RAT Documentation

Release 1.0

S. Seibert et al.

Jun 29, 2018

Contents

This manual describes how to configure and run RAT for simulation and analysis. Those who wish to modify the source code of RAT should first be familiar with this guide, then read the Programmer Guide.

RAT-PAC (RAT, Plus Additional Codes) is hosted on GitHub. For information on accessing and working with the code using Git and GitHub, see `Using GitHub with RAT-PAC Code`.

CHAPTER 1

User's Guide

1.1 Overview

1.1.1 Goals

The Reactor Analysis Tool (RAT) is intended to be a framework that combines both Monte Carlo simulation of the Braidwood detector with event-based analysis tasks, like reconstruction. The primary goals are:

- Make it easy to analyze Monte Carlo-generated events as well as data from disk using the same software with only a few command changes. Even in the proposal/R&D phase, where there is no real data, this is still useful for dumping Monte Carlo events to disk to be analyzed by another job. When there is real data, being able to do the analysis with the same code path as was used on Monte Carlo is very reassuring.
- Allow for a modular, user-controlled analysis of events. This includes allowing the user to selected which analysis tasks to perform (different fitters, pruning unneeded data from the event data structure, etc.). It should also be relatively straightforward for users to introduce their own code into the analysis process.
- Separate analysis into small tasks which can be developed asynchronously by different people, yet integrated with minimal (or perhaps zero) pain.
- Integrate into existing GEANT4 and GLG4sim efforts with a minimum of code duplication. As much as possible, RAT should be designed incorporate upgrades of these packages just by relinking. No cut and paste of code (mainly a question with GLG4sim).

1.1.2 Design

The overall design of RAT is much like SNOMAN: View analysis as a big loop, iterated through once for each event. The body of the loop is assembled by the user in their macro file as a list of "processors." A processor is a self-contained module that takes an event as input and does some work on the event, possibly altering the contents of the event data structure. A fitting processor would add reconstruction information to the event structure, and an I/O processor would write the event to disk, but leave the data structure in memory unchanged. (The SNOMAN event loop actually allows for branching, which has not yet been implemented in RAT.)

Processors can read and modify existing events, but where do the events originally come from? This is the job of event "producers." A producer can be something like a Monte Carlo simulation. We might decide to simulate the following reaction:

nu-bar + p -> n + e+

Given the delay between the observation of the positron and the neutron, this single physics event will be detected as (at least) two separate detector events. The job of the event producer is ultimately to generate physics events and hand them over to the processors selected by the user, one event at a time. (Other processors may convert the physics event into detector events. There is a place in the data structure to put multiple detector events.)

Other event generators are possible. Generators which read events from disk or over the network would function in a similar manner, creating event data structures in memory and handing them to the event processors one by one.

The event-sequential nature of this computation model is both powerful and simple, but can be awkward for certain kinds of analyses. Multi-pass analyses, which must go through a list of events more than once, can be implemented in RAT without much difficulty as long as each pass is sequential. Time-correlation processors can also be implemented if the processor buffers some data internally. A processor that needs full random access to the event stream cannot be implemented efficiently in RAT. The events should be dumped to disk and analyzed in some other program (such as ROOT).

1.1.3 Relationships with Other Software

RAT makes use of several other software packages to do the heavy-lifting:

- CLHEP - CLHEP is a library containing classes useful to physics software, such as 3D vectors and random number generators. It is also used by GEANT4.
- GEANT4 - While GEANT4 is intended to simulate particle interactions in detectors, RAT does not use it directly for that purpose, delegating that to GLG4sim. Instead, RAT makes direct use of the GEANT4 command interpreter to provide a language for both interactive use and executing macro files. RAT also uses the GEANT4 compilation system and makefiles.
- ROOT - ROOT is used to load and save objects to/from disk and over the network.
- GLG4sim - This package is a generalized version of the KamLAND Monte Carlo, intended to simulate KamLAND-like neutrino experiments with GEANT4. GLG4sim is a self-contained application, but RAT just uses the classes defined in GLG4sim to create a Monte Carlo event producer that integrates with the rest of the RAT framework. The way in which this was done allows the RAT application to be completely backward compatible with the GLG4sim application (for now, anyway).

In order to control the version of GLG4sim used by RAT, we include the entire GLG4sim source tree. Updates to the RAT copy of GLG4sim are made from the KSU GLG4sim repository manually as needed. Long term, we are planning to slowly modify the GLG4sim classes to integrate with the RAT environment.

1.2 Installation

1.2.1 Prerequisites

These software packages should be installed in the order presented before you attempt to build RAT. Take note of the version numbers as many of these packages make incompatible changes between releases.

- Python 2.7.X with development headers -
  - on Scientific Linux – run the following commands in [InstallDir] where [InstallDir] is the location you are installing python:

```
# Python 2.7.6:
wget http://python.org/ftp/python/2.7.6/Python-2.7.6.tar.xz
tar xf Python-2.7.6.tar.xz
cd Python-2.7.6
./configure --prefix=[InstallDir]/python --enable-unicode=ucs4 --enable-
↪shared LDFLAGS="-Wl,-rpath=[InstallDir]/python/lib"
make && make altinstall

#you will need to add [InstallDir]/python/Python-2.7.6 to the PATH, this can␣
↪be done on the command line, or in your .bashrc file
PATH=[InstallDir]/python/Python-2.7.6:$PATH

#you will need to add [InstallDir]/python/lib to the LD_LIBRARY_PATH, this␣
↪can be done on the command line, or in your .bashrc file
LD_LIBRARY_PATH=[InstallDir]/python/lib:$LD_LIBRARY_PATH
```

NOTE: The PATH and LD_LIBRARY_PATH will need to be set in this manner whenever you are using rat-pac

- on Ubuntu: ... ... `sudo apt-get install python-dev`
- on SUSE Linux: ... `sudo zypper install python-devel`

- **ROOT 5.34 - Used for object serialization and network processors. (other versions of ROOT 5 are okay too). RAT-PAC re**

  - `./configure --enable-python --enable-minuit2`

- GEANT4 10.01.p02 tar file- Toolkit used by the Monte Carlo simulation. When running `cmake` to configure GEANT4, be sure to use `-DGEANT4_INSTALL_DATA=ON` to download the interaction cross-section files (or download them manually).

  For begining GEANT4 users

  In the directory you want to install Geant4 (referenced as `[InstallDir]` below), type the following commands:

```
DIR=$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )
wget http://geant4.cern.ch/support/source/geant4.10.01.p02.tar.gz
tar -zxvf geant4.10.01.p02.tar.gz
mkdir $DIR/geant4.10.01.p02-build
cd geant4.10.01.p02-build
cmake -DCMAKE_INSTALL_PREFIX=$DIR/geant4.10.01.p02-build $DIR/geant4.10.
↪01.p02 -DGEANT4_USE_SYSTEM_EXPAT=OFF -DGEANT4_INSTALL_DATA=ON -DGEANT4_
↪BUILD_MULTITHREADED=ON -DGEANT4_USE_QT=ON
make -j1
make install
cd ..
mkdir envSetupScripts
```

  The geant4 enviroment variable files referenced below will be located in [InstallDir]/geant4.10.01.p02-build/InstallTreeFiles/

- SCons - Using common package managers, type
  - on Scientific Linux: `sudo yum install scons`
  - on Ubuntu: ... ... `sudo apt-get install scons`

1.2.2 Build Steps

- Make sure that you have setup your environment variables:

  Source the GEANT4 enviroment variable files

  - for bash shell `geant4.sh`
  - for C shell `geant4.csh`

  Source the ROOT enviroment variable files

  - for bash shell `thisroot.sh`
  - for C shell `thisroot.csh`

  If you do not know where to find these files, type

  - `locate [fileName]`

  Replacing [fileName] with the file you wish to locate. If the file path does not appear, type

  - `sudo updatedb` # `sudo updatedb` may require an administrator to run

- **Download RAT from GitHub** Move to the directory you wish to install rat-pac in [rat-pacWorkingDIR] hinceforth. For Anonymous Users type

  - `git clone https://github.com/rat-pac/rat-pac`

  For Developers type

  - `git clone git@github.com:rat-pac/rat-pac`

- Build rat-pac Starting from [rat-pacWorkingDIR] type
  - `cd rat-pac`
  - `./configure`

  Source the newly generated rat-pac enviroment variable files :Note: You will also need to source this file in the future befo

  - for bash shell type `env.sh`
  - for C shell type `env.csh`

  Build the rat-pac development package by typing

  - `scons`

1.2.3 Test Drive

- **Run an interactive session by typing**

  - `rat`

    Here's a sample of what you might see (type `exit` to exit the interactive rat-pac terminal):

```
RAT, version c8164f1
Status messages enabled: info
Hostname: wrangler.phys.vt.edu PID: 31590
DB: Loading /rat-pac-working-dir/rat-pac/data/DAQ.ratdb ... Success!
DB: Loading /rat-pac-working-dir/rat-pac/data/DETECTOR.ratdb ... Success!
DB: Loading /rat-pac-working-dir/rat-pac/data/ELEMENTS.ratdb ... Success!
DB: Loading /rat-pac-working-dir/rat-pac/data/IBD.ratdb ... Success!
```

(continues on next page)

(continued from previous page)

```
DB: Loading /rat-pac-working-dir/rat-pac/data/IO.ratdb ... Success!
DB: Loading /rat-pac-working-dir/rat-pac/data/MATERIALS.ratdb ... Success!
DB: Loading /rat-pac-working-dir/rat-pac/data/MC.ratdb ... Success!
DB: Loading /rat-pac-working-dir/rat-pac/data/NOISE.ratdb ... Success!
DB: Loading /rat-pac-working-dir/rat-pac/data/NTUPLE.ratdb ... Success!
DB: Loading /rat-pac-working-dir/rat-pac/data/OPTICS.ratdb ... Success!
DB: Loading /rat-pac-working-dir/rat-pac/data/PMT.ratdb ... Success!
DB: Loading /rat-pac-working-dir/rat-pac/data/PMTCHARGE.ratdb ... Success!
DB: Loading /rat-pac-working-dir/rat-pac/data/PMTTRANSIT.ratdb ...␣
↪Success!
DB: Loading /rat-pac-working-dir/rat-pac/data/RUN.ratdb ... Success!
DB: Loading /rat-pac-working-dir/rat-pac/data/SPECTRUM.ratdb ... Success!
DB: Loading /rat-pac-working-dir/rat/rat/data/PMTTRANSIT.ratdb ...␣
↪Success!

*************************************************************
 Geant4 version Name: geant4-08-01-patch-01    (27-July-2006)
                      Copyright : Geant4 Collaboration
                      Reference : NIM A 506 (2003), 250-303
                            WWW : http://cern.ch/geant4
*************************************************************

Visualization Manager instantiating...
Visualization Manager initialising...
Registering graphics systems...
Visualization Manager initialising...
Registering graphics systems...

You have successfully registered the following graphics systems.
Current available graphics systems are:
  ASCIITree (ATree)
  DAWNFILE (DAWNFILE)
  G4HepRep (HepRepXML)
  G4HepRepFile (HepRepFile)
  OpenGLImmediateQt (OGLIQt)
  OpenGLStoredQt (OGLSQt)
  RayTracer (RayTracer)
  VRML1FILE (VRML1FILE)
  VRML2FILE (VRML2FILE)

Registering model factories...

You have successfully registered the following model factories.
Registered model factories:
  generic
  drawByCharge
  drawByParticleID
  drawByOriginVolume
  drawByAttribute

Registered filter factories:
  chargeFilter
  particleFilter
  originVolumeFilter
  attributeFilter

You have successfully registered the following user vis actions.
```

(continues on next page)

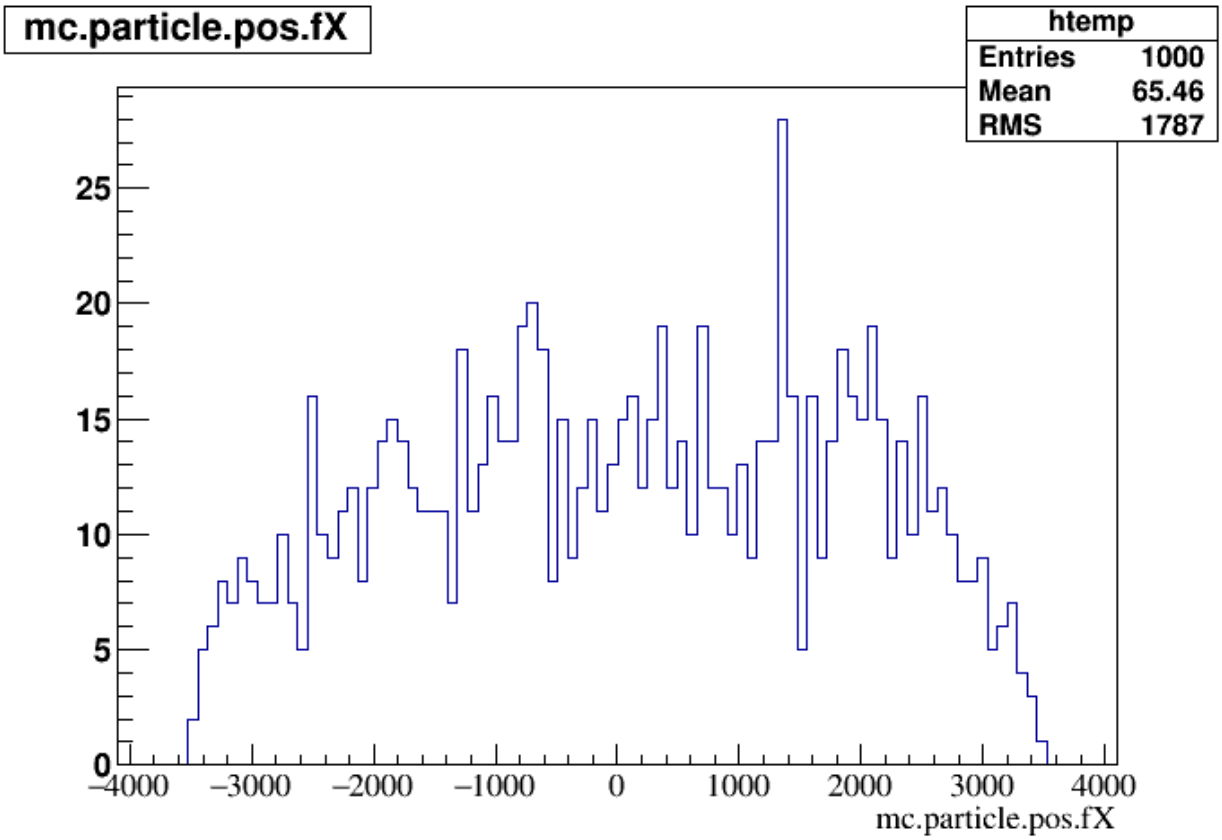
(continued from previous page)

```
Run Duration User Vis Actions: none
End of Event User Vis Actions: none
End of Run User Vis Actions: none

Some /vis commands (optionally) take a string to specify colour.
Available colours:
  black, blue, brown, cyan, gray, green, grey, magenta, red, white, yellow

Available UI session types: [ Qt, GAG, tcsh, csh ]
***** Can not open a macro file <prerun.mac>
PreInit>
```

- Run a macro example job by typing
  - `rat mac/electron_demo_cylinder.mac -o test.root`

  This will simulate 1000 – 10 MeV electrons in a cylindrical detector.
- Now you can start ROOT to analyze the events you just created by typing
  - `root test.root`
  - `T->Draw("mc.particle.pos.fX")`

You should get a plot of particle x coordinates similar to the plot below.



Note that with the RAT environment sourced, you are getting a special copy of ROOT that automatically loads the RAT ROOT event library.

report errors to Javier Caravaca <jcaravaca@berkeley.edu> or Nuno Barros <nfbarros@hep.upenn.edu> page update Sept. 9 2015

1.3 Tutorials

- Tutorial 1

1.4 Command Interface and Macro Files

1.4.1 Issuing Commands to RAT

Interactive

RAT is controlled through a text-command interface. If RAT is executed without any command-line options, it will start in interactive mode. The interactive shell has a command history and understands most of the keyboard navigation shortcuts available in tcsh. If you would like RAT to execute some commands before starting the interactive session, you can place those commands into a file called prerun.mac in the current directory. The interactive session can be terminated by typing exit.

