RAT Documentation

Release 1.0.0

S. Seibert et al.

May 29, 2026

CONTENTS

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

Ratpac-two is hosted on GitHub. For information on accessing and working with the code using Git and GitHub, see `Using GitHub with Ratpac Code`.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 User's Guide

Table of Contents

1.1.1 Overview

All source code, documentation, and examples can be found at the following locations:

- github.com/rat-pac/ratpac-two.git
- ratpac.readthedocs.io
- hub.docker.com/repositories/ratpac
- github.com/rat-pac/ratpac-setup.git

Goals

Ratpac is intended to be a framework that combines both Monte Carlo simulation 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 efforts with a minimum of code duplication.

Ratpac-two is a refactor of ratpac which makes the necessary changes to compile with modern versions of GCC and is compatible with the latest versions of Geant4 and ROOT. This version is not backwards compatible with the previous version.

Design

The overall design of RAT is much like SNOMAN (the original SNO Monte Carlo package): 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.

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:

$$\bar{\nu}_e + p \rightarrow n + e^+$$

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 Ratpac 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 Ratpac. The events should be dumped to disk and analyzed in some other program.

Architecture

Ratpac-two is structured in a way to be fully extensible without requiring recompilation or modification of the base code. The core structure is designed to work with a CMake build system and compiles to a shared library which can be loaded via `find_package(Ratpac)`. For large scale application it is suggested that the user define a complete experiment package which includes the Ratpac library as a dependency. An example of such implementation can be found at

- github.com/rat-pac/RatpacExperiment

For smaller scale applications, the user can simply use geometry and configuration files, which are loaded at runtime, to define and experiment and run the simulation with the stand-alone executable—`rat`—along with any user defined macro files.

Relationships with Other Software

Ratpac 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. The version of GLG4sim used by Ratpac is heavily modified from the original.

Modular Software Extensions

Additional software packages can be used to extend the functionality of Ratpac. These packages are not required to compile Ratpac, but can be included at compile time to enable external physics generators and analysis capabilities.

- Cry — Cosmic-ray particle shower generator
- Chroma — Optical photon propagation and detection

- Tensorflow / cppflow — Enable tensorflow model evaluation event-by-event during simulations.
- PyTorch — Enable PyTorch model evaluation event-by-event during simulations.

1.1.2 Installation

Prerequisites

These software packages should be installed in the order presented before you attempt to build Ratpac.

- Python 3+ with development headers -
- ROOT 6.25+
- Geant-4 11.0+
- CMake 3.22+

Build Steps

A helper package exists at ratpac-setup to help with the installation of Ratpac and its dependencies. This package is recommended for general installation and will install a local version of ROOT and Geant4.

Containers

Building Containers

ratpac can be built in a container using the provided dockerfiles found in both ratpac-two itself and ratpac-setup. The Dockerfile provided by ratpac-setup creates a container with a all of ratpac-two's dependencies installed, while the Dockerfile in ratpac-two creates a container with ratpac-two itself installed. The container is based on the ubuntu 22.04 image and includes all of the dependencies required to build ratpac. the container can be built using the following command:

The first command will build the base image which only includes the dependencies but not the Ratpac source code. The second command will build the Ratpac image which includes the Ratpac source code and uses ratpac/ratpac-two:latest-base.

A Singularity (Apptainer) image can be built from the Docker image using the following command:

```
singularity build ratpac-two.sif docker-daemon://ratpac/ratpac-two:latest
```

Or if you are planning to develop Ratpac, you can build the image only from the base docker image and then compile Ratpac inside the container.

```
singularity build ratpac-two.sif docker-daemon://ratpac/ratpac-two:latest-base
```

Pulling containers from Github CI

The ratpac-two repository also creates nightly builds of the latest commit to the repository.

```
# Base container:
apptainer build ratpac-two-base.sif docker://ratpac/ratpac-two:latest-base
# Full container:
apptainer build ratpac-two.sif docker://ratpac/ratpac-two:main
```

Build from source

Using the convenience Makefile

A convenience Makefile exists to automate the above process, simply type *make*. Several targets are available:

- `make` or `make all`: Builds in installs ratpac-two to `install/` directory. Default to the `RelWithDebInfo` build type.
- `make debug/release/Minsize/relwithdebinfo`: Builds and installs ratpac-two with the specified build type.
- `make clean`: Removes the `build/` and `install/` directories.

