileManager.temp`: This is a file containing the names of the other files in the order given below. You will use this file's name as the input to other scripts and to `RunGenerator::readParametersFromFile()`.
- `runProperties.temp`: This file holds all of the parameters not related to the camera and chamber properties. The order does not matter, but the keywords (case sensitive) are important as is the order of the tokens on each line. Not every parameter must be included in any of these files, but anything that is not included will revert to default and will not be included in the run summary file.
- `chamberProperties.temp`: This file holds all of the parameters related to the actual chamber being simulated.
- `cameraPropertiesX.temp`: This holds all of the camera properties. X is an integer starting with 0 and counting up as you add more cameras. Multiple cameras are not yet implemented, so the X=0 case is the only one that matters at present.

3.2 readFromFile and readFromFile.cxx

These two files are a BASH script and the ROOT macro it calls to make one run. Although the name says that it will read from a file, you do not need a file for it to work. With no input argument, the script will call `makeRunParameterFiles.py` and then use the output of that script to make a run. If you've already run the Python script and don't want to do that again, then use the file manager filename as the input. Using the defaults, the command would be like this:

```
% ./scripts/readFromFile $filename &
[or: run in background]
% nohup ./scripts/readFromFile $filename &> nohup.log &
```

3.3 multipleRunsFromFile

This is a simple script that will create several identical runs using `readFromFile` and a set of files containing the run information. The syntax is as follows:

```
% ./scripts/multipleRunsFromFile $filename $number_of_runs &
```

3.4 matchFiles and matchFiles.awk

These are files that will look through the run summary files (`.sum` files found in `output/summary/`) and output a list of all the ROOT files matching a set of parameters entered into `matchFiles.awk`. The default currently is to search for all runs of "neutron" event type with gain of 9.3 and read noise of 7.3, as in our current camera 0. The AWK script is short and should be easy enough to modify for whatever parameters you wish to looks for. This is possible because the summary columns list a single parameter per line. Each parameter has a unique name, which is held in the first column, while the following columns are values. As with the other scripts this can be run like this:

```
% ./scripts/matchFiles
[or: move output to files.txt]
% ./scripts/matchFiles > files.txt
% less files.txt
```

4 Running the Monte Carlo

If you have read the entire document up to now, you likely have a good idea how to run the Monte Carlo. If not, here is the minimum amount of information you need to create a run. All of the command are from the `Simulations/v1/` directory.

4.1 Creating Parameter Files

Once you have compiled all of the code and entered the correct name of the shared library in the `rootlogon.C` file, you may begin. To create the parameter files you need to read into the MC classes, first put together a list of all of the parameters you want for your run, including names of SRIM and ENDF files: If Python 2 is the default program for the command `python`.

```
% ./scripts/makeRunParameterFiles.py
```

If not, suppose you have Python 2.5, so the command is `python2.5`:

```
% python2.5 scripts/makeRunParameterFiles.py
```

Just follow the directions. This will output four files into the `runParameters/` directory. If you wish to rename these, make sure you place them all in the same directory and make sure the names (with no directories) in `fileManager.temp` match the new names.

4.2 Making a Run

Assuming you did not change any of the names, you can make a single run with

```
% ./scripts/runFromFile runParameters/fileManager.temp &
```

You can make N identical runs with

```
% ./scripts/multipleRunsFromFile runParameters/fileManager.temp N &
```

You may want to use `nohup` if you think you might need to log off the computer. This is especially important for alpha runs, which can take a long time.

4.3 Description of Output

All of the output will be found, naturally, in the `output/` directory. There are three subdirectories in this folder. Runs are labeled by date (GMT, YYMMDD) and run number (XXX) starting with 001 for each day.

The `logs/` subdirectory holds the ROOT output for all of the runs in a single day. It is mostly useful for seeing error messages if something went wrong with your run.

The `summary/` subdirectory holds the summary files for each run. These files are basically just the text files holding the run parameters merged together into a single text file. These are useful to keep track of what you did in each run. If a run does not complete properly, this file will not exist, so any run with no summary file should be deleted.

The `data/` subdirectory holds all of the data files. These files, like real data files, hold a few `TString` comments which you can set, as well as a bias

frame and a `DmtpcDataset` object called "dmtpc." There are also several trees holding the run parameters in case you want to look at these in ROOT as well as a `TTree` object called "Simulation" which holds the projectile and recoil information for each event.

5 Examples

I have included several run parameter files, which are held in the `runParameters` directory. There is one run each for alphas, neutrons, and wimps. I will also in the near future try to add some example output files and maybe some example plots made using the MC.

6 Future Work

There are several things which I would like to add in the future. First, I would like to add a separate class for the electronic readout. It would not be possible to read out the entire imaging area with one channel because the capacitance would be too high, so we need to have a series of smaller readouts. This can be most easily done by making an array of charge readout objects. I would also like to do the same with the cameras. This would also require modifying the output summary files to make camera number part of the entry for each camera parameter.

Some other smaller things to work on are getting a better way to model straggling, which is currently disabled because it was done by changing the energy to match the new length after straggling. I would also like to get an accurate model of electronic readout with realistic parameters and values for these parameters rather than the arbitrary Gaussian it is now.