Macro Files

If RAT is started with one or more files listed on the command line:

```
rat macro1.mac macro2.mac macro3.mac
```

then RAT will start in batch mode and execute each command found in the macro files sequentially. When all commands have been executed, RAT will terminate. If one of these macro filenames is "-", then RAT will start an interactive session when it reaches that point in the file list. Note that prerun.mac will not be loaded in this case.

Command Syntax

RAT uses the GEANT4 command interface, so the syntax of commands is identical to any other GEANT4 application. Comments are preceded by # and blank lines are ignored. Commands themselves consist of the command name followed by zero or more parameters, separated by spaces. For example, the command:

```
/rat/procset update 5
```

starts with the command name `/rat/procset` followed by two parameters, "update" and the number 5. Whitespace is generally not important, but commands may not have leading whitespace. This means you cannot indent commands to aid in readability, for example.

Commands are organized into a filesystem-like hierarchy of directories and sub-directories. (PAW users should find this very familiar.) RAT automatically contains all the standard GEANT4 commands, as well as any additional commands defined by GLG4sim. This allows RAT to execute GLG4sim macro files, and the goal is to preserve this compatibility as long as it is feasible. All RAT-specific commands are confined to the /rat/ command subdirectory.

1.4.2 Tips for Macro Files

Layout

The order of commands in a macro file can be important, so it is best to follow this convention for organizing your macro files:

```
# example.mac - Example Macro File
# author: Bill Bobb

# --------------------------------

# Set parameters here

# --------------------------------

/run/initialize

# BEGIN EVENT LOOP

# Issue /rat/proc commands here to add processors to the event loop

# END EVENT LOOP

# --------------------------------

# Run commands to start appropriate event generator (Gsim, InROOT, etc.)
```

Inclusion

You can include one macro file in another using the /control/execute command. For example:

```
/control/execute setup.mac
```

The included macro is immediately read and executed in place.

1.5 Event Producers vs. Event Processors

RAT distinguishes "producers" from "processors" during computation. An event producer is an object which creates new events in memory, either out of nowhere (as in the Gsim Monte Carlo producer) or from some other external source like a ROOT file. The producer is responsible for allocating the memory, controlling the loop over events, and calling the event loop for each event it creates.

An event processor is part of the event loop. It does not create new events, but instead receives events one-by-one and may either change the event by adding to or altering its contents, or it may simply passively observe the event.

The only reason you need to worry about this is because they way you interact with these two entities in your macro file is very different. Producers are ''executed immediately," whereas processors are ''declared." When the macro file reader gets to a line like:

```
/rat/proc count
```

it creates a new instance of the [wiki:UserGuideCount Count] processor in memory and adds it to the end of the global event loop, but nothing else happens. No computation has occurred, and no events are generated.

However, when you get to a line that invokes a producer, such as this one which starts Gsim:

```
/run/beamOn 100
```

RAT immediate begins to simulate 100 events, and each one is passed to the event loop that has been declared thus far. Execution of your macro file will not continue until those 100 events have been generated and processed.

This is why, as described in the [wiki:UserGuideCommandInterface Command Interface] section, you must first create your processors before calling the event producer, which procedurally looks backwards:

```
# Event loop
/rat/proc count
/rat/procset update 5
/rat/proc fitcentroid
/rat/proc outroot
/rat/procset file "fit.root"


# Run RATGsim
/generator/rates 3 1
/generator/gun gamma 0 0 0  0 0 0  1.022
/run/beamOn 10
```

You can also call several event producers sequentially in the same macro file if you like. For example, you could generate events at three energies this way:

```
/generator/rates 3 1
/generator/gun gamma 0 0 0  0 0 0  1.022
/run/beamOn 10
/generator/gun gamma 0 0 0  0 0 0  2.461
/run/beamOn 10
/generator/gun gamma 0 0 0  0 0 0  8.600
/run/beamOn 10
```

or even mix different event generators, though that probably isn't very useful.

1.6 Random Numbers

All of RAT uses random numbers generated by the !HepJamesRandom class from CLHEP. This class implements the random number algorithm from ''F.James, Comp. Phys. Comm. 60 (1990) 329'', which was also used in the FORTRAN library MATHLIB.

Normally, the random number generator is initialized at startup using a seed which is a mixture of the current time and the process ID of RAT. With this seeding scheme, two RAT instances should generate different event sequences, even if they are started at the same time on the same machine or different machines. This assurance is probabilistic however, and not an absolute guarantee.

The seed is always written to the log file:

```
This is RAT, version 0.1
Status messages enabled: info detail
Seeding random number generator: 937308832
```

If desired, the seed can be selected at the command line using the -s switch. For example, a seed of zero can be forced using:

```
rat -s0 mac/std_test.mac
```

This is useful when debugging to reproduce an error caused by a particular event sequence. Any long integer may be used as the seed.

1.7 Data Structure

The event data structure is a tree of information about a particular event. Event producers (like [wiki:UserGuideGsim Gsim]) create an instance of the the data structure for each event, and processors can then operate on this structure, transforming it as desired.

A point of clarification: Unfortunately, "event" is an ambiguous term. There is what you might call a ''physics event," which represents a single interaction, like nu-bar + p -> e+ + n. The simulated detector, however, will probably turn that into at least two ''detector events." Unless specified, assume "event" means "physics event."

1.7.1 Data Structure Definition

Brackets in the name indicate that item is an array of the listed type. The brackets are not valid ROOT syntax, so you should drop them when plotting, as in:

```
T->Draw("mc.particle.x");
```

1.7.2 General Information

name	type	description
lastProcResult	int	Result code of last processor, OK = 0, FAIL=1, ABORT=2
ratversion	int	Version of RAT which produced this event, ver 0.1 = 100, ver 1.2 = 1200, and so on

1.7.3 Monte Carlo Information

name	type	description
mc	MC	Contains all Monte Carlo information
mc.runID	int	Run number
mc.eventID	int	Event number
mc.UT	double	Time of this event (units, origin?)

Monte Carlo Particles

Monte Carlo PMT Information

Monte Carlo Track Information

1.7.4 Detected Event Information

Event Reconstruction

1.8 The RAT Database (RATDB)

RATDB is the database of constants that will be used by RAT for all adjustable parameters. "Parameters" in this sense includes

- physical properties of materials (density, optics, etc.)
- geometric configuration of parts of the detector (dimensions and locations of the acrylic vessel, PMTs, . . .)
- calibration constants
- lookup tables
- control parameters which specify how processing should be done

Note that RATDB is NOT intended to hold actual event information, just everything else.

Despite the suggestive name, RATDB is not really a proper Database, like MySQL or Oracle. Rather it is just a simple set of C++ classes used to read parameters from various sources (more on that later) and make the data easily available to other parts of the RAT application.

1.8.1 How is the data organized?

Data in RATDB is organized in a two-level space. At the top level are ''tables'', which contain groups of ''fields''. So, every item in the RATDB is located by these two things.

Tables

Tables are identified by names written in all capital letters which follow the C++ rules for identifiers. Only letters, numbers, and the underscore are allowed, and the name must start with a non-digit. Tables also carry an index, which also follows identifier rules (mix case allowed). The convention when writing out table names is to put the index in brackets right after the name, with the exception that it may be left off if the index is "" (the empty string).

Some examples of valid table names are:

```
CALIB, DAQ_CHANNELS, THETA13, MEDIA[vacuum], MEDIA[water], MEDIA[acrylic]
```

The index is intended to allow several tables with the same fields to exist. For example, you might standardize a MEDIA table that holds properties for all the materials in the detector. All materials will have the same set of properties (density, index of refraction, etc), and each material will be given a different index. Many tables will not need this functionality, so you are free to ignore the index when dealing with them, and the index will be implicitly understood to be empty by RATDB.

Fields

Every table contains zero or more fields. Similar to table names, field names also follow the C++ identifier convention, but use only lowercase letters. Each field has associated with it a piece of data of a distinct type. The currently allowed types are: integers, floats, doubles, strings, integer arrays, float arrays, double arrays, and string arrays.

Arrays will only contain elements of the same type, and there must be at least one element in an array. The length of the array, however, is not specified and is allowed to change.

Planes

Normally, tables and fields are all you have to think about, but RATDB also addresses an additional complication: overriding constants. It is a common use case to have default values of constants, values which are only valid in certain time intervals and can change (like the optical properties of the scintilator), and user-specified values which are intended to override everything.

RATDB handles this by internally grouping tables into three ''planes''. The name is intended to suggest a stack of layers where you start at the top, and keep going down until you find your answer. The RATDB planes are (from highest to lowest priority):

- the user plane
- the time plane
- the default plane

When an item is requested, RATDB will attempt to locate it in the user plane first, then the time plane, and finally the default plane. Note that this is all handled in the background. You simply request the index_of_refraction field in the MEDIA[acrylic] table, and RATDB figures out the appropriate plane from which to retreive the data.

1.8.2 How do I load data into RATDB?

RATDB has the potential to read data from a variety of sources (such as real SQL databases), but right now only supports reading data in the RATDB text format. Read the [wiki:RATDB_TextFormat RATDB text format] page for instructions on how to compose such a file.

Once you have your text file, you have two options for loading it:

- Give it the .ratdb extension and place it into the $GLG4DATA directory. This is usually the same as $RATROOT/data. All .ratdb files in that directory are automatically loaded when RAT first starts.
- Manually load the file in your macro using a command like:

```
/rat/db/load myfile.ratdb
```

You can also set the value of individual fields in the database inside the your macro using commands like:

```
/rat/db/set MEDIA[acrylic] index_of_refraction 1.52
/rat/db/set GEO_DETECTOR av_radius 6.2
```

The /rat/db/set command alters fields in the user plane, so they will override any other values set on the time or default planes.

Note that these changes are not saved when you exit RAT. If you want permanently change the value of a field, you need to edit the relevant .ratdb file in the data/ directory. Also note that /rat/db/set will create new tables in memory if they do not already exist.

1.8.3 Limitations

RATDB is not finished by any means, but is complete enough to address immediate needs. Current limitations include:

- Tables cannot be added to the time plane. Properly implementing the time plane will be non-trivial and require implementing caching and load-on-demand semantics for tables. At the moment, only the user and default planes exist, but adding the time plane will not require any changes to user code or macros. We don't need it now, so there is no sense in investing the time to implement it yet.
- Array fields cannot be altered using the /rat/db/set command.

1.9 Gsim Generators

Gsim creates the initial particles simulated in the event using ''generators'', which are enabled in the macro file. A generator decides how often a particular kind of event occurs, where it happens, and what kind of particles and energies the event starts with. Multiple generators can be used at once, and different events will be interleaved according to their rates, or even pile up if they occur in coincidence.

Generators are activated in the macro file using the command:

```
/generator/add gen_name  generator_options
```

The first parameter, ''gen_name'', identifies the kind of event generator being added. For example, the ''combo" generators allows you to piece together separate vertex, position, and time generators. The second parameter, ''generator_options" is a string which is passed to the generator itself to configure it. For example:

```
/generator/add combo gun:point:poisson
```

adds a new combo generator to the simulation which will be comprised of a particle gun with events filling a detector volume and poisson-distributed random times.

Once a generator has been added, you can configure the vertex, position, and time components. For example, we can generate isotropic positrons in the center with 1 MeV of kinetic energy at a mean rate of 1 per second with the commands:

```
/generator/vtx/set 0 0 0  1.0
/generator/pos/set 0 0 0
/generator/rate/set 1.0
```

1.9.1 Top-level generators

Top-level generators are understood by the /generator/add command.

combo

```
/generator/add combo VERTEX:POSITION:TIME
```

or

```
/generator/add combo VERTEX:POSITION
```

Creates a new combo generator using the vertex, position, and time generators described below. If the variant without a TIME parameter is used, it implies the "poisson" time generator.

decaychain

```
/generator/add decaychain ISOTOPE:POSITION:TIME
```

or

```
/generator/add decaychain ISOTOPE:POSITION
```

Creates a new decaychain generator using the position, and time generators described below. If the variant without a TIME parameter is used, it implies the "poisson" time generator.

The ISOTOPE parameter can be any chain or element found in the file data/beta_decay.dat. The alpha, beta, and gamma particles emitted by the radioactive decay chain will be included in the event, with times and kinetic energies randomly generated according to the physics of the decay. In the current implementation, the times are set such the final decay occurs at t=0, so that earlier decays are at negative times.

Californium source

```
/generator/add cf 252:POSITION:TIME
```

or

```
/generator/add cf 252:POSITION
```

Creates a new Cf252 generator using the position, and time generators described below. If the variant without a TIME parameter is used, it implies the "poisson" time generator.

The syntax of the command may lead you to think that other isotopes of Cf252 (e.g., Cf255) are supported. One day that may happen, but right now only the value 252 can occur in the command, otherwise you'll get an error message.

This generator models the products of the spontaneous fission of Cf252: neutrons and prompt photons. It does *not* model the radioactive decay of Cf252; neither does the ''decaychain" generator above (though that could be added by revising the data/beta_decay.dat file). Note: Geant4 models the fission of nuclei due to de-excitation from radioactive decays, and has its own implementation of radioactive decay chains, but it does not include a model for the spontaneous fission of nuclei; the only way to include that is by writing a separate event generator.

external

This top-level generator creates one or more particles with type, momentum, and optional time offset, spatial offset, and polarization, based on lines read from a text file. Synax:

```
/generator/add external VERTEX:TIME:INPUT
```

VERTEX is either 'external' or a vertex generator. If 'external' is specified, vertex positions in INPUT are used.

TIME is a time generator.

INPUT is either a file or a program which outputs text in the format described below(*). INPUT should be surrounded by quotes.

Examples:

```
/generator/add external external:poisson:"muons.dat"
```

Generates events listed in muons.dat using vertex locations specified in that file and times generated by the poisson time generator.

```
/generator/add external point:poisson:"/some/path/generateMuons 10 |"
/generator/pos/set 0 0 0
```

Calls the program /some/path/generateMuons and generates the events output by this program with position 0 0 0.

The format of the text input is:

```
NHEP
 ISTHEP IDHEP JDAHEP1 JDAHEP2 PHEP1 PHEP2 PHEP3 PHEP5 DT X Y Z PLX PLY PLZ
 ISTHEP IDHEP JDAHEP1 JDAHEP2 PHEP1 PHEP2 PHEP3 PHEP5 DT X Y Z PLX PLY PLZ
 ISTHEP IDHEP JDAHEP1 JDAHEP2 PHEP1 PHEP2 PHEP3 PHEP5 DT X Y Z PLX PLY PLZ
 ... [NHEP times]
NHEP
 ISTHEP IDHEP JDAHEP1 JDAHEP2 PHEP1 PHEP2 PHEP3 PHEP5 DT X Y Z PLX PLY PLZ
 ISTHEP IDHEP JDAHEP1 JDAHEP2 PHEP1 PHEP2 PHEP3 PHEP5 DT X Y Z PLX PLY PLZ
 ISTHEP IDHEP JDAHEP1 JDAHEP2 PHEP1 PHEP2 PHEP3 PHEP5 DT X Y Z PLX PLY PLZ
 ... [NHEP times]
```

where:

```
ISTHEP   == status code
IDHEP    == HEP PDG code
JDAHEP   == first daughter
JDAHEP   == last daughter
PHEP1    == px in GeV
PHEP2    == py in GeV
PHEP3    == pz in GeV
PHEP5    == mass in GeV
DT       == vertex _delta_ time, in ns (**)
X        == x vertex in mm
Y        == y vertex in mm
Z        == z vertex in mm
PLX      == x polarization
PLY      == y polarization
PLZ      == z polarization
```

PHEP5, DT, X, Y, Z, PLX, PLY, and PLZ are all optional. If omitted, the respective quantity is left unchanged. If DT is specified, the time offset of this and subsequent vertices is increased by DT: (**) note DT is a relative shift from the previous line. (This is because there is often a very large dynamic range of time offsets in certain types of events, e.g., in radioactive decay chains, and this convention allows such events to be represented using a reasonable number of significant digits.) If X, Y, Z, PLX, PLY, and/or PLZ is specified, then the values replace any previously specified.

vertexfile

The VertexFile generator is used to take event vertices generated by RAT or any other program (for example GENIE), and simulate them in RAT. Unlike the InROOT event producer which only runs RAT processors on fully simulated events, this generator starts from just the MCParticle and MCParent information and runs the full Geant4 simulation as well.