Custom Configuration with CMake

Ratpac-two follows standard CMake conventions. The following is a step-by-step list CMake commands to compile:

```
mkdir build install
cd build
cmake .. -DCMAKEINSTALL_PREFIX=../install # Add any additional configuration flags here
make  # use -jN to parallelize with N threads
make install
```

1.1.3 Controlling ratpac-two via macro files

This section of the guide will provide a high-level overview of controlling ratpac-two, which is accomplished through the use of macro files. Macro files are plain text files containing a sequence of commands that control ratpac-two. Using macros, you can configure various aspects of a ratpac experiment, such as:

- Detector geometry, materials, and calibration constants (via RATDB).
- Data processing chain (processors for digitization, reconstruction, output, etc.).
- Data processing chain (processors for digitization, reconstruction, output, etc.).
- Primary particle sources (event generators).
- Run-specific parameters like the number of events and random number seeds.

Macro files are structured to follow your ratpac-two workflow. This is typically a **pipeline** connecting event **producers** to a sequence of **processors**. Each producer creates or loads an event and then passes it through the processors. Each processor can modify the event, record information, or simply observe it, and the order in which you declare them determines how every event will be handled. Ignoring the finer details of event pileup scenarios, the conceptual operation of ratpac-two looks like this:

```
Configuration of ratpac parameters (RATDB, etc.)

Producer 1:
Event 1 -> Processor 1 -> Processor 2 -> ... -> Processor N
Event 2 -> Processor 1 -> Processor 2 -> ... -> Processor N
...
Event M ->  Processor 1 -> Processor 2 -> ... -> Processor N

Producer 2:
Event 1 -> Processor 1 -> Processor 2 -> ... -> Processor N
...
```

First, we will briefly review ratpac-two commands and their syntax. Then we will work through an example synthesizing multiple commands into a macro file. A comprehensive understanding requires extensive knowledge of the software. Revisiting this guide is recommended as you read through the rest of the documentation.

Table of Contents

1. ratpac-two Commands and Workflow

After successfully installing ratpac-two and loading the associated environment, ratpac-two is accessed via a command line interface by running the `rat` command. `rat` can execute a sequence of macro files if multiple are provided on the command line.

```
rat example.mac
```

This will execute `example.mac`

In addition to macro files, ratpac-two can be configured with various flags, which can be accessed by running `rat` with the `--help` flag

```
rat --help

options:
 -b, --database          URL to database
 -d, --debug             Enable debug printing
 -i, --input             Set default input filename
 -l, --log               Set log filename
 -o, --output            Set default output filename
 -p, --python            Set python processors
 -q, --quiet             Quiet mode, only show warnings
 -r, --run               Simulated run number
 -s, --seed              Set random number seed
 -x, --vector            Set default vector filename
 -v, --verbose           Enable verbose printing
 -V, --version           Show program version and exit
 -g, --vis               Load G4UI visualization
```

We will start our discussion of controlling ratpac-two by inspecting the structure of a single command.

1.1 Command Syntax

ratpac-two inherits its command interface from Geant4's G4RunManager. G4 command reference

The syntax rules are straightforward:

- **Commands:** A command consists of a command name followed by zero or more parameters, separated by spaces. For example:

  `/rat/procset update 5`

  Here, `/rat/procset` is the command, and `update` and `5` are parameters.

- **Comments:** Lines beginning with a hash symbol `#` are treated as comments and are ignored by the interpreter. They are essential for documenting macros.
  Comments can also be used inline with commands, e.g.

  `/run/beamOn 10 #Run 10 events`

- **Blank Lines:** Blank lines are also ignored and can be used to improve macro readability.

- **No Leading Whitespace:** Commands must start at column 1. Any spaces or tabs before the command cause an "unknown command" error. For example:

```
# Note the additional whitespace below
    /run/beamOn 1
```

Running a macro with this indentation will produce an "unknown command" error.