The syntax is

```
/generator/add vertexfile filename[:POSITION][:TIME][:NEVENTS][:NOFFSET]
```

where filename points to any RAT root file that at least has MC particle information. POSITION and TIME arguments can be used to override the position and time of vertices in the file by giving any position or time generator listed below instead. If they are set to 'default', the positions / times given in the root file will be used. It will simulate NEVENTS

of the events in the file, starting with event NOFFSET. If NEVENTS is less than the total number of events to simulate set by /run/beamOn, the simulation will exit after NEVENTs.

1.9.2 Vertex generators

Vertex generators select the initial particle types, number, momenta, and polarization.

gun

```
/generator/vtx/set  pname  px  py  pz  [ke]  [polx poly polz]
```

Single particle gun. Creates a particle identified by pname with initial momentum (px, py, pz) given in MeV/c. The optional parameter ke sets the kinetic energy of the particle in MeV and overrides the magnitude of the momentum vector. (If you use ke, you can treat px, py, and pz as just a direction vector.) The optional polarization vector of the particle is given by (polx, poly, polz).

If px=py=pz=0, then the gun generates particles with isotropic initial directions. Similarly, if polx=poly=polz=0, or the polarization vector is left out, the particles will be randomly polarized.

Valid particle names include:

```
 GenericIon,                He3,              alpha,         anti_kaon0
anti_lambda,       anti_neutron,          anti_nu_e,         anti_nu_mu
anti_omega-,        anti_proton,        anti_sigma+,        anti_sigma-
anti_sigma0,           anti_xi-,           anti_xi0,    chargedgeantino
   deuteron,                 e+,                 e-,               eta
  eta_prime,              gamma,           geantino,              kaon+
      kaon-,              kaon0,             kaon0L,             kaon0S
     lambda,                mu+,                mu-,            neutron
       nu_e,             nu_mu,             omega-,      opticalphoton
        pi+,               pi-,                pi0,             proton
     sigma+,             sigma-,             sigma0,             triton
        xi-,               xi0,
```

(This list comes from the /particle/list command.)

gun2

```
/generator/vtx/set  pname  px  py  pz angle E1 E2 [polx poly polz]
```

Modification of gun, the single particle gun. Creates a particle identified by pname (as above) with initial momentum (px, py, pz) given in arbitrary units for pointing. The angle parameter sets the opening angle of a 'cone of fire' such that angle = 90 fires particles evenly into the hemisphere along the [px,py,pz] direction. Setting angle to 0 gives the same behavior as gun.

E1 and E2 determine the range of particle kinetic energies in MeV. Setting E1 and E2 the same results in the same behavior as gun. If E2 != E1 the particle energy is randomly drawn from a flat distribution between E1 and E2.

The optional polarization vector of the particle is given by (polx, poly, polz).

If px=py=pz=0, then the gun generates particles with isotropic initial directions. Similarly, if polx=poly=polz=0, or the polarization vector is left out, the particles will be randomly polarized.

ibd

```
/generator/vtx/set  dir_x dir_y dir_z
```

Inverse beta decay events caused by the interaction of a neutrino with a stationary proton. The event is initialized with the products of the reaction, a positron and a free neutron. The initial direction of the neutrino is along the (dir_x, dir_y, dir_z) vector. The neutrino energy is drawn from the spectrum given in the [wiki:RATDB_IBD IBD table], and the positron direction distribution is weighted by the differential cross section of the interaction.

reacibd ‘’‘

```
/generator/vtx/set dir_x dir_y dir_z
```

Inverse beta decay events caused by the interaction of a reactor neutrino with a stationary proton. The initial energy of the neutrino for each event is selected from a probability density function dependent on the total neutrino flux from a reactor and the inverse beta-decay cross-section. The initial direction of the neutrino is along the (dir_x, dir_y, dir_z) vector. The positron direction is currently randomized relative to the neutrino’s incident direction.

The relative isotopic abundances of U235, U238, Pu239, and Pu241 can be controlled using macro commands:

```
/generator/reacibd/U235 U235Amp  #Default is 0.496
/generator/reacibd/U238 U238Amp  #Default is 0.087
/generator/reacibd/Pu239 Pu239Amp  #Default is 0.391
/generator/reacibd/Pu241 Pu241Amp  #Default is 0.066
```

The abundances should be provided for all four isotopes and should add to 1. The addition of other isotopes and elements is currently not supported.

es

```
/generator/vtx/set  dir_x dir_y dir_z
```

Elastic-scattering events caused by the interaction of a neutrino with an electron. The event is initialized with the product of the reaction, an electron. The initial direction of the neutrino is along the (dir_x, dir_y, dir_z) vector. The neutrino energy is drawn from the spectrum given in the [wiki:RATDB_IBD IBD table], and the electron direction distribution is weighted by the differential cross section of the interaction.

Note that the flux for elastic scattering is taken from the [wiki:RATDB_IBD IBD table] values; that is, it’s the same neutrinos that cause both types of events.

There are two parameters that control the elastic-scattering cross-section that can be controlled by macro commands:

```
/generator/es/wma  sin_squared_theta
```

This command sets the value of sine-squared of the weak mixing angle; the default is 0.2277.

```
/generator/es/vmu  neutrino_magnetic_moment
```

This command sets the value of the neutrino magnetic moment (units are Bohr magnetons); the default is 0.

pbomb

Generate a photon bomb, i.e. an isotropic distribution of photons, of a given number and wavelength.

Example:

```
/generator/vtx/set 1000 385
```

Produces events where each contains 1000 photons, each with a wavelength of 385 nanometers.

spectrum

Generates particles with isotropic momentum and kinetic energy drawn from a user-defined spectrum stored in a SPECTRUM table in RATDB. The spectrum is linearly interpolated between points, which do not have to be uniformly spaced.

Example:

```
/generator/vtx/set e- flat
```

Produces electron events drawn from the spectrum stored in the SPECTRUM[flat] table:

```
{
name: "SPECTRUM",
index: "flat",
valid_begin: [0, 0],
valid_end: [0, 0],

// default spectrum is flat
spec_e:     [ 1.00, 1.50, 2.00, 2.50, 3.00, 3.50, 4.00, 4.50, 5.00], // (MeV)
// (Note that first point is minimum of spectrum, last is maximum)
spec_mag:   [ 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00], // don't worry␣
↪about normalisation
}
```

1.9.3 Position generators

Position generators select points in space for the initial particles.

point

```
/generator/pos/set x y z
```

Events at a single point. Coordinates x, y, z are in mm.

fill

```
/generator/pos/set x y z
```

Events uniformly fill a physical volume containing the point (x, y, z) (mm).

regexfill

```
/generator/pos/set volume_pattern
```

Events uniformly fill all physical volumes with names matching the POSIX regular expression (regex) given as `volume_pattern`. In general volume names correspond to the index of `GEO` table entries, but complex geometry factories may generate other volumes as sub components, many volumes for arrays, or both.

For a concrete example, this can be used to generate events in the wall (glass) of all PMTs built with a `pmtarray` type geometry factory. If the index of the `GEO` table for the `pmtarray` was `inner_pmts` the PMTID number of the physical PMT would be appended to the volume name as it is created, so an appropriate regex would be `inner_pmts[0-9]+`

```
/generator/pos/set inner_pmts[0-9]+
```

Note that the volume name is considered a match if the regex matches any part of the volume name, e.g. the regex `mts1` would match the volume name `inner_pmts100`. This can be avoided by using start `^` and end `$` of line characters when specifying a unique `^volume$` by name.

fillshell

```
/generator/pos/set X Y Z Ri Ro volname
```

Events uniformly fill a shell centered at (X, Y, Z) (mm) with inner radius Ri and outer radius and are contained only in the volume named "volname."

Note that the old syntax (old as of r1188) still works, for backwards compatibility. The old syntax is:

```
/generator/pos/set Vx Vy Vz X Y Z Ro Ri
```

where points are contained only within the same volume as the point (Vx, Vy, Vz).

paint

```
/generator/pos/set x y z
```

Events a distributed uniformly over the surface of the logical volume containing the point (x, y, z) (mm).

multipoint

```
/generator/pos/set number_of_locations inner_radius outer_radius
```

Generates events at different locations in the detector between two radii. For a given value of number_of_locations, the points are unique and fixed for all runs on all platforms. The generator will cycle between the different points as the event number increments, so the number of events you generate in each job should be a multiple of number_of_locations. This generator is typically used to benchmark reconstruction, as you can fit events at each generated location to compute a bias and resolution.

1.9.4 Time generators

Time generators control the interval between events for a given generator.

uniform

```
/generator/rate/set evrate
```

Time between events is exactly 1/evrate (seconds).

poisson

```
/generator/rate/set evrate
```

The event rate is a poisson distribtion with mean of evrate (1/seconds).

1.10 Gsim Geometry

The detector geometry used in Gsim is controlled by the GEO tables stored in RATDB. Each piece of the detector is represented by a GEO table that gives the name of the element, the shape, position, material, and color (for visualization purposes). This allows simple changes to the detector configuration to be made without having to edit the RAT source code.

Unlike other RATDB files, geometry files end in .geo rather than .ratdb. Because of this, when RAT starts, no GEO tables are loaded into memory. Instead, the program waits until the /run/initialize command is issued, and then loads the geometry file listed in the DETECTOR.geo_file field. It also optionally loads the geometry file given in the DETECTOR.veto_file field if it exists. Once these files have been loaded into the database, the detector is constructed from all of the GEO tables currently in memory.

1.10.1 Customizing the Geometry in a Macro File

There are several ways to customize the detector geometry:

Change DETECTOR.geo_file

If you want to make drastic changes to the detector, you should start by copying one of the geometry files (like simple.geo) and editing it. Then you can add a command like:

```
/rat/db/set DETECTOR geo_file "mydetector.geo"
```

to the top of your macro (before /run/initialize). None of the default geometry will be loaded, just your new file. Note that this file should be placed in the data directory.

Load an additional geometry file

Since geometry files are just RATDB files, you can load additional GEO tables using a command like:

```
/rat/db/load "calibration_source.geo"
```

This file can contain either completely new detector pieces, or it can override parts of the default geometry. Make sure you set the validity ranges on the tables to put them in the user plane, otherwise they will be overwritten by the defaults when they are loaded. Any extra GEO tables you create will be built right along with the defaults when /run/initialize is executed.

Alter individual fields

For a very small change, like changing just a few numbers, you can use the commands:

```
/rat/db/set GEO[av] r_max 2800.0
/rat/db/set GEO[scint] r_max 2700.0
```

Unfortunately, you cannot change array fields using the set command yet.

1.10.2 GEO Table Fields

GEO tables can contain a wide variety of fields to control the properties of the volume. The common fields shared by all tables:

Field	Type	Description
`index`	`string`	Name of the volume. To conform with RATDB standards, it should follow identifier conventions (no spaces).
`mother`	`string`	Name of the mother volume. The mother volume should fully contain this volume. The world volume has the mother “”.
`enable`	`int` (optional)	If set to zero, this volume is skipped and not constructed.
`type`	`string`	Shape of this volume, see below for list.
`sensitive_detector`	`string` (optional)	Name of sensitive detector if this volume should register hits. Limited to ‘’/mydet/pmt/inner” and ‘’/mydet/veto/genericchamber’‘

Allowed types:

- box - Rectangular solid
- tube - Cylindrical solid (or section over limited phi range)
- ptube - Cylindrical solid with circular perforations along z cut out
- sphere - Spherical solid (or section over limited theta and phi range)
- psphere - Spherical solid with circular perforations radially cut out
- revolve - Solid of revolution defined by (r_max, r_min, z)
- AV - Generic spherical acrylic vessel
- tubearray - Array of tubes
- lgarray - Array of tubes where one end has the PMT face cut out
- pmtarray - Array of PMTs
- waterboxarray - Array of standard cubitainer water boxes
- extpolyarray - Array of extruded polygonal solids
- bubble - Collection of bubbles

All types except “pmtarray”, “waterboxarray”, and “bubble” have these additional fields:

Field	Type	Description
material	string	Material filling this volume. See the MaterialList for details.
color	float[3|4] (optional)	Color to be used for this element in visualization. Either RGB or RGBA (A=alpha transparancy) components ranging from 0.0 to 1.0.
invisible	int	If set to 1, mark this volume as invisible during visualization
position	float[3] (optional)	X, Y, Z (mm) components of the position of the volume center, 'in coordinate system of the mother volume'. Default position is the center.
rotation	float[3] (optional)	X, Y, Z axis rotations (deg) of element about its center. Rotations are applied in X, Y, Z order. Default is no rotation.
replicas	int (optional)	Replicate this volume N times inside the mother (position and rotation are ignored if this is set)
replica_axis	string (optional)	Axis along which to replicate volume: x, y, z
replica_spacing	float (optional)	Distance (mm) between replicas

Box Fields:

Field	Type	Description
size	float[3]	X, Y, Z half-lengths (mm) of box (perpendicular distance from center to each face)

Tube Fields:

Field	Type	Description
r_max	float	Outer radius of tube (mm)
r_min	float (optional)	Inner radius of tube (mm) Default is 0.0 (solid)
size_z	float	Half-height of tube (mm)
phi_start	float (optional)	Angle (deg) where tube segment starts. Default is 0.0
phi_delta	float (optional)	Angle span (deg) of tube segment. Default is 360.0

Sphere Fields:

Field	Type	Description
r_max	float	Outer radius of sphere (mm)
r_min	float	Inner radius of sphere (mm) Default is 0.0 (solid)
theta_start	float (optional)	Polar angle (deg) where sphere segment starts. Default is 0.0
theta_delta	float (optional)	Polar angle span (deg) of sphere segment. Default is 180.0
phi_start	float (optional)	Azimuthal angle (deg) where sphere segment starts. Default is 0.0
phi_delta	float (optional)	Azimuthal angle span (deg) of sphere segment. Default is 360.0

PMTArray Fields:

Field	Type	Description
pmt_model	string	Serves as the index for PMT, PMTCHARGE, and PMTTRANSIT tables giving the geometry, charge response, and time response models.
pos_table	string	Specifies the PMTINFO table to use when placing these PMTs (see PMT Simulation)
start_idx	int (optional)	Index to start building PMTs in the PMTINFO table specified (inclusive, defaults to 0)
end_idx	int (optional)	Index to stop building PMTs in the PMTINFO table specified (inclusive, defaults to length-1)
orientation	string	Method of determining PMT direction. "point" will aim all PMTs at a point in space. "manual" requires that the position table also contain dir_x, dir_y, and dir_z fields which define the direction vector for each PMT.
orient_point	float[3] (optional)	Point (mm) in mother volume to aim all PMTs toward.
rescale_radius	float (optional)	Assumes all PMTs are spherically arranged around the center of the mother volume and rescales their positions to a particular radius. By default, no rescaling is done.

1.10.3 Creating a parameterized geometry

Using a DetectorFactory one can build a DB defined geometry on the fly (less useful), or modify a normal DB defined geometry template (more useful) before the geometry itself is built. Using only .geo files there is no nice way to have a property of a geometry component defined as a formula (a function of other geometry parameters), and no nice way to algorithmically define components of a scalable geometry, e.g. PMT positions for various photocathode coverage fractions.

The DetectorFactory to use is specified by name in the *DETECTOR* table under the field detector_factory and supersedes the geo_file field if used. If no DetectorFactory is specified, the geo_file specified is loaded as described above. A DetectorFactory should define tables in the DB in the same way a .geo file would and make use of GeoFactory components.

```
/rat/db/set DETECTOR experiment "Watchman"
/rat/db/set DETECTOR geo_file "Watchman/Watchman.geo"
```

v.s.

```
/rat/db/set DETECTOR experiment "Watchman"
/rat/db/set DETECTOR detector_factory "Watchman"
```

Example usage would be to load a normal (statically defined) .geo file into the DB and modify it as necessary for the dynamic functionality. See the WatchmanDetectorFactor for example use.

1.11 Physics Processes

The standard RAT simulation includes many standard GEANT4 physics processes, as well as some custom processes:

1.11.1 Scintillation

(''DISCLAIMER'': While the scintillation code in RAT is based on GLG4Sim by Glenn Horton-Smith, we have made several modifications to the code which change its behavior. Assume all bugs are ours!)

The scintillation simulation in RAT is handled differently than all other physics processes. In order to conserve energy on a step-by-step basis, scintillation photons are computed not as a standard GEANT4 physics process, but rather as a separate task after all other physics processes have run. The scintillation code can then look at the energy deposited during that completed step and calculate the number of scintillation photons that would be generated. A secondary task of the scintillation code is to handle reemission of photons in volumes which contain wavelength-shifter.

Code Structure

When the [source:rat/src/core/Gsim.cc Gsim::Init()] method is called, all of the GEANT4 user callbacks are established. One of these callbacks is the for a custom G4UserSteppingAction called [source:rat/src/core/GLG4SteppingAction GLG4SteppingAction]. At the end of each step, this class performs several tasks, among which is calling the static method [source:rat/src/core/GLG4Scint.cc GLG4Scint::GenericPostPostStepDoIt()]. GLG4Scint::!GenericPostPostStepDoIt() returns at G4VParticleChange object which contains the new secondary tracks (either scintillation photons or wavelength shifted photons) to be registered with the GEANT4 Stepping Manager.

In order to handle particle-specific scintillation parameters, a list of GLG4Scint objects are built by GLG4PhysicsList at startup, each responsible for a different particle. The static [source:rat/src/core/GLG4Scint.cc GLG4Scint::GenericPostPostStepDoIt()] method picks one of these objects based on the mass of the particle in the track. This list of particles is current limited to:

```
* (default)
* neutron
* alpha
* Ne20
* Ar39
* Ar40
```

If scintillation parameters are not specified for one of these particle types, the GLG4Scint object will load the default parameters instead. Once a suitable GLG4Scint object has been identified for the track, the GLG4Scint::!PostPostStepDoIt() method is called. The rest of this page describes what GLG4Scint::!PostPostStepDoIt() actually does.

Computing Number of Scintillation Photons

Normal particles (i.e. not optical photons) can deposit energy gradually in the medium through ionization and other processes. At the end of each track step, GLG4Scint determines the total deposited energy, ‘’dE’‘, and the step length, ‘’dx’‘. Then it applies Birk’s Law to compute the deposited energy after quenching:

$$dE_{\text{quench}} = \frac{dE}{1 + B \times dE/dx}$$

where ‘’B” is Birk’s Constant for your scintillator. If ‘’B” is set to zero, then Birk’s Law has no effect and the scintillator response is independent of ‘’dE/dx’‘.

An additional particle-dependent quenching factor, ‘’P(E)” can also be set which depends on the kinetic energy of the particle at the end of the step. This is useful if the scintillator quenching has been measured directly for a range of energies.

The deposited energy is converted to scintillation photons using the product of the light yield (‘’Y’‘) of the scintillator (which is in units of photons per MeV), the deposited energy, Birk’s Law scaling, the particle-dependent quenching, and a “reference ‘’dE/dx’‘” for Birk’s Law. The reference ‘’dE/dx” is useful if you have measured the light yield of the scintillator only with highly ionizing particles, like alphas, which already have a significant Birk’s Law component. The reference dE/dx effectively removes the quenching already in the light yield.

Finally, the mean number of photons can be scaled down by the "Photon Thinning" factor (''T'') selected by the user. Photon thinning is used to accelerate the simulation by reducing the number of optical photons produced by a constant factor, and then increasing the PMT photocathode efficiency by the same factor such that the product of light yield and detection efficiency is held constant.

Put together, the mean number of scintillation photons produced in the step is

$$N = Y \times dE \times \frac{1 + B \times dE/dx_{\text{ref}}}{1 + B \times dE/dx} \times P(E) \times T$$

Most of the factors in this equation are optional, and if not specified default to 1 for ''P(E)" and ''T" and 0 for ''B" and ''dE/dx_{ref}''.

The actual number of scintillation photons produced in the step is drawn from a Poisson distribution with mean N.

Scintillation Spectrum

Once the number of scintillation photons has been specified, the photon energy is drawn from a spectrum supplied for the material. The direction of each photon is randomly drawn from an isotropic distribution, and the polarization vector is randomly selected, but constrained to be orthogonal to the direction vector. The position of the photon is drawn from a uniform distribution along the line connecting the start and end points of the step.

Time Structure

The scintillation process has some time structure associated with it. The start time of a scintillation photon is the time the particle passed through the origin point of the photon, plus a delay drawn from the user-specified distribution. There are three possible options for the delay distribution:

1. A sampled time distribution, in the form of a list of (time, intensity) pairs.
2. A sum of decaying exponential distributions, each with an associated branching fraction and time constant.
3. A sum of two decaying exponential distributions, whose time constants are a function of particle energy.

The specification of delay distribution is described in the RATDB section below.

1.11.2 Wavelength Shifting

There are a few ways of doing bulk wavelength shifting in RAT. The default behavior is for GLG4Scint to handle opticalphotons as well as charged particles. Alternatively, you can also let GLG4Scint handle the primary scintillation, then use Geant4's G4OpWLS process or the custom BNLOpWLSModel to do the reemission.

GLG4Scint Model

The previous sections only apply to particles other than optical photons. Optical photons are ignored by GLG4Scint, *except* when the photon is absorbed inside the medium, but not at a geometry boundary. If the photon is absorbed in the bulk, then it is possible that it was absorbed by wavelength-shifter present in the scintillator.

The decision whether to reemit the photon is made by looking at the REEMISSION_PROB table, which gives the Poisson mean number of photons number of photons produced per photon absorbed. (NOTE: This model is used because TPB shifts extreme UV light to visible light, so it is energetically possible for more than one photon to be produced. This model of reemission may not be applicable to all wavelength shifters.) The number of outgoing photons is drawn from this Poisson distribution.

The spectrum of the outgoing photons is drawn from a separate distribution from the primary scintillation distribution, unless no wavelength-shifting distribution is specified. In this case, the scintillation distribution is reused.

Wavelength shifted photons are delayed from their absorption time according to the same time distribution as the original scintillator. (WARNING: THIS IS ALMOST CERTAINLY WRONG FOR MEDIA WITH BOTH SCINTILLATOR AND WAVELENGTH SHIFTER. SHOULD FIX!)

G4OpWLS Model

Choose this model in the macro with:

```
/PhysicsList/setOpWLS g4
```

before calling initialize. See the Geant4 documentation for more details on the required material properties.

BNLOpWLS Model

Choose this model in the macro with:

```
/PhysicsList/setOpWLS bnl
```

This was written by L. Bignell at BNL to better model measurements of scintillator cocktails with secondary fluors. The reemission spectrum (and probability) is sampled depending on the photon wavelength, based on measured data. The file to read this data from is in RATDB, in *BNL_WLS_MODEL[].data_path*, which defaults to *data/ExEmMatrix.root*. The reemission time can be set to either a delta function or an exponential distribution, but currently is hard-coded to use an exponential. The latter is set through the property in the OPTICS table *WLSTIMECONSTANT*.

This model also requires OPTICS properties *QUANTUMYIELD* (vector, decides how many secondary photons to generate) and *WLSCOMPONENT* (vector, WLS wavelength intensity) for WLS materials.

This WLS model has been validated by Chao Zhang of BNL. See these slides for details: `bnl_wls_validation.pdf`.

1.11.3 RATDB Fields

This section needs work!

All of the RATDB fields which control scintillation are found in the OPTICS table for that material. The following table describes these parameters. Note that ''x'' denotes a slot where a particle name (neutron,alpha,Ar40,Ar39,Ne20) which can be substituted in for particle-specific scintillation behavior. For example, one could create a field named SCINTILLATIONalpha_option. The default case would be written SCINTILLATION_option.

|| Name || Data type || Description || ||LIGHT_YIELD || double || Number of scintillation photons emitted per MeV of deposited energy || ||SCINTILLATION''x''_option || string || Units of SCINTILLATION''x''_value2. Always set to "dy_dwavelength". || ||SCINTILLATION''x''_value1 || double array || List of wavelength points in nanometers. || ||SCINTILLATION''x''_value2 || double array || Relative number of emitted photons at each wavelength (integral is unimportant) || ||SCINTILLATION_WLS''x''_option || string || Units of SCINTILLATION''x''_value2. Always set to "dy_dwavelength". || ||SCINTILLATION_WLS''x''_value1 || double array || List of wavelength points in nanometers. || ||SCINTILLATION_WLS''x''_value2 || double array || dN/dlambda at each wavelength (integral is unimportant) || ||SCINTMOD''x'' || double array || Three elements: [0.0d (always), Birk's Constant, Reference ''dE/dx'']. || ||REEMISSION_PROB_option || string || Units of REEMISSION_PROB_value1. Always set to "wavelength". || || REEMISSION_PROB_value1 || double array || List of wavelength points in nanometers. || || REEMISSION_PROB_value2 || double array || Mean number of reemitted photons for absorbed photon at that wavelength || ||QF''x''_option || string || Units of QF''x''_value1. Always set to "energy". || ||QF''x''_value1 || double array || List of energy points in MeV || ||QF''x''_value2 || double array || Quenching factor at each energy point ||

The time delay for scintillation photons is controlled by SCINTWAVEFORM, LONGTIMECONST and ShortTIMECONST (note capitalization).

=== Case 1: Sampled time distribution ===

||SCINTWAVEFORM''x''_value1 || double array || List of time points in nanoseconds || ||SCINTWAVEFORM''x''_value2 || double array || dN/dt at each time (integral is unimportant) ||

=== Case 2: Sum of exponential distributions ===

||SCINTWAVEFORM''x''_value1 || double array || List of exponential time constants (in ns). All elements are negative. || ||SCINTWAVEFORM''x''_value2 || double array || Branching fraction for each time constant. Sum of elements must be equal to 1.0. ||

=== Case 3: Energy dependent, two exponential === || LONGTIMECONST''x''_option || string || Units of value1. Always set to "energy".|| || LONGTIMECONST''x''_value1 || double array || List of energy points in MeV. Correspond to the current energy of the incident particle. || || LONGTIMECONST''x''_value2 || double array || Exponential time constant (ns) at each energy point. All elements are negative. || || ShortTIMECONST''x''_option || string || Units of value1. Always set to "energy".|| || ShortTIMECONST''x''_value1 || double array || List of energy points in MeV || || ShortTIMECONST''x''_value2 || double array || Exponential time constant (ns) at each energy point. All elements are negative. || || LONGTIMEWEIGHT''x''_option || string || Units of value1. Always set to "energy".|| || LONGTIMEWEIGHT''x''_value1 || double array || List of energy points in MeV || || LONGTIMEWEIGHT''x''_value2 || double array || Fraction of photons which are drawn from the long time constant at each energy. The fraction of photons drawn from the short time constant is (1 - this value).||

Note that all material property names (the names before _value1 and _value2) must be listed in the PROPERTY_LIST string array. (See examples for this to be more clear.)

1.12 PMT Simulation

RAT uses a custom PMT simulation extracted from GLG4Sim.

1.12.1 PMT Models

Various PMTs build types are stored in tables named `PMT` and referenced by their index which refer to a `pmt_model`. RAT-PAC supports multiple build types of PMTs which are placed in the geometry with the `pmtarray` geometry factory. The following build types are supported with the specified fields

Toroidal (GLG4TorusStack):

Field	Type	Description
`construction`	`string`	Must be "toroidal".
`dynode_material`	`string`	Material used for the dynode stack.
`glass_material`	`string`	Material used for the PMT body.
`pmt_vacuum_material`	`string`	Material used for the PMT vacuum.
`photocathode_surface`	`string`	Surface containing photocathode properties.
`mirror_surface`	`string`	Surface containing reflective back properties.
`dynode_surface`	`string`	Surface containing dynode surface properties.
`dynode_radius`	`double`	Radius of dynode.
`dynode_top`	`double`	Top Z of dynode relative to PMT equator.
`wall_thickness`	`double`	Thickness of the PMT body.
`z_edge`	`double[]`	PMT shape argument to GLG4TorusStack.
`rho_edge`	`double[]`	PMT shape argument to GLG4TorusStack.
`z_origin`	`double[]`	PMT shape argument to GLG4TorusStack.

Revolution (G4Polycone):

Field	Type	Description
construction	string	Must be "revolution".
dynode_material	string	Material used for the dynode stack.
glass_material	string	Material used for the PMT body.
pmt_vacuum_material	string	Material used for the PMT vacuum.
photocathode_surface	string	Surface containing photocathode properties.
mirror_surface	string	Surface containing reflective back properties.
dynode_surface	string	Surface containing dynode surface properties.
dynode_radius	double	Radius of dynode.
dynode_top	double	Top Z of dynode relative to PMT equator.
dynode_height	double	Total height of the dynode stack.
rho_inner	double[]	Cylindrical radius of inner edge coordinates.
z_inner	double[]	Z value of inner edge coordinates.
rho_edge	double[]	Cylindrical radius of outer edge coordinates.
z_edge	double[]	Z value of outer edge coordinates.

PMTINFO Tables:

These tables are used by pmtarray to place PMT and the information contained is passed along to Gsim. The order of items in this PMT array assigns IDs to the PMTs, though the order of placement determines the order in which various PMTINFO tables are assigned indexes when using multiple pmtarray elements. Positions given in these tables are in global coordinates. If the PMTs are placed in sub volumes the appropriate transformations are made on the fly.

Field	Type	Description
x	double[]	List of X positions in global coordinates.
y	double[]	List of Y positions in global coordinates.
z	double[]	List of Z positions in global coordinates.
dir_x	double[]	List of X direction vector (will be normalized).
dir_y	double[]	List of Y direction vector (will be normalized).
dir_z	double[]	List of Z direction vector (will be normalized).
type	int[]	List of logical PMT types (stored in DB).
efficiency	double[]	List of individul PMT efficiency corrections (optional).

1.12.2 Q/T Response

Gsim checks the database for single photoelectron charge and transit time PDFs automatically for PMT models that are added to the geometry. These PDFs are stored in tables named PMTCHARGE and PMTTRANSIT respectively, where the index corresponds to a pmt_model field used in GEO tables. These PDFs are sampled whenever a photon is absorbed by the photocathode to create a realistic Q/T response automatically for PMTs independent of any DAQ processor. If no tables are defined for a pmt_model the time defaults to approximately zero spread from photoelectron absorption time and the charge defaults to a phenomenological model used by MiniCLEAN.

PMTCHARGE fields:

Field	Type	Description
charge	double[]	"X" values of the charge PDF (arbitrary units)
charge_prob	double[]	"Y" values of the charge PDF (will be normalized)

PMTTRANSIT fields:

Field	Type	Description
cable_delay	double	Constant offset applied to all PMTs of this model (nanoseconds)
time	double[]	"X" values of the time PDF (nanoseconds)
time_prob	double[]	"Y" values of the time PDF (will be normalized)

1.12.3 Efficiency Model

Efficiency for detecting a photo electron before any DAQ process is applied is based on the following.

- PMT Geometry - reflections of the photons from the PMT and internal reflections.
- Photocathode QE - defined in the photocathode surface OPTICS table as the EFFICIENCY property (wavelength dependent).
- Placement Correction - each pmtarray GEO placement has an (optional) efficiency_correction muplicative factor that applies to all PMTs it palces.
- Individual Correction - the (optional) efficiency field in PMTINFO tables specifies an individual correction to each PMT (settable from macro).

1.12.4 Dark Current

PMTs have an intrinsic noise rate, or "dark current", which results from thermal excitation at the single electron level. These thermal electrons can exactly mimic a photoelectron from the PMT's photocathode and, thus, noise hits cannot be distinguished from 'true' hits caused by incident photons.