1.2. Command Hierarchy

Commands in ratpac-two (and Geant4) are organized into a hierarchical structure, much like a filesystem. This structure helps in organizing commands logically by function. For example:

- `/run/` commands control aspects of Geant4 simulation runs (e.g., `/run/initialize`, `/run/beamOn`).
- `/generator/` commands control generation of particle sources and their properties (e.g., `/generator/add`, `/generator/vtx/set`).
- `/rat/` is the base directory for all commands specific to ratpac-two functionalities. Within `/rat/`, further subdirectories exist, such as:
  - `/rat/db/` for RAT Database interactions.
  - `/rat/proc/` (or `/rat/procset/`) for managing data processing modules (processors).

1.3. Command Execution Order

ratpac-two will execute the commands in a macro file sequentially from top to bottom. The exact ordering of commands will depend on the specific task to be accomplished by ratpac-two. In the next section, we will provide an example macro outlining a workflow suitable for detector simulations to illustrate usage.

Macro files can include other macro files using the `/control/execute` command. For example, in your macro file you can add:

```
/control/execute common_detector_setup.mac
```

When this command is encountered, the specified macro file (common_detector_setup.mac in this case) is immediately read and its commands are executed in place before continuing with the original macro. This feature is extremely useful for modularizing simulation configurations. Nested macros are allowed. A file invoked with `/control/execute` can itself contain additional `/control/execute` commands, and each included file is processed immediately when encountered. If the file cannot be found, Geant4 prints an error and stops executing the current macro.

Users can create libraries of common settings, such as standard detector geometry definitions or output format configurations. Specific simulation scenarios can then be composed by including these base macros and then overriding or adding specific parameters as needed. This approach promotes reusability, consistency across different studies, and better organization of complex simulation setups, mirroring how functions or modules are used in programming to avoid code duplication and improve maintainability.

A minimal macro might contain just a few commands that cannot run independently without other setup commands:

```
# mini.mac
/rat/proc noise
/rat/proc splitevdaq
```

1.4 Example Macro

Now let's take a look at an example macro file. This macro uses the internal simulation tools in ratpac-two to produce 2 MeV electron interactions in the example geometry. It instantiates a series of processors that add Poisson dark noise to photosensors, simulate readout electronics, and save the processed output in the `outntuple` format. This example serves as a template for how one might structure a simulation macro:

The macro is broken down into:

1. ratpac configuration (Physics configurations & RATDB definitions)
2. `/run/initialize`
3. Logical ordering of processors
4. Generator definitions

NOTE Processors are specified **before** generators.

Example macro:

```
#------------------------------------------------------------------------------
# Informational Header
#------------------------------------------------------------------------------
# example.mac - An example macro file for this tutorial
# ratpac-two Tutorial
# Date (YYYY-MM-DD): 2025-06-01

#------------------------------------------------------------------------------
# Physics Settings
#------------------------------------------------------------------------------

# Example commands that allow us to select what physics processes we would like
# to simulate.  For example, 2 MeV electrons do not require muon and hadronic
# physics processes

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

# We disable particle tracking in this example to save on RAM during runtime.
/tracking/verbose 0

#------------------------------------------------------------------------------
# RATDB Configuration
#------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
# Set your experiment (explained later in the guide)
/rat/db/set DETECTOR experiment "Validation"
# For this example we will use the ratpac validation geometry
/rat/db/set DETECTOR geo_file "Validation/Valid.geo"

#------------------------------------------------------------------------
# Initialize the Simulation
#------------------------------------------------------------------------

#This command 'locks in' the geometry and parameters above for the simulation
/run/initialize

#------------------------------------------------------------------------
# Add Event Processors
#------------------------------------------------------------------------

# Next we add our processors.
# Simulate Poisson dark noise in the PMTs
/rat/proc noise
# Simulate the DAQ and waveform digitization with trigger.
/rat/proc splitevdaq
/rat/procset trigger_threshold 1.0
# Print the event count every 10 events.
/rat/proc count
/rat/procset update 10
# Save the results in a root file using outntuple
/rat/proc outntuple

#------------------------------------------------------------------------
# Add Event Generators
#------------------------------------------------------------------------

#Generators are specified after processors; here we use a basic particle gun
/generator/add combo gun:point:poisson
# 2.0 MeV electrons at the center of the detector along the negative z axis
/generator/vtx/set e- 0.0 0.0 -1.0 2.0
/generator/pos/set 0.0 0.0 0.0

##### RUN ############
# Run the event generator 100 times
/run/beamOn 100
# End of macro
```