On the upside, this makes the noise hits fairly simple to simulate: we can draw from the same charge spectrum as is used for regular PMT hits. The only subtleties are in the timing distribution of the hits, and the rates at which noise is generated. This document describes the inclusion of noise hits in RAT.

The old noise processor (source:rat/src/daq/NoiseProc.cc) had several problems with the implementation, in particular the hits were incorrectly distributed in time (generated for one sampling window width from the start of the simulated event, therefore not extending throughout the event window), and noise was defined in terms of a number of hits per event instead of a rate (which is what we are more likely to measure). In addition, under the principle of "apathy should lead to the most realistic simulation possible" (- Dr Stanley Seibert), it was decided to incorporate noise hits into the default event simulation, rather than retaining the optional processor. This avoids the possibility of noise not being included either through forgetfulness, or because the noise processor was run in the wrong order, for example after the DAQ processors. As a result, running RAT will now include PMT noise hits by default, unless they are switched off. Details on how to do so follow.

Control

There are three options for the inclusion of noise, as follows:

0: No noise hits simulated.

1: Average noise rate read from 'noise_rate' in DAQ.ratdb.

2: Tube-by-tube noise rates read from 'PMTnoise' in DAQ.ratdb.

These options are controlled by the use of the 'noise_flag', in the DAQ.ratdb file. This flag can be include' in RAT macros as follows:

```
/rat/db/set DAQ noise_flag x
```

where x = 0, 1, or 2, depending on the noise model chosen.

The noise is included in the simulation after the physics event has been propagated (all particles followed to extinction, and PMT hits recorded) but before the DAQ, which runs as a separate processor. All noise hits are flagged with the 'isNoise' bit in the MCPhoton class (set to 1 for noise hits, and 0 for regular PMT hits).

Timing Distribution

Noise hits are generated uniformly in time, throughout a pre-defined 'noise window'. The DAQ records data beginning from a predefined time before a trigger. This time period (pretrigger time) is given by a fixed number (currently 1350) of sampling windows (4ns each). We want noise to be simulated throughout any recorded waveform. The noise window therefore begins at a time before the first photon hit given by this pretrigger time. To allow for a noise hit in coincidence with the last recorded photon to cause a trigger, the noise window continues until the width of the event window, plus the width of a single discriminator pulse, past the last photon hit time.

Speed and file size Comparison

To determine the effect of including noise in the default simulation, I generated 1000 isotropic 20keV electrons at the centre of the detector using each noise model. Both the average and the individual tube noise rates were set to 2000Hz, to emphasise any impact of including noise hits (default is 500Hz). The results, in terms of CPU usage (output file size), were as follows:

Noise model 0: 2280.91 s (46M)

Noise model 1: 2285.77 s (48M)

Noise model 2: 2341.45 s (48M)

So including noise in the simulation increases the processing time by 0.2%, and simulating noise tube-by-tube increases it by a further 2.4%.

The file size increased by ~5% when noise was included in the simulation.

Parameters

All are stored in DAQ.ratdb

noise_rate: 500.0d, // The mean noise rate across all PMTs, in Hz

PMTnoise: [], // an array with 92 entries: individual noise rates per PMT, in Hz

noise_flag: 0, // the flag to determine which noise model you use (default is to turn noise off completely)

1.13 Input Producers

1.13.1 InROOT

The InROOT event producer reads events from a ROOT format data file, as produced by [wiki:UserGuideOutRoot OutROOT], and passes them one at a time to the event loop.

Command

```
/rat/inroot/read filename
```

- filename - name of ROOT file to open. Note the lack of quotation marks.

1.13.2 InNet

The InNet event producer listens to a network socket and passes events it receives to the event loop. Multiple remote hosts may all send events at once. !InNet will process them in roughly first-come-first-serve order, but no attempt is made at strong fairness. Runs forever until manually terminated with Ctrl-C.

InNet has ‘‘NO SECURITY’‘ and will accept connections from any computer on the network. You should only use it on computers which are isolated from the rest of the Internet by a firewall.

Command

```
/rat/innet/listen port
```

- port - integer, TCP/IP port number to listen for events on

1.14 Event Processors

1.14.1 simpledaq

The SimpleDAQ processor simulates a minimal data acquisition system. The time of each PMT hit is the time of the first photon hit plus the timing distribution of the appropriate PMT, and the charge collected at each PMT is just the sum of all charge deposited at the anode, regardless of time. All PMT hits are packed into a single event.

Command

```
/rat/proc simpledaq
```

Parameters

None.

1.14.2 count

The count processor exists mostly as a simple demonstration processor. It also displays messages periodically showing both how many physics events and detector events have been processed. The message looks something like:

```
CountProc: Event 5 (8 triggered events)
```

Command

```
/rat/proc count
```

Parameters

```
/rat/procset update [interval]
```

- interval (optional, integer) - Sets number of physics events between between status update messages. Defaults to 1 (print a message for every event).

1.14.3 prune

The Prune processor is not a kitchen aid, but rather a processor for removing unwanted parts of the data structure to save space. The prune processor is very useful to call before the [wiki:UserGuideOutRoot OutROOT] processor to avoid writing large amounts of data to disk.

Note that there is minimal benefit to pruning in order to save memory in the running program. Only one data structure is present in memory at any given time, and it is never copied. Only when lots of events are written to disk does the overhead become considerable.

Command

```
/rat/proc prune
```

Parameters

```
/rat/procset prune "cutlist"
```

- cutlist - (required) a comma separated (no spaces) list of parts of the data structure to remove. [[BR]]The currently allowed entries are:
- mc.particle
- mc.pmt
- mc.pmt.photon
- mc.track
- ev
- ev.pmt

If /tracking/storeTrajectory is turned on, mc.track:particle is used, where particle is the name of the particle track you want to prune (mc.track:opticalphoton will prune optical photon tracks).

A complex example of pruning can be seen in the [source:RAT/trunk/mac/prune.mac#latest prune.mac] macro file included in the RAT source.

1.14.4 fitcentroid

The !FitCentroid processor reconstructs the position of detector events using the charge-weighted sum of the hit PMT position vectors.

Command

```
/rat/proc fitcentroid
```

Parameters

None

Position fit information in data structure

- name - "centroid"
- figures of merit - None

1.14.5 fitpath

The `fitpath` processor is an implementation (still a work in progress) of the successful PathFitter algorithm used in SNO. It fits position, time, and direction for cherenkov events using a maximum likelihood fit of hit time residuals while taking into account different paths the hit could have taken. For "direct" light (i.e. neither reflected nor scattered) an angular distribution of cherenkov light is taken into account to fit the direction. All other light is considered "other" and does not contribute to the direction fit.

Minimization is done in three stages: 0) Hit time residuals are minimized directly using *SimulatedAnnealing* from a static seed. 1) PathFitter likelihood is minimized with *SimulatedAnnealing* from stage 0's result. 2) PathFitter likelihood is minimized with Minuit2 from stage 1's result.

Command

```
/rat/proc fitpath
```

Parameters

None required from macro. `fitpath` reads parameters from a table `FTP` containing the following fields:

Field	Type	Description
num_cycles	int	Number of annealing iterations (times to lower temp)
num_evals	int	Number of evaluations per iteration (evals per temp)
alpha	double	Controls the rate of cooling in *SimulatedAnnealing*
seed_pos	double[3]	Static position seed to stage 0
pos_sigma0	double	Size of initial stage 0 simplex in position coordinates
seed_time	double	Static time seed to stage 0
time_sigma0	double	Size of initial stage 0 simplex in time
temp0	double	Initial temperature of *SimulatedAnnealing* for stage 0
seed_theta	double	Static theta (detector coordinates) seed to stage 1
theta_sigma	double	Size of initial stage 1 simplex in theta
seed_phi	double	Static phi (detector coordinates) seed to stage 1
phi_sigma	double	Size of initial stage 1 simplex in phi
pos_sigma1	double	Size of initial stage 1 simplex in position coordinates
time_sigma1	double	Size of initial stage 1 simplex in time
temp1	double	Initial temperature of *SimulatedAnnealing* for stage 1
cherenkov_multiplier	double	Number of cherenkov photons generated per hits detected
light_speed	double	Speed of light in material in mm/ns
direct_prob	double	Fraction of direct detected light
other_prob	double	Fraction of late detected light
photocathode_area	double	Area of photocathode mm^2
direct_time_first	double	Time (ns) of first entry in direct_time_prob
direct_time_step	double	Time step (ns) between entries in direct_time_prob
direct_time_prob	double[]	Probability (need not be normalized) of being "direct" light with a certain time residual
other_time_first	double	Time (ns) of first entry in other_time_prob
other_time_step	double	Time step (ns) between entries in other_time_prob
other_time_prob	double[]	Probability (need not be normalized) of being "other" light with a certain time residual
cosalpha_first	double	Cos(alpha) of first entry in cosalpha_prob
cosalpha_step	double	Cos(alpha) step between entries in cosalpha_prob
cosalpha_prob	double[]	Probability (need not be normalized) of Cherenkov light being emitted at a certain cos(alpha) w.r.t. particle direction

Fit information in DS

In the EV branch the PathFit class contains Get/Set methods for the following data:

Field	Type	Description
Time0	double	Time seed from simple hit time residual minimization
Pos0	TVector3	Position seed from simple hit time residual minimization
Time	double	Time resulting from final stage of minimization
Position	TVector3	Position resulting from final stage of minimization
Direction	TVector3	Direction resulting from final stage of minimization

PathFit implementes PosFit under the name fitpath.

1.14.6 outroot

The OutROOT processor writes events to disk in the ROOT format. The events are stored in a TTree object called "T" and the branch holding the events (class [source:RAT/trunk/include/RAT_DS.hh#latest RAT_DS]) is called "ds".

Command

```
/rat/proc outroot
```

Parameters

```
/rat/procset file "filename"
```

- filename (required, string) Sets output filename. File will be deleted if it already exists.

1.14.7 outnet

The !OutNet processor transmits events over the network to a listening copy of RAT which is running the [wiki:UserGuideInNet InNet] event producer. Multiple listener hostnames may be specified, and events will be distributed across them with very simplistic load-balancing algorithm.

This allows an event loop to be split over multiple machines. I'll leave it to your imagination to think up a use for this…

Command

```
/rat/proc outnet
```

Parameters

```
/rat/procset host "hostname:port"
```

- hostname:port (required) Network hostname (or IP address) and port number of listening RAT process.

=== Notes ===

The "load balancing" mentioned above distributes events by checking to see which sockets are available for writing and picking the first one that can be found. The assumption is that busy nodes will have a backlog of events, so their sockets will be full. In principle, this means that a few slow nodes won't hold up the rest of the group.

This processor and its [wiki:UserGuideInNet sibling event producer] have no security whatsoever. Don't use your credit card number as a seed for the Monte Carlo.

1.15 User-Defined Processors

One highly-encouraged method of coding up an analysis task is to write it as a user-processor. You can write a RAT macro file which uses your user processor (on either fresh Monte Carlo data or old data loaded from a file) to operate on the data, rather than write a separate ROOT macro. This has some advantages:

- You have access to all the RAT classes, which you do not in ROOT.
- Your code is compiled, so it runs faster than in ROOT. (Not a major advantage, since you can also compile ROOT macros relatively easily.)
- If your processor grows into something generally useful, you can easily convert it into an "official" processor and submit it for inclusion in RAT.

1.15.1 Preparation

Make sure that you (or your site administrator) has already compiled RAT and set up your environment as described in the [wiki:UserGuideInstallation installation page].

1.15.2 Getting started

A good place to start is to copy the $RATROOT/user directory to some location where you will develop your processor. You can name this new directory anything you like. (If your copy of the user directory contains .svn/, you can remove that.)

The contents of the user directory are:

- SConstruct - Scripts the build process for making a new copy of RAT
- TestUser.cc - Contains both the declaration and definition of a new RAT user processor.
- testuser.mac - A rat macro which runs the user processor defined in !TestUser.cc.

Now compile the sample processor and run the test macro by typing:

```
scons
./myrat testuser.mac
```

Note SCons created a custom version of RAT called "myrat". This new executable contains the standard RAT application, plus the extra user processor in !TestUser.cc.

If all runs correctly, you should have a file called "nhits.root" in the current directory. This ROOT file contains a single histogram called "nhits" which will show the number of PMTs hit in each event.

1.15.3 Making your own changes

If you only need one user processor, you can just edit !TestUser.cc directly. You can also add new source files to your directory (make sure they end with .cc), and they will all be compiled and linked into your custom version of RAT.

1.15.4 Replacing the main() function

By default, the build script compiles and links rat.cc, which contains the standard RAT main() function. If you would like to replace this main() function with your own, edit the SConstruct file, and change:

```
mymain = False
```

to:

```
mymain = True
```

Then the main() function you define in one of your source files will be used instead.

1.16 Off-line Analysis in ROOT

1.16.1 Introduction

ROOT is currently the tool of choice for analyzing RAT output for a variety of reasons which include that ROOT commands look like C++. For a high-level look on how to use ROOT, you should consult the [http://www.slac.stanford.edu/BFROOT/www/doc/workbook/root1/root1.html BARBAR collaboration's ROOT page], because it is well done and there is no point to repeat that here. The purpose of this article, as it stands, is to familiarize the user with ROOT analysis pertaining to RAT in particular.

Macros versus command-line

Like in RAT, it is better to use macros than command-line commands to perform certain tasks. In both cases, this stems from the complexity of the commands and also because there happen to be a lot of commands needed to do just about anything. It will be assumed here that you are using macros.

1.16.2 Starting your macro

Function definition and asthetics

Macros can be used to generate plots to disk in some easily accessible format like jpeg. To start your macro you need to define a function and also you will need to change some of the visual information in ROOT. For example, here is the top of $RATROOT/absorbtion.c, whose code is either commented or clearly purposed:

```
void absorbtion()
{
  //
  // First, let's re-set some graphical options, to overcome root's
  // disasterous defaults.
  //
  gROOT->SetStyle("Plain");

  gStyle->SetOptStat(0); // This determines if you want a stats box
  gStyle->SetOptFit(0); // This determines if you want a fit info box
  gStyle->GetAttDate()->SetTextColor(1);
  gStyle->SetOptTitle(0); // no title; comment out if you want a title
  gStyle->SetLabelFont(132,"XYZ");
  gStyle->SetTextFont(132);
  gStyle->SetTitleFont(132,"XYZ");

  gROOT->ForceStyle();
```

File access

Now that you have your plots looking pretty, it is time to get information into ROOT so that way you actually have something to display. To do this, you declare a a TFile object and a TTree object. After declaring the TFile object, as shown below, to link to your ROOT output, then the TTree object links to the tree in the ROOT file. The [http://en.wikipedia.org/wiki/Tree_data_structure tree] is just a way to store information, so it means the same thing here as it does in regular programming. Here is an example:

```
TFile absfile("../test_absorbtion.root");
TTree *T1=(TTree*)absfile.Get("T");
```

Creating a histogram

Now we have access to our data, but we need to create something like a histogram or n-dimensional plot to put the data in. Earlier we had only defined how things should look, we have not defined a histogram or something of the like. Histograms, in root are objects called TH1F (and if you figure out what it stands for, add it to this page). The constructor to TH1F takes a few arguments, as seen below: the name, options, the number of bins, the minimal value and the maximum value. The functions important functions that act on TH1F are well named, so you can figure out what they do just by looking at the example below:

```
TH1F *noabs = new TH1F("noabs"," ", 200, 1500, 3000);
noabs->SetLineColor(kBlue);
noabs->SetLineWidth(3);
noabs->SetLineStyle(1);
noabs->SetXTitle("Number of photons hits per event (hits)");
noabs->SetYTitle("Number of events (events)");
noabs->SetTitle("Photons to hit PMTs with and without attenuation");
```

Filling the histogram

We now have a histogram, but it isn't being displayed anywhere, so it is time to fix that. There is a function which acts upon the TTree object we defined earlier called "Draw" which takes data from the tree and puts it in some plot like a histogram. So, for example:

```
T1->Draw("numPE>>noabs");  // Puts "numPE" from file into "noabs" histogram
```

Rendering the histogram

TH1F, like all plot objects like histograms, also has a function called Draw, which doesn't have to take arguements, that is used for writing to a canvas, where a canvis is the window you see on your screen. If no canvas has been created, it creates one called "c1". So after running the command:

```
noabs->Draw();
```

You should see your plot.