Let's discuss the format and styling of this macro by providing additional detail on some of the most important commands above

2. Essential Geant4 Commands for ratpac-two Simulation

Since ratpac-two is built upon the Geant4 toolkit, a wide array of standard Geant4 User Interface (UI) commands are available and often essential for controlling simulations. For users new to Geant4, understanding these core commands is a prerequisite for effectively using ratpac-two. This section focuses on the most commonly used Geant4 commands relevant in the context of a ratpac-two simulation.

2.1. Run Control

These commands manage the initialization and execution of simulation runs and dictate the order of commands passed to ratpac-two in a macro.

- `/run/initialize`: This is one of the most critical commands in any Geant4-based simulation. It must be issued before the first run can start. This command triggers the construction of the detector geometry, calculation of physics tables, and preparation of user actions. In ratpac-two, it ensures that all detector parameters from RATDB are processed and the simulation world is built.

  The typical placement of this command is after setting up detector parameters (e.g., via `/rat/db/` commands) but before defining event processors or starting the event loop.

  Note: The run manager must be initialized *after* the geometry has been fully configured and *before* issuing any `/run/beamOn` command. Calling `/run/beamOn` prior to `/run/initialize` will cause Geant4 to halt with an error similar to `G4RunManager::BeamOn() called before initialization`.

- `/run/beamOn <numberOfEvents>`: This command starts a simulation run, processing the specified number of events.

  Multiple `/run/beamOn` commands can appear in a single macro, for instance, to simulate different particle types or energies sequentially after reconfiguring the event generator.

2.2. Verbosity Control

Geant4 provides extensive control over the amount of information printed during a simulation. These commands are invaluable for debugging and understanding the simulation process. Verbosity is typically controlled by an integer level, where 0 means minimal output and higher values provide more detail.

- `/run/verbose <level>`: Controls the verbosity of the run manager (e.g., messages about run initialization and termination).
- `/event/verbose <level>`: Controls verbosity related to event processing (e.g., information at the beginning and end of each event).
- `/tracking/verbose <level>`: Provides detailed information about particle tracking, including step-by-step details of particle interactions and movement through the geometry. This is particularly useful for debugging physics processes or identifying issues with the detector geometry.

By strategically increasing verbosity for specific components like tracking or particular physics processes, new users can gain a much clearer picture of what Geant4 (and by extension ratpac-two) is doing "under the hood" for each event. Observing this textual output can be more illuminating for learning than just examining final results or visualizations, as it connects abstract concepts of particle interactions and geometry definitions to concrete simulation steps.

2.3. Detector Visualization

Basic visualization commands let you quickly open a viewer and inspect the geometry. Note that to successfully run the visualizer, you must start rat with the `--vis` flag, e.g. `rat visualization.mac --vis`. Without this option rat may exit with an error when the visualization commands are executed. Useful commands include:

- `/vis/open OGL` - open an OpenGL viewer.
- `/vis/drawVolume` - draw the full detector geometry.
- `/vis/viewer/update` - refresh the current viewer after changes.
- `/glg4vis/reset` - reset the viewpoint to default settings.
- `/glg4vis/upvector x y z` - choose the "up" direction for the viewer.

2.4. Other Useful UI Commands

- `/control/execute <macroFile>`: As mentioned in Section 1.3, this command executes another macro file. It is a standard Geant4 command.
- `/control/loop <macroFile> <counterName> <initialValue> <finalValue> <stepSize>`: This command allows for looping. The specified is executed multiple times. In each iteration, the Geant4 UI variable is set to values from to with an increment of . The can then be used within (e.g., `{counterName}`).
- `/control/foreach <macroFile> <variableName> <valueList>`: Similar to `/control/loop`, but iterates over a discrete list of values provided in (space-separated). The is set to each value in the list for each execution of . The availability of Geant4's looping commands (`/control/loop`, `/control/foreach`) directly within the macro system allows users to perform simple parameter scans (e.g., varying particle energy, source position, or even a RATDB parameter if set within the looped macro) without resorting to external scripting. This is a powerful built-in feature for conducting systematic studies efficiently.
- `exit`: Terminates an interactive ratpac-two session.

The following table summarizes some of the most common Geant4 UI commands useful for ratpac-two users.

Table 2.1: Common Geant4 UI Commands for ratpac-two

Command	Typical Parameters	Description	Example Usage in ratpac-two Context
`/run/initialize`	(none)	Initializes detector geometry, physics lists, and run conditions.	`/run/initialize`
`/run/beamOn`	`<numberOfEve`	Starts a simulation run for the specified number of events.	`/run/beamOn 1000`
`/run/verbose`	`<level>`	Sets the verbosity level for run-related messages (0 = quiet).	`/run/verbose 1`
`/event/verbose`	`<level>`	Sets the verbosity level for event-related messages.	`/event/verbose 1`
`/tracking/verbose`	`<level>`	Sets the verbosity level for particle-tracking messages.	`/tracking/verbose 1`
`/control/execute`	`<filename>`	Executes commands from the specified macro file.	`/control/execute detector_setup.mac`
`exit`	(none)	Exits an interactive ratpac-two session.	`exit`

This set of commands provides a foundational toolkit for basic simulation control within ratpac-two.

3. Navigating ratpac-two Specific Commands: The /rat/ Directory

While ratpac-two leverages the standard Geant4 command interface, it also introduces a suite of custom commands to manage its unique features and functionalities. These ratpac-two-specific commands are neatly organized under the `/rat/` command directory. This clear separation into the `/rat/` namespace is a deliberate design choice that prevents potential conflicts with standard Geant4 commands or commands from other Geant4-based applications. Such organization makes the command space more predictable and manageable for users and developers alike.

Commands within the `/rat/` directory provide fine-grained control over various components of the ratpac-two framework, including its parameter database (RATDB), the sequence of data processing modules (processors), specialized event generators, custom physics configurations, and I/O settings. The internal structure of subdirectories within `/rat/` (e.g., `/rat/db`, `/rat/proc`) often mirrors the modular C++ class structure within the ratpac-two source code.

For example, commands found in `/rat/db/` are typically implemented by messenger classes associated with RAT::DB or similar database management classes in the C++ backend. An awareness of this correlation can be beneficial for

advanced users or those delving into the source code to understand command implementations. Developers extending the command interface typically create a messenger class whose name mirrors the subdirectory.

Following this pattern, a new `/rat/foo/` directory would pair with `FooMessenger` in `src/cmd/src/FooMessenger.cc`, keeping the command layout synchronized with the source tree. The following table gives a high-level overview of the common subdirectories expected within `/rat/` and their primary functions. The exact commands and their detailed functionalities are determined by analyzing the ratpac-two source code. Subsequent sections of this guide will elaborate on commands within these key areas.

Table 3.1: Overview of Key /rat/ Command Subdirectories

Sub-directory	Primary Function	Key Commands	Example Usage
`/rat/db/`	Manages interaction with the RAT Database (RATDB), including loading and modifying detector parameters.	`/rat/db/load`, `/rat/db/set`, `/rat/db/server`, `/rat/db/run`	`/rat/db/load myfile.ratdb`
`/rat/proc/`	Controls the chain of ratpac-two processors (data processing modules) and their configurations.	`/rat/proc`, `/rat/procset`	`/rat/proc MyProcessor`
`/rat/physics`	Configures ratpac-two specific physics options, custom physics lists, or specialized physics processes.	`/rat/physics/add`, `/rat/physics/select`	`/rat/physics/enableCerenkov true`
(others)	Other specialized control directories as defined within the ratpac-two framework for specific components.	(dependent on ratpac-two features)	(varies)

The `/rat/db/load` command is implemented by `DBMessenger` in `src/cmd/src/DBMessenger.cc`.