Writing the image to disk

Here is an example of writing the image to disk in many formats:

```
c1->Print("absorbtion.gif");
c1->Print("absorbtion.eps");
c1->Print("absorbtion.jpg");
```

And now you are done.

1.16.3 Further examples

There are further examples in CVS in $RATROOT/root which cover a wide range of functions, such as drawing lengends, drawing multiple sets of data on the same canvas and so on and so forth. These are worth taking a look at since most of what will ever need to be done in RAT has been done, and is available in the root directory.

1.17 Particle Tracks

The Monte Carlo simulation has the ability to store the entire track for every particle, including optical photons, generated in an event. This can consume a great deal of disk space (tens of megabytes per event, depending on energy) and even slow down the event loop by bogging down the [wiki:UserGuideOutRoot outroot] processor. For this reason, track storage is ''disabled" by default.

However, if track information in the event loop is desired, it can be enabled with the command:

```
/tracking/storeTrajectory 1
```

1.17.1 Track Representation: Lists, Trees, and "Reality"

Deciding how to represent particle tracks given the diverse selection of particles and interactions is not a simple question. There are several ways used in different parts of RAT, which are reviewed below.

"Reality"

In GEANT4's version of reality, a particle track consists of a list of discrete points in spacetime. You may assume that the particle travels in a straight line between each point. There are some subtleties here since GEANT4 can also simulate multiple scattering 'within' a single track segment in order to boost performance. However, we don't understand when and where that happens quite yet, so we will ignore it.

Each point, except the initial point, in the track has associated with it a "process". (Not to be confused with RAT "processors, "

- The energy and momentum of the particle can be altered.
- The particle can be destroyed. (AKA "the track is killed")
- New particles can be created.

However, the type of particle (electron, positron, Gd-153 nucleus, etc) cannot change during a track. New particles always get new tracks.

In principle, new tracks created by a process can start anywhere, but in practice they either (a) always start at the interaction point (such as in a discrete process like pair production) or (b) are distributed along the track segment between the current point and the previous one (such as in a continuous process like ionization).

For extra confusion, there is a special "post-process" that isn't really a process at all. This process is run specially by the GLG4s

1. If the track is an optical photon and it was killed (i.e. "absorbed"), then throw a random number and check if it should be re-emitted by the wavelength shifter in the scintillator. The new photon will have a different wavelength in general, and be placed in a new track.
2. Otherwise, if energy was deposited in the scintillator (such as via an ionization process), generate scintillation photons according to the light yield, and distribute them along the track segment.

Lists

The most simple-minded way to pack all this information into the data structure that is written to disk is to make a list of tracks. Each track in turn contains a list of track steps, one for each point along the track. The nested lists are easy and quick to generate as the simulation runs, and they are easy to store in classes without the use of pointers, which is a big plus. This is how tracks are stored in the ROOT files on disk.

Trees

Of course, this list format is not very convenient for browsing the tree, or answering questions which concern the relationships between tracks. For these sorts of studies, we need a data structure that allows us to find the parent and child tracks.

To do this, we use a separate class (RATTrackNav) to convert the lists in the ROOT file into a full tree (in the graph theory sense) in memory. Each node represents a particle at a given point in its track. A node has a time, position, momentum, energy, particle type, process, etc. A node also has a pointer to the "previous" and "next" node. The previous node is either the previous point in the same track, or if this is the first point in a given track, the previous node will be a the node in the parent track where this track was created. Similarly, if new tracks are created at a given point by the active process (like ionization), then the node will have pointers to them in a separate "child" list.

This model is a direct representation of "reality" in the case of discrete processes which always create new tracks at the point of interaction. Continuous processes, like scintillation, which distributes new tracks between interaction points create tracks with ambiguous parent nodes (but not ambiguous parent tracks). Do we assign the parent to be the node at the beginning or end of the parent segment? Since the tracks were created in the GEANT4 simulation by the process acting on the point at the ''end'' of the segment, that is the convention we have adopted in the tree structure.

1.17.2 Working with Tracks in ROOT

To construct the track tree for an event, you first need to load an event into memory. The easiest way to do that is to use the RATDSReader class in your ROOT macro:

```
RAT::DSReader r("testIBDgen.root");
RAT::DS::Root *ds = r.NextEvent();
```

Once you have a data structure object, you can convert the tracks into a tree using the RATTrackNav class, and get a "cursor" which represents a location in the tree:

```
RAT::TrackNav nav(ds);
RAT::TrackCursor c = nav.Cursor(true);
```

The boolean argument to the Cursor() method selects whether you want the cursor to print out a summary of the current track every time you move the cursor. This is very useful when interactively browsing a tree. However, for a macro which is moving around the tree frequently, it is a nuisance, so by default, a RATTrackCursor prints nothing to the screen.

Now that you have a cursor, you can move around the tree using methods which start with "Go". Here is an example, with the verbose output from the cursor being displayed:

```
root [16] RATTrackCursor c = nav.Cursor(true);
Track 0: TreeStart
-------------------------------------------------------------------------------
 # |          position                |  MeV |     process    |   subtracks
-------------------------------------------------------------------------------
* 0. (   0.0,   0.0,   0.0)      _____ <0.001                  ->e+(1),neutron(2)
root [17] c.GoChild(0)
Track 1: e+  parent: TreeStart(0)
-------------------------------------------------------------------------------
 # |          position                |  MeV |     process    |   subtracks
-------------------------------------------------------------------------------
* 0. ( 485.6,-543.3, 439.7)      scint  1.770           start ->17 tracks
  1. ( 485.6,-543.4, 439.7)      scint  1.757           eIoni ->107 tracks
  2. ( 485.5,-543.8, 439.6)      scint  1.693           eIoni ->613 tracks
  3. ( 485.4,-545.8, 440.1)      scint  1.408           eIoni ->855 tracks
```

(continues on next page)

(continued from previous page)

```
  4. ( 485.5,-547.4, 440.1)       scint  1.212           eIoni ->501 tracks
  5. ( 485.3,-547.8, 440.0)       scint  1.145           eIoni ->683 tracks
  6. ( 484.4,-549.0, 438.6)       scint  0.748           eIoni ->735 tracks
  7. ( 483.5,-549.2, 438.1)       scint  0.572           eBrem ->515 tracks
  8. ( 483.1,-549.9, 438.1)       scint  0.440           eIoni ->111 tracks
  9. ( 483.1,-550.0, 438.0)       scint  0.415           eIoni ->166 tracks
 10. ( 483.1,-550.1, 437.9)       scint  0.398           eIoni ->987 tracks
 11. ( 483.0,-551.9, 438.8)       scint  0.120           eIoni ->259 tracks
 12. ( 483.1,-552.1, 439.1)       scint  0.024           eIoni ->21 tracks
 13. ( 483.1,-552.1, 439.1)       scint <0.001           eIoni
 14. ( 483.1,-552.1, 439.1)       scint <0.001         annihil ->44 tracks
(class RATTrackNode*)0x565aea0
root [18] c.GoParent()
Track 0: TreeStart
-------------------------------------------------------------------------------
 # |          position                |  MeV |     process    |   subtracks
-------------------------------------------------------------------------------
* 0. (   0.0,   0.0,   0.0)      _____ <0.001                  ->e+(1),neutron(2)
(class RATTrackNode*)0x4f6e2a0
root [19] c.GoChild(1)
Track 2: neutron  parent: TreeStart(0)
-------------------------------------------------------------------------------
 # |          position                |  MeV |     process    |   subtracks
-------------------------------------------------------------------------------
* 0. ( 485.6,-543.3, 439.7)       scint  0.011           start
  1. ( 498.8,-536.3, 439.7)       scint  0.008        LElastic ->proton(3)
  2. ( 503.5,-529.5, 437.0)       scint  0.005        LElastic ->proton(6)
  3. ( 513.9,-525.1, 438.0)       scint  0.004        LElastic ->proton(9)
  4. ( 516.1,-524.2, 437.5)       scint  0.001        LElastic ->proton(10)
  5. ( 526.0,-528.4, 423.1)       scint  0.001        LElastic ->C12[0.0](13)
  6. ( 524.8,-528.8, 425.1)       scint <0.001        LElastic ->proton(14)
  7. ( 530.1,-520.9, 438.2)       scint <0.001        LElastic ->proton(16)
  8. ( 530.9,-518.2, 439.0)       scint <0.001        LElastic ->proton(17)
  9. ( 492.1,-526.1, 450.3)       scint <0.001 NeutronDiffusionAndCapture ->5 tracks
(class RATTrackNode*)0x5f0ba80
```

The asterisk on the left shows you which step in the current track your cursor is pointing at.

Also, you'll notice that the Go methods return a RATTrackNode pointer in addition to moving the cursor. With this pointer, you can get information about the current node, like the energy/momentum/position/etc. If fact, you can get the pointer to the current node at any time using the Here() method on the cursor:

```
root [36] RAT::TrackNode *n = c.Here();
root [37] n->GetParticleName()
(string 0x5f0bae8)"neutron"
```

Names of the node attributes can be found in the [source:RAT/trunk/include/RATTrackNode.hh#latest RATTrackNode header file] and the [source:RAT/trunk/include/RAT_MCTrackStep.hh#latest RAT_MCTrackStep header file].

Other RATTrackCursor methods can be found by looking at the [source:RAT/trunk/include/RATTrackCursor.hh#latest header file].

Iterating through the Tree

Once you have the tree in memory, you will probably want to be able to step through every track in a loop. A depth-first iteration algorithm has been provided for you via the FindNextTrack() method. This will step through tracks, starting

from the current cursor location, going up and down the tree in a pattern that will ensure you visit every track once and only once. Since FindNextTrack() is concerned with visiting each ''track" and not each node, it returns the first node of each track, and none of the later nodes. When no more tracks exist to check, it returns 0.

```
RAT::TrackCursor c = nav.Cursor(false);
RAT::TrackNode *n = c.Here();
while (n != 0) {
    // Do something with n

    n = c.FindNextTrack();
}
```

Searching the Tree

A common task is to iterate through the tree, stopping at nodes which match some sort of criteria. For example, you may want to stop at each neutron track and ignore all the other particles. The generic way to do something like this is to write a ''boolean functor" that recognizes the node you want to stop on. For example, this is the functor that tests particle type:

```
class RAT::TrackTest_Particle : public RAT::TrackTest {
  std::string fParticleName;
public:
  RAT::TrackTest_Particle(const std::string &particleName) :␣
↪fParticleName(particleName) { };
  virtual bool operator() (RATTrackNode *c) { return fParticleName == c->particleName;
↪ };
};
```

Notice this functor uses a constructor to customize the type of particle it tests for. A functor to find electrons would be created with:

```
RAT::TrackTest *t = new RAT::TrackTest_Particle("e-")
RAT::TrackNode n = c.FindNextTrack(t);
```

and a positron test would look like:

```
RAT::TrackTest *t = new RAT::TrackTest_Particle("e+")
RAT::TrackNode n = c.FindNextTrack(t);
```

Other tests can be implemented by subclassing RATTrackTest in a similar fashion.

Search by particle type is such a common operation, that a shortcut method has been provided:

```
RAT::TrackCursor c = nav.Cursor(false);
RAT::TrackNode *n = c.FindNextParticle("e-");
```

You can call the FindNextTrack()/FindNextParticle() methods over and over again with the same test to iterate over just the tracks you are interested it.

1.17.3 Dealing with Optical Photons

By far, the bulk of the tracks generated by most events are composed of optical photons. However, for many studies the optical photons are of no interest at all (beyond perhaps the hits they register on the PMTs). In these situations, you can add a [wiki:UserGuidePrune prune processor] to your event loop to remove just the optical photons:

```
/rat/proc prune
/rat/procset prune "mc.track:opticalphoton"
```

You can use any other particle name in place of "opticalphoton" as well, and typing just "mc.track" will prune all tracks from the data structure. Note that this has no impact on the PMT hits. The photons are propagated to the PMTs no matter what, but the prune processor lets you delete them after they are no longer needed.

1.18 Creating and Running *rattest* Tests

1.18.1 Introduction

Rattest is a framework for creating unit and functional tests for RAT. These tests should be simple, testing only one aspect of the simulation. For instance, a test of attenuation in acrylic consist of a single light source in a world volume of acrylic – no PMTs or other geometry.

At minimum, a test consists of a RAT macro and a ROOT macro – the Monte Carlo and the analysis. New (simplified) geometries, modified RATDB databases, etc. can also be included. When run, these tests are compared to a standard via a KS test, and a web page is created with histograms (standard and current) and KS test results. The standard RAT logs and output ROOT file is also available for analysis.

The existing rattests are included with the standard RAT distribution, in *$RATROOT/test/*, with the functional tests in *$RATROOT/test/full/<test-name>*. To run a single test, *cd* to the test directory and simply run *rattest <test-name>* where *<test-name>* corresponds to a folder in *$RATROOT/test/full*. Rattest will iterate through the directory structure to find the test, run the RAT macro, run the ROOT macro on the output, and generate a report page.

The *rattest* utility takes the following options:

```
Usage: rattest [options]

Options:
  -h, --help          show this help message and exit
  -u, --update        Update "standard" histogram with current results
  -m, --regen-mc      Force Monte Carlo to be regenerated
  -r, --regen-plots   Force histograms to be regenerated
  -t, --text-only     Do not open web pages with plots
```

1.18.2 Existing RAT Tests

```
acrylic_attenuation
 Tests the attenuation length of acrylic by generating photons in an acrylic block␣
↪and checking track lengths
```

1.18.3 Writing a RAT Test

1. Create a new folder in *$RATROOT/test/full* with useful but short name for your test
2. Create a *rattest.config* file, like this:

```
#!python
# -*- python-*-
description = '''Tests the attenuation length of acrylic by generating photons in␣
↪an acrylic block and checking track lengths'''
```

(continues on next page)

(continued from previous page)

```
rat_macro = 'acrylic_attenuation.mac'
root_macro = 'acrylic_attenuation.C'
```

The RAT macro and ROOT macro do not need to have the same name as the test, they just have to be consistent with the actual filenames. *rattest* will find your ROOT output file, so you don't have to worry about it.

3. If necessary, create a RAT geometry (.geo) and any modified RATDB (.ratdb). As an example, *acrylic_attenuation* uses the default RATDBs (the default behavior), but the following geometry:

```
// -------- GEO[world]
{
name: "GEO",
index: "world",
valid_begin: [0, 0],
valid_end: [0, 0],
mother: "",
type: "box",
size: [10000.0, 10000.0, 10000.0],
material: "acrylic_polycast",
}
```

RAT will prefer a geometry or database in your test directory, and default to the ones in *$RATROOT/data*.

4. Create your RAT macro.

Keep things as simple as possible, and turn off as many options as possible. The *acrylic_attenuation* RAT macro:

```
/glg4debug/glg4param omit_muon_processes  1.0
/glg4debug/glg4param omit_hadronic_processes  1.0

/rat/db/set DETECTOR geo_file "acrylic_sphere.geo"

/run/initialize

# BEGIN EVENT LOOP
/rat/proc count
/rat/procset update 50

/rat/proc outroot
/rat/procset file "acrylic_attenuation.root"

# END EVENT LOOP
/tracking/storeTrajectory 1

/generator/add combo pbomb:point
/generator/vtx/set 100 100
/generator/pos/set  0.0 0.0 0.0


/generator/add combo pbomb:point
/generator/vtx/set 100 200
/generator/pos/set  0.0 0.0 0.0

...

/run/beamOn 500
```

5. Write a ROOT macro

The ROOT macro should create a histogram that captures the benchmark you are looking for. It should consist of a single *void* function *make_plots(TFile *event_file, TTree *T, TFile *out_file)*.

Basically, do your analysis, make a histogram, and output it with *[histogram name]->Write()*. Note that when using *Draw()* to make histograms, you'll probably want the *"goff"* option.

rattest will pull histogram names from this macro automatically for creation of the results page.

The ROOT macro from *acrylic_attenuation*:

```
void make_plots(TFile *event_file, TTree *T, TFile *out_file)
{

 ...

  TH1F *acr_attn_300 = new TH1F("acr_attn_300", "Photon track length (300 nm)", 20, 0,
↪ 2500);
  acr_attn_300->SetXTitle("Track length (mm)");
  acr_attn_300->SetYTitle("Count");
  T->Draw("mc.track.GetLastMCTrackStep()->length>>acr_attn_300","TMath::Abs(1.
↪2399727973642156e-03/(mc.track.GetLastMCTrackStep()->ke)-300)<10","goff");
  acr_attn_300->Fit("expo");
  acr_attn_300->Draw("goff");
  acr_attn_300->Write();

 ...

}
```

6. Test it

   Run your RAT macro with the usual *rat [macro name]*, then, in ROOT, run the contents of your analysis macro and ensure that you get what you were looking for.

7. Create a standard

   From the test directory, run *rattest -u [your test name]*. This will create the file *standard.root*, which will be the basis for comparison until the next time you run *rattest* with the *-u* option. Take a look at *results.html* to see how things worked out.

This is pretty much it. If you run *rattest [your test name]* again, you should get a results page (which will open in your default browser unless you specified the *-t* option) with very similar results.

If you think the test is useful to others, commit it to the RAT repository with svn. Be sure to commit only the *rattest.config*, RAT and ROOT macro, any geometry or RATDB files, and *standard.root*.

1.19 Setting up GEANT4 with HEPREP visuals and viewing in Wired 3

1.19.1 Background

Before you start, the people at KSU believe that JAS3 is the way to go for visuals, so for full disclosure, here is the link to their JAS3 how-to: [http://neutrino.phys.ksu.edu/cgi-bin/BWKSUwiki?JAS3 KSU JAS3 how-to]

One of the advantages of GEANT4 is that it allows for visualizations of the simulation. There are two types of visualizations: live and play-back. The live ways of visualizations included OPENGL. An example of a play-back style visualization is HEPREP.

This brief article will cover using HEPREP. The advantages of using HEPREP instead of a live solution is that you can play data back at the speed you want to see it, and you can also save simulation runs. HEPREP also has a program called wired associated with it which is responsible for viewing the HEPREP file. Setting up wired

Read the installation section and skim the rest of this site to setup WIRED:

[http://www.slac.stanford.edu/BFROOT/www/Computing/Graphics/Wired/ Wired3 Users Home Page]

1.19.2 Setting up GEANT4

In your mac directory you should create a file called vis.mac. This file will hold all of your visualization information. Here is an example:

```
/glg4debug/glg4param omit_muon_processes  1.0
/glg4debug/glg4param omit_hadronic_processes  1.0

/run/initialize

/process/activate Cerenkov

/vis/open HepRepFile
/vis/scene/create
/vis/scene/add/volume
/vis/sceneHandler/attach
/vis/scene/add/trajectories
/vis/scene/add/hits
/tracking/storeTrajectory 1 # view every photon (~18000 of them)
```

1.19.3 Running wired

After you run an event, you should have a file called G4DataX.heprep, where X is a positive integer, in the directory in which you ran your program. This file is probably very large. Wired reads in HEPREP files compressed, so you should gzip the file:

```
gzip G4DataX.heprep
```

Load of wired on the machine you want to view the data on and then load up the .heprep.gz file by during the normal File->Open sequence. You should now be viewing your data.

1.19.4 Examples

Examples can probably be found at:: [http://wired4.freehep.org/heprep/examples/Geant4-A01/]

1.20 Tools

1.20.1 Genie2Rat

Genie2Rat allows vertices generated with GENIE (www.genie-mc.org) to be processed with RAT. It converts the root output of GENIE's gevgen_atmo atmospheric neutrino simulation tool and converts it into RAT root format. Note that gevgen_atmo requires a ROOT geometry for your detector to create vertices in it. Once the GENIE file is created, this tool is then run using the syntax

```
genie2rat -i [input genie filename] -o [output root filename] (-N [number of events␣
↪to process])
```

and will output a RAT root file.

CHAPTER 2

Programmer's Guide

2.1 Extending the Data Structure

2.1.1 Planning

In order to extend the data structure (RAT::DS), you must decide several things:

What sort of data would you like to record?

If it is an atomic value, like a float or an integer, your task will be easy. If it is several pieces of data, it may make more sense to make a new class to put in the event data structure.

The only types allowed in the data structure are standard C++ atomic types like int and double (and ROOT-specific typedefs to them), STL containers, like vector and map, and classes which follow the RAT::DS style. That is, they are subclasses of TObject which follow the RAT style conventions. Note that C-style arrays and ROOT container classes (e.g. TClonesArray) are not allowed!

What do you want to call it?

Short (but not cryptic) names are preferred, and it helps to make them unique over all branches of the event data structure. This simplifies plotting with TTree::Draw() in ROOT later, so that the branch need not be specified to disambiguate a name.

Where will this data be created and where will it be used?

Data stored in the event structure must be "created" somewhere, either in a generator or a processor. You should decide where the code which computes this information should go. If the information is at all optional, it's best to put the computation in a separate processor so users who don't need the data can skip it and save running time.

Also remember that RAT::DS is the interface by which processors "communicate" with each other. Intermediate calculations should not be stored in the data structure unless they would be of interest to an end user. However, any data from one processor that is the input to the calculation of another processor should be stored in the data structure.

2.1.2 Adding Existing Data Types to Objects

If you are adding data of an already existing data type, you can just edit the appropriate header file and recompile. No other action is required, except editing the appropriate processor or generators to actually put some data there!

2.1.3 Creating a New Class for the Data Structure

If you need to create a new class, you need to:

Make the header file

Your new class must subclass TObject and be declared within the namespace RAT::DS. For ease, copy an existing event class and edit it. Do not forget to put the new class name in the !ClassDef macro.

Note that DS class members should not use a prefix like "f" (that is, use "pmtCount" rather than "fPMTCount"), since they are part of the user interface, in a sense. Data members should be private, with the interface defined by "getter" and "setter" methods. The name of the file should match the name of the class.

For example, a class with PMT information might look like this:

```
/**
 * @class RAT::DS::PMT
 * @author A. Hamsterton <hamster@urodents.edu>
 * @brief Example of PMT information
 *
 * This is a long description of the class. This whole
 * block is here so we can generate Doxygen documentation.
 */

#ifndef __RAT_DS_PMT___
#define __RAT_DS_PMT___

#include <TObject.h>
#include <vector>

namespace RAT {
  namespace DS {

class MyClass : public TObject {
public:
  MyClass() {}
  virtual ~MyClass() {}

  /// PMT channel number
  int GetPMTID() { return pmtID; }
  void SetPMTID(int _pmtID) { pmtID = _pmtID; }

  /// Digitized hit time information
  int GetADCTime() { return adcTime; }
  void SetADCTime(int _adcTime) { adcTime = _adcTime; }
```

(continues on next page)

(continued from previous page)

```
  /// Digitized hit charge information
  int GetADCCharge() { return adcCharge; }
  void SetADCCharge(int _adcCharge) { adcCharge = _adcCharge; }

  /// True hit time information
  int GetTime() { return time; }
  void SetTime(int _time) { time = _time; }

  /// True hit charge information
  int GetCharge() { return charge; }
  void SetCharge(int _charge) { charge = _charge; }

private:
  int pmtID;  ///< PMT channel number;
  int adcTime;  ///< Digitized hit time
  int adcCharge;  ///< Digitized hit charge
  float time;  ///< True hit time
  float charge;  ///< True hit charge

  ClassDef(PMT, 1)
};

  }  // namespace DS
}  // namespace RAT

#endif  // __RAT_DS_PMT__
```

Make an implementation

Make in implementation in a cc file with the same name as the header, if necessary. Remember that DS classes should not do very much work – they are vessels for data which is computed elsewhere (such as a generator or processor). Never should a DS class itself be computing the data that it stores.

Add to CINT

Edit the src/ds/LinkDef.h file to ensure your new class is added to the CINT dictionary, which is needed for ROOT macros to work properly. You will need to add a line that looks like:

```
#pragma link C++ class RAT::DS::PMT;
```

And further down, there needs to be a line like:

```
#pragma link C++ class vector<RAT::DS::PMT>;
```

Add to SConstruct

Add your class name to the cint_cls list in the $RATROOT/SConstruct file. This will ensure the build system invokes rootcint on your new event class and generates the code that allows the event data structure to be streamed to disk or over network connections.

Recompile

Run scons to recompile.

2.1.4 Update Documentation

Don't forget to update the documentation in $RATROOT/doc to reflect the changes you have made to the data structure! Remember to explain what the data is and what units it is stored in.

2.2 Accessing RATDB

If you are unfamiliar with using RATDB, you should go read the description of RATDB in the User's Guide for an explanation of the layout and terminology.

2.2.1 How do I access the data from inside my code?

Assuming the database has been already loaded (see section "How do I load data into RATDB?"), accessing the data inside your program requires only a few steps:

1. #include <RAT/DB.hh> at the top of your source file.
2. Get a pointer to the global database.
3. Obtain a link to the table you want to access.
4. Read the value of the field using your link.

Here is a code snippet that shows how this works:

```
RAT::DB *db = RAT::DB::Get();
RAT::DBLinkPtr lmedia = db->GetLink("MEDIA", "acrylic");  // Link to MEDIA[acrylic]
double rindex = lmedia->GetD("index_of_refraction");
```

Of course, writing those three lines over and over again would be extremely cumbersome, so it is better to obtain the link in your class constructor and just use it in your other methods. A longer example:

```
#include <RAT/DB.hh>

class Dummy {
public:
  Dummy();
  virtual ~Dummy();
  float SpeedOfLight();

protected:
  DBLinkPtr lmedia_acrylic;
};

Dummy::Dummy() {
  lmedia_acrylic = DB::Get()->GetLink("MEDIA", "acrylic");
  // Don't need to save the RAT::DB pointer. All you need is the
  // link object. You can leave off the index if you just want
  // empty index ""
}
```

(continues on next page)

(continued from previous page)

```
Dummy::~Dummy() {
  // You don't have to delete your links!  Automatically handled for you.
}

float Dummy::SpeedOfLight() {
  return 299792458.0 / lmedia_acrylic->GetF("index_of_refraction");
}
```

Note that you can get a link from the database at any time and save it for when you need it, even if the table you want hasn't been loaded yet. This works because the table isn't looked up until you call one of the Get methods on the link object. The link knows how to find the field in the appropriate table (checking the user, time and default planes).

The indirection of the link might seem like overkill, but it allows for the case where the value you are looking up changes during processing. For example, if you were processing a very long stretch of data, the attenuation length in the mineral oil might not be constant. Each time you call GetD(), the link will find the attenuation length for the current "detector" time. Figuring out how to do this fast (with caching or load-on-demand) becomes an implementation detail you don't have to worry about.

Get Methods

There is one Get method in the DBLink class for each data type, so you must know the data type of the field you are accessing (which you should anyway). The simple types have the names GetI(), GetD(), and GetS() and return int, double and std::string, as you might expect. There are no floats in the database – they are always promoted to doubles.

The array types are a little more complicated. To avoid copying large amounts of data around, these Get methods return a ''const reference" to the array. Rather than explain what that means exactly, it's probably easier to show an example:

```
RAT::DBLinkPtr lgeo_pmt = RAT::DB::Get()->GetLink("GEO_PMT");
const std::vector<double>& pmtx = lgeo_pmt->GetDArray("xpos");
const std::vector<double>& pmty = lgeo_pmt->GetDArray("ypos");
const std::vector<double>& pmtz = lgeo_pmt->GetDArray("zpos");

std::cout << "PMT 0 is at " << pmtx[0] << "," << pmty[0] << ","
          << pmtz[0] << std::endl;
```

You can use the reference to the vector just like an array. The const qualifier means that the compiler will forbid you from replacing elements of the array. This is a good thing as the physical array will be shared with other objects in the program that might not appreciate you changing it. If you do want a copy of the array for some reason, you can just do this:

```
std::vector<double> pmtx = lgeo_pmt->GetDArray("xpos");
```

You are now free to do whatever you want to pmtx. Don't do this too often, though, as copying the contents of the array could be rather slow if it is a big array.

As you might have guessed, the names of the Get methods for the array types follow a similar pattern: GetIArray(), GetDArray(), GetSArray().

2.3 Logging

The goal of the RAT::Log class is make it easy to output informational messages to the user immediately at the console, while also ensuring they get captured in a log file on disk. We also want it to be very easy for programmers to output

status messages.

The policy for RAT code is that all text output to the screen or intended to be logged must go through the RAT::Log system. That means no one should use cout, cerr, G4out or G4err in their RAT code.

All informational messages are classified into one of four categories (in ascending order of detail):

1. warn – Something unusual but not fatal has occurred. Fatal errors are handled separately (see Die() below).

2. info – Information about normal operation. Should not be lengthy or used too frequently to avoid overloading the user.

3. detail – Detailed information about software activities. A user who wishes to know exactly what the program did should be able to get it from the detail messages.

4. debug – Output only of interest to those trying to debug the operation of the software.

The logging system will keep track which of these will be output to the screen and which will be output to the log file.

2.3.1 Setting Up

If you are writing code to run inside the RAT application, the logging system is already set up for you. The user will have selected the name of the log file, and their desired display and logging levels. By default, the user will see all warn and info messages, and all warn, info, and detail messages will be written to the log file.

If you are using librat from your own application, then you will have to set up the logging system yourself before you use it:

```
#include "RATLog.hh"

int main(int argc, char* argv[]) {
  RAT::Log::Init("mylogfile.log", RATLog::INFO, RATLog::DETAIL);
  // Do stuff here.
}
```

The first parameter is the name of the file, the second is the maximum level of detail you want to see at stdout, and the third is the maximum level of detail you want to write to the log file.

2.3.2 Producing output

Inside your code, writing messages is easy. Just make sure to #include “RAT/Log.hh” at the top, then use logging objects just like you would cout or cerr:

```
#include "RAT/Log.hh"

namespace RAT {

info << "Adding FitCentroid to event loop." << newline;
warn << "No seed found for fit, using default." << newline;
detail << "Fit converged in " << iterations << " iterations." << newline;
debug << "Hit tube list: " << newline;

}
```

The messages will be displayed on screen and/or written to the log file according to the current user settings. Nothing needs to be checked in your code. For example, if the user has selected “info” display and “detail” logging, then on their screen, they will see:

```
Adding FitCentroid to event loop.
No seed found for fit, using default.
```

And in the log file, they will see:

```
Adding FitCentroid to event loop.
No seed found for fit, using default.
Fit converged in 17 iterations.
```

The debug line did not appear anywhere because they did not select that level of detail.

For complex formatted output with variables, C++ syntax like that used above is really clumsy. There is also a function included in RAT::Log called dformat() which works just like C's printf(), but returns an STL string:

```
detail << dformat("Fit converged in %d iterations.  Chi2 = %1.4f\n",
                  iterations, chi2);
```

See the [http://stlplus.sourceforge.net/stlplus/docs/dprintf.html STL+ documentation on dformat] for more details.

2.3.3 When really bad stuff happens...

If your code encounters a major problem, it is best to bail out immediately and tell the user why. For this purpose, use the Die() method in the RAT::Log class:

```
RAT::Log::Die("Could not open " + filename + " for input.");
```

This will print that message to the "warn" log stream and then terminate the program.

For convenience, there is also a function Assert() which aborts the program with a log message when a condition fails:

```
RAT::Log::Assert(2 + 2 == 5, "O'Brien was wrong");
```

2.4 Creating a Processor

Creating a new processor and adding it to RAT requires only a few steps.

2.4.1 Create the Class Header

First, you need to select a name for your processor class. By convention, processors are named "RAT::XXXXProc" where XXXX is some short descriptive name for what your processor does. If the processor is a fitter, it should be named "RAT::FitXXXXProc". All processors are subclasses of the RAT::Processor class, which defines the common interface for processors. The easiest way to create a processor class which follows this interface is to copy the CountProc files in src/core and edit them.