Understanding this top-level structure helps users navigate to the relevant command set for the aspect of the simulation they wish to control.

For instance, to modify a material property, one would look for commands under `/rat/db/`. To add a new analysis step to the event processing, commands under `/rat/proc/` would be relevant.

4. Configuring the ratpac-two Environment with RATDB via Macros

The RAT Database (RATDB) is a cornerstone of ratpac-two's flexibility, serving as a centralized repository for all parameters that define the simulated experiment using JSON-like structure. This includes detailed descriptions of detector geometry, material properties, optical parameters (like refractive indices, absorption lengths, Rayleigh scattering lengths), PMT characteristics (quantum efficiency, transit time spread), electronics and DAQ settings, and even fundamental physics constants. The ability for analysis code to trivially access the same detector geometry and physics parameters used in the detailed simulation is a key design feature facilitated by RATDB. This comprehensive parameterization is crucial for achieving "As Microphysical as Reasonably Achievable" (AMARA) simulations.

RATDB parameters are typically stored in external files, often using a like JSON format, which is both human-readable and easily parsable. The choice of JSON for RATDB files offers significant advantages: users can inspect or manually tweak parameters using standard text editors, and the structured nature of JSON (key-value pairs, arrays, nested objects) is well-suited for representing complex detector configurations. Furthermore, many programming languages have built-in JSON parsers, making it easier to integrate ratpac-two configurations with external scripts or analysis tools if necessary. Macros provide the interface to load these database files and, importantly, to modify specific parameters at runtime before the simulation initializes.

4.1. Key RATDB Macro Commands

The following commands, primarily under the /rat/db/ path, are used to interact with RATDB. The exact syntax and full range of commands can be found by inspecting the RAT::DB related messenger classes in the ratpac-two source code.

- /rat/db/load <filename> : This command loads data from the specified file into RATDB. Load material optical properties, e.g., /rat/db/load OPTICS.ratdb
- /rat/db/set <TABLE>[<index>] <field> <json_value>: Modify a value in the specified RATDB table. <TABLE> may include an index such as name:AV to select a particular entry. The last argument is interpreted as JSON and converted to the appropriate data type automatically. For example /rat/db/set MATERIALS name:LS LIGHT_YIELD 10000.0 /rat/db/set PMTPROPS pmt_model:R12345 QE [0.20,0.22,0.25, 0.22,0.20] The ability to modify RATDB parameters at runtime via /rat/db/set is extremely powerful for systematic studies. Users can employ macro control flow (like /control/loop) to iterate over different material properties, PMT efficiencies, or minor geometry variations without needing to create and manage numerous distinct RATDB files. This greatly streamlines the process of studying the impact of detector parameter uncertainties.
- /rat/db/server <url>: Connects to a remote CouchDB instance holding RATDB tables.
- /rat/db/run <run_number>: Sets the default run number for database lookups.

4.2. Timing of RATDB Commands

It is crucial to understand that RATDB parameters defining the detector geometry, fundamental material properties, or physics constants generally need to be set before the /run/initialize command is issued. The /run/initialize step uses the information in RATDB to construct the simulation world. Changes made to these fundamental parameters after initialization may not take effect or could lead to inconsistent states. The interaction between macros and RATDB highlights ratpac-two's flexibility. While default or baseline parameters are loaded from files (which might be shared and version-controlled within a collaboration), macros provide a dynamic overlay. This allows users to make temporary changes or specific adjustments for a given simulation run without altering the underlying database files, which is particularly beneficial in collaborative environments.

Table 4.1: Essential RATDB Macro Commands

Task	Command Syntax (Conceptual)	Example
Load a RATDB file	/rat/db/load <filename.ratdb>	/rat/db/load special_definitions.ratdb
Set a scalar parameter in a table	/rat/db/set <TABLE>[<idx>] <param> <json_value>	/rat/db/set MATERIALS name:Acrylic DENSITY 1.19
Set an array parameter in a table	/rat/db/set <TABLE>[<idx>] <param> <json_array>	/rat/db/set OPTICS name:Water RAYLEIGH_LENGTH [30000.,35000.,40000.]
Connect to CouchDB server	/rat/db/server <url>	/rat/db/server http://localhost:5984
Set default run number	/rat/db/run <run_number>	/rat/db/run 42

5. Orchestrating the Workflow: Processor Commands

In ratpac-two, the concept of "processors" is central to defining the simulation and analysis workflow for each event. Processors are modular units of C++ code, each designed to perform a specific task. The sequence of these processors, assembled by the user in a macro file, dictates the entire chain of operations applied to every event generated. This user-defined chain effectively forms a data processing pipeline.