Next, you need to decide whether you want your processor to be invoked once per physics event or once per detector event. If you are interested in Monte Carlo information primarily, or need to consider all the detector events as a group, you should overload the DSEvent() method:

```
virtual RATProcessor::Result DSEvent(RAT_DS &ds);
```

However, if you are writing a processor that is primarily interested in detector events, it may be easier to instead overload the Event() method instead (this is not done in RAT::CountProc):

```
virtual RATProcessor::Result Event(RAT_MC &mc, RAT_EV &ev);
```

Event() will be called once for every "detector" event, even if there are multiple detector events in a particular physics event. The Event() method is only provided as a convenience, since you could implement the same behavior by writing your own loop in DSEvent() instead. You should only overload DSEvent() or Event(), but NOT BOTH.

Finally, you need to decide what parameters your processor will accept at runtime. Many processors will not need this feature at all. If your processor needs constants or other external data to function, you should put it into a RATDB table. Users can override RATDB values using the /rat/db/set command in their macro files.

Instead, parameters are for the constants that should be set per "instance" of a processor. A particular processor can be instantiated in an event loop more than once, and parameters are the only way to differentiate one instance from another. For example, the prune and outroot processors can be used multiple times:

```
/rat/proc prune
/rat/procset prune "mc.particle"
/rat/proc outroot
/rat/procset file "prune_mcparticle.root"
/rat/proc prune
/rat/procset prune "mc.pmt.photon"
/rat/proc outroot
/rat/procset file "prune_mc_particle_photon.root"
```

If you do need parameters, you will need to select names for them and decide what type of data you want. Currently, processors can accept parameters in int, float, and double, and string format by overloading the appropriate methods:

```
virtual void SetI(std::string param, int value);
virtual void SetF(std::string param, float value);
virtual void SetD(std::string param, double value);
virtual void SetS(std::string param, std::string value);
```

Only overload the methods you need.

2.4.2 Write the Class Implementation

When you implement your class, you should take a look at CountProc.cc for an example of how to implement the DSEvent() and SetI() methods.

The return value of DSEvent() and Event() both have the same meaning:

- Processor::OK - This event was successfully processed
- Processor::FAIL - A non-fatal error has occurred. This event will continue to be processed through the event loop, but a later processor may use this information to change its behavior.
- RATProcessor::ABORT - A non-recoverable error with this event has occurred. If a processor returns this value, then the processing of this event immediate stops, and the event loop starts over with the next event, if any.

If you encounter a big problem and want to terminate RAT immediately (this has a way of getting the attention of the user), use the RAT::Log::Die() method as described in the Logging section.

If you are implementing one of the parameter methods, you should use this general pattern (stolen from CountProc.cc):

```
void CountProc::SetI(std::string param, int value) {
  if (param == "update") {
    if (value > 0) {
      updateInterval = value;
    }
```

(continues on next page)

(continued from previous page)

```
    else {
      throw ParamInvalid(param, "update interval must be > 0");
    }
  }
  else {
    throw ParamUnknown(param);
  }
}
```

The exceptions will be caught by the RAT command interpreter and appropriate error messages will be shown to the user before aborting the application.

2.4.3 Register the Class with ProcBlockManager

Finally, once you have your processor implemented, you need to edit src/cmd/ProcBlockManager.cc to register your processor so that users can add it to their event loops. Include the header for your processor at the top, then find the relevant code in the constructor:

```
// Create processor allocator table
procAllocators["count"] = new ProcAllocatorTmpl<CountProc>;
procAllocators["outroot"] = new ProcAllocatorTmpl<OutROOTProc>;
procAllocators["outnet"] = new ProcAllocatorTmpl<OutNetProc>;
procAllocators["prune"] = new ProcAllocatorTmpl<PruneProc>;
procAllocators["fitcentroid"] = new ProcAllocatorTmpl<FitCentroidProc>;
procAllocators["fitpoisson"]= new RATProcAllocatorTmpl<FitPoissonProc>;
```

To add your new processor to the list, pick a short name for it. (A good choice is the part between "RAT" and "Proc".) Then add a line to the end of that list, like:

```
procAllocators["test"]= new RATProcAllocatorTmpl<TestProc>;
```

Also, the header files seem to be included at the top of ProcBlockManager.cc.

2.4.4 Update Documentation

Finally, don't forget to document your processor by adding it to the User's Guide in $RATROOT/doc. Copy one of the existing manual sections for an example of what information to include. If your processor is particularly complicated, it's a good idea to also add a Discussion section to the end of the manual page that goes into detail on your method, algorithms, assumptions, etc. In most cases, a user should be able to figure out if your processor does what they want without reading your source code.

2.5 Adding a New Gsim Generator

If you find that the existing event generators are insufficient for your needs, then you will have to write a new event generator for Gsim.

First you need to ask yourself: Do you need to control some combination of momentum, position or timing simultaneously? If not, then you will be able to use the combo generator as your top-level generator, and just implement a vertex, position or time generator as described below. If your problem does not factorize in those independent parts, then you will need to implement a new top-level generator.

2.5.1 Writing a top-level generator

Top level generators are subclasses of GLG4Gen. By convention, the name of the subclass should be RAT::Gen_XXXX, where XXXX is some short identifier of your generator.

You will need to implement the following methods:

- IsRepeatable() - Returns true if this event generator should be used to generate more than one event. This is almost certainly the case for any generator you write. If this is false, the generator will be used to generate one event in the simulation and then deleted.
- SetState()/GetState() - Read a string passed by the user for general configuration when they run /generator/add. GetState() should return a string in the same form that SetState() accepts.
- ResetTime(double offset) - Pick a new time for the event to occur relative to the given offset. offset should be considered ''now'', and the new time is selected relative to it. It must be >= offset. Store the new time in the protected member variable nextTime.
- GenerateEvent() - This adds G4PrimaryVertex objects which contain the G4PrimaryParticle objects for this event. Called once at the beginning of each event.

Optionally, you can also override:

- SetVertexState() - called when user runs /generator/vtx/set
- SetPosState() - called when user runs /generator/pos/set
- SetTimeState() - called when user runs /generator/rate/set

This is not required if no further customization is needed beyond your normal SetState() method.

Next, you will need to edit RAT::Gsim::Init() to register your new generator with the ''generator factory'':

```
GLG4Factory<GLG4Gen>::Register("cosmic",
                               new GLG4AllocImpl<GLG4Gen, Gen_Cosmic>);
```

Don't forget to include your class header file at the top of Gsim.cc.

Now you will be able to invoke your generator with the command:

```
/generator/add cosmic parameters etc etc
```

2.5.2 Writing a vertex generator

Vertex generators pick the particles and their momenta and polarization. The assumption is that some other generator has been used to pick the position and time.

Vertex generators are subclasses of GLG4VertexGen. By convention, the name of the subclass should be RAT::VertexGen_XXXX, where XXXX is some short identifier of your generator. You will need to implement the methods:

- GeneratePrimaryVertex(G4Event* argEvent, G4ThreeVector& dx, G4double dt) - Add vertices to argEvent with position and time offset to dx and dt.
- SetState()/GetState() - control the generator config, usually called by /generator/vtx/set.

The new generator class is registered with the vertex generator factory in RAT::Gsim::Init():

```
GLG4Factory<GLG4VertexGen>::Register("betadecay",
    new GLG4AllocImpl<GLG4VertexGen, RAT::VertexGen_BetaDecay>);
```

2.5.3 Writing a position generator

Position generators pick the locations of events in the detector.

Position generators are subclasses of GLG4PosGen. By convention, the name of the subclass should be RAT::PosGen_XXXX, where XXXX is some short identifier of your generator. You will need to implement the methods:

- GeneratePosition(G4ThreeVector& argResult) - Assign a new event position to argResult.
- SetState()/GetState() - control the generator config, usually called by /generator/pos/set.

The new generator class is registered with the position generator factory in RAT::Gsim::Init():

```
GLG4Factory<GLG4PosGen>::Register("plane",
    new GLG4AllocImpl<GLG4PosGen, RAT::PosGen_Plane>);
```

2.5.4 Writing a time generator

Time generators pick the time interval between events. (Only from the same generator instance, time between events from different instances cannot be controlled).

Time generators are subclasses of GLG4TimeGen. By convention, the name of the subclass should be RAT::TimeGen_XXXX, where XXXX is some short identifier of your generator. You will need to implement the methods:

- GenerateEventTime(G4double offset) - Return a new time until the next event (in ns). Offset defines "now" so the returned time should never be less than offset.
- SetState()/GetState() - control the generator config, usually called by /generator/rate/set.

The new generator class is registered with the time generator factory in RAT::Gsim::Init():

```
GLG4Factory<GLG4TimeGen>::Register("burst",
    new GLG4AllocImpl<GLG4TimeGen, RAT::TimeGen_Burst>);
```

2.6 Extra Utilities

2.6.1 SimulatedAnnealing

Included in `src/util/SimulatedAnnealing.hh` is a general purpose algorithm for globally minimizing a D dimensional continuous function that contains many local minima. Effectively this is a hybrid between the Nelder-Mead downhill simplex method and simulated annealing as described in section 10.9 of `Numerical Recipes in 'C'` but reimplemented in a less confusing way for C++.

Usage

The `SimulatedAnnealing` class is templated to the dimensionality of function to be minimized. The constructor

```
SimulatedAnnealing(Minimizable *func)
```

takes a single argument that is an object implementing the pure virtual `Minimizable` class.

`Minimizable` only requires that the object implement the call operator as follows

```
virtual double operator()(double *args);
```

where the result is the value of the function evaluated at function(args[0],...,args[D]).

After creating the `SimulatedAnnealing` object, set the initial simplex using

```
void SetSimplexPoint(size_t pt, std::vector<double> &point)
```

where `pt` specifies the index of the point you are setting [0, D], i.e. D+1 required points, and the simplex is copied from the D length vector `point`.

Once the initial D+1 simplex points are specified, call the `Anneal` method to actually minimize

```
void Anneal(double temp0, size_t nAnneal, size_t nEval, double alpha)
```

where `temp0` is the initial temperature and `alpha` controls the temperature according to the annealing schedule T = `temp0` * (1 - `cycle` / `nAnneal`) ^ `alpha`. The annealing is run at `nAnneal` different temperatures (`cycle`) where each `cycle` tests `nEval` new points.

Once the algorithm has finished `GetBestPoint` will return the tested point with the lowest function value in the last `Anneal` cycle

```
void GetBestPoint(std::vector<double> &point)
```

where the best point will be copied into the D length vector `point`.

Example

As a concrete example, to minimize f(x,y) = x^A+y^B with A=B=2, first implement the `Minimizable` function

```
class Func : public Minimizable {
    public:
    double A,B;
    Func(double _A, double _B) : A(_A), B(_B) { };
    virtual double operator()(double *params) {
        return pow(params[0],A) + pow(params[1],B);
    }
}
```

Then the code to minimize would look something like this

```
Func func(2.0,2.0); //A = B = 2.0
SimulatedAnnealing<2> anneal(func);

vector<double> point(2), seed(2);

seed[0] = seed[1] = -1.0;
anneal.SetSimplexPoint(0,seed); // Point0 -> (-1,-1)
seed[0] = 1.0;
anneal.SetSimplexPoint(1,seed); // Point1 -> (1,-1)
seed[1] = 1.0;
anneal.SetSimplexPoint(2,seed); // Point2 -> (1,1)

anneal.Anneal(10,150,50,4.0); // Minimize

anneal.GetBestPoint(point);
```

(continues on next page)

(continued from previous page)

```
cout << point[0] << ',' << point[1] << endl;
```

2.7 C++ Style Guidelines

C++ coding style debates are a never-ending source of flamewars, so it's best to keep guides like this short. For some really interesting discussion of C++ style, take a look at these two books:

- Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd Edition by Scott Meyers
- C++ Coding Standards by Herb Sutter and Andrei Alexandrescu

That said, there are a few additional things to emphasize for RAT.

2.7.1 Filenames

1. Class declarations should be placed in header files in the relevant subdirectory of src. Header filenames should end with .hh and be protected against multiple inclusion with the standard #ifndef/#define trick. The convention is to #define __namespace_class__; for example #define __RAT_CountProc__ or #define __RAT_DS_PMT__.
2. Class definitions (when needed) should placed in the same subdirectory of src. Source filenames should end with .cc.
3. When possible, the both the declaration and definition files should have the same name as the class they contain.

2.7.2 Identifiers

1. Class names and method names should be written in CamelCase (acronyms like RAT can stay all upper case).
2. Class member variables should be written in CamelCase, but with an initial "f", as in fNhitThreshold, fExample.
3. Local variable names should be written in lowerCamelCase.
4. Classes should be defined under namespace RAT.
5. These rules do not apply to classes used in the data structure.

2.7.3 Data Structure Classes

1. They must be subclasses of TObject.
2. Member variables should be in lowerCamelCase.

2.7.4 Comments

1. A Doxygen-compatible comment with both brief and full description should precede each class declaration in a header file. 2. A brief comment should precede each method declaration (public, protected, or private) in the header file explaining how to use the method. If there is a method definition in a source file, it should be preceded by a detailed comment giving further information on how to use the method and explaining the overall implementation.

2.7.5 General Tips

Avoid ROOT collection classes

As anyone who has tried to use them can attest, the ROOT collection classes (like TObjArray and TClonesArray) are extremely awkward to use. This is because they are trying to tackle the problem of storing different kinds of objects in the same list, and so make you use pointers everywhere. Anything you want to put in a collection has to be a subclass of TObject, and when you extract an item from the collection, you have to recast back to whatever kind of object you thought you had in the array. This is all very messy and prone to error, and it makes you use pointers more than you should (see later item).

Instead, use the Standard Template Library (STL). It provides arrays, linked lists, maps, etc., all ready to go. Note that if you really do need a list of heterogenous objects, you are still better off using a vector<> of pointers to a common base class rather than using the ROOT collections, which force you to make the very complicated TObject the base of your class hierarchy.)

Avoid C-style arrays

C-style arrays are problematic because:

- They require you to separately know the size of the array. (Especially annoying if you need to pass the array to another function.)
- You can't resize them.
- The C++ standard does not allow you to have variable-sized automatic arrays:

```
void do_something useful(int number_of_foos) {
  int foo[number_of_foos];  // ILLEGAL
  // ... etc ...
}
```

Much as before, STL vectors work just like arrays, remember their size, and allow things like:

```
void do_something useful(int number_of_foos) {
  vector<int> bar(number_of_foos);
  vector<int> baz(bar.size() * 2);
  // ... etc ...
}
```

2.7.6 Avoid pointers when you can

Pointers are both essential in C++ and the thing most likely to make you crazy (a bad combination). Thankfully, in many cases, pointers are completely unnecessary:

- Pass by reference - If you want pass a large object to a function but don't want it to be copied, use the & operator in the function declaration:

```
int foo(MyObject &obj)  {
  // Do something
}
```

- Passing arrays - Use the vector<> template and pass it by reference.

Basically, you only need pointers when you must allocate new memory from the heap or need to manipulate a derived class without knowing which derived class it is (ex: a pointer to a RAT::Processor when you don't know which

particular processor it is). If you pass pointers between functions, be sure to make it clear who "owns" the object the pointer points to, and thus who is responsible for eventually deleting it.

CHAPTER 3

Authors

- Tim Bolton
- Dan Gastler
- Josh Klein
- Hugh Lippincott
- Andy Mastbaum
- James Nikkel
- Gabriel Orebi Gann
- Michael Akashi-Ronquest
- Stan Seibert
- Stephen Sekula
- William Seligman
- Chris Tunnell
- Matthew Worcester

YOUR NAME COULD BE HERE!

CHAPTER 4

Acknowledgements

- The SNO Collaboration – Much of the design of RAT is inspired by SNOMAN, the SNO Monte Carlo and ANalysis program.
- Glenn Horton-Smith – RAT uses GLG4sim as the basis for its Monte Carlo processor.
- The Double-CHOOZ Collaboration – We use their Gd capture simulation in the Monte Carlo

CHAPTER 5

Indices and tables

- genindex
- modindex
- search