Examples of tasks handled by processors include:

- Collecting and processing PMT hits.
- Simulating electronics response and digitization.
- Simulating trigger logic.
- Performing event reconstruction (e.g., vertex fitting, energy estimation).
- Writing output data to files (e.g., ROOT ntuples).

The order in which processors are added to the chain is critical, as they typically operate on the output of preceding processors. For instance, PMT digitization must occur after optical photons have been tracked to the PMTs, and event reconstruction generally requires digitized data.

5.1. Key Processor Macro Commands

Processors are controlled with a small set of macro commands:

- `/rat/proc <ProcessorName>` adds the named processor to the processing chain. The order of these commands determines the execution order. Use `/rat/proclast <ProcessorName>` to place a processor after any that were specified on the command line.
- `/rat/procset <parameter> <value>` sets a parameter on the most recently added processor. `ProcBlockManager` only allows `procset` after a successful `proc` command. For example, `/rat/proc noise` followed by `/rat/procset update 100` configures the `noise` processor to report every 100 events.

It is important to note that ratpac-two is designed to be extensible, allowing users and collaborations to develop their own custom processors in C++ to address unique experimental requirements or implement new analysis algorithms. While the development of such custom processors is beyond the scope of the user guide, the macro commands for adding and configuring processors are designed to work uniformly with both standard and custom-developed modules. This extensibility means the list of available processors can grow beyond the set shipped with the core ratpac-two distribution.

5.2. Typical Processor Chain Example

A conceptual processor chain for a typical simulation might look like this:

- mc_particle_tracking: (Assumed processor for Geant4 tracking of primary and secondary particles).
- pmt_response: Simulates PMT hit generation from optical photons.
- daq_electronics: Simulates front-end electronics, digitization, and waveform generation.
- trigger: Applies a trigger logic to the digitized data.
- reconstructor_bonsai: Performs event reconstruction using an algorithm like BONSAI.
- output_ntuple: Writes selected event information (raw data, reconstructed quantities) to an output file.

A macro would typically contain a series of `/rat/proc` and `/rat/procset` commands to build this chain.

5.3. Table of Processor Control Commands and Common Processors

The following table outlines the essential commands for processor control and lists some common types of processors one might expect in ratpac-two. These commands are implemented by `ProcBlockManager` in the source tree.

Table 5.1: Essential Processor Control Commands

Command / Item	Purpose / Description	Example
/rat/proc	Adds a processor to the event processing chain.	/rat/proc noise
/rat/proclast	Adds a processor after all other processors specified.	/rat/proclast outroot
/rat/procset	Sets a parameter for the most recently added processor.	/rat/procset update 5

This structure allows users to build customized simulation and analysis workflows by selecting and configuring the appropriate sequence of processors.

6. Defining Event Sources: The Generator Commands

Event generators in ratpac-two are responsible for defining the initial state of particles that the simulation will track. This includes their type, energy, momentum, starting position, and time of creation. ratpac-two provides a flexible system for event generation, controlled by commands primarily under the /generator/ path. The available commands are defined in the source tree (e.g. GLG4PrimaryGeneratorMessenger and the various generator classes). The examples below are verified directly against those implementations.

6.1. Key Generator Macro Commands

- /generator/add <generator_type> [options_string]: This is the primary command for activating a new event generator instance.
  - <generator_type>: A string identifying the kind of generator to add. Examples might include:
    - gun: For a simple particle gun.
    - combo: A versatile generator that allows piecing together separate vertex, position, and time generators.
    - isotopes: For simulating radioactive decays from specific isotopes or decay chains.
    - supernova: For simulating bursts of neutrinos from a supernova.
    - hepevt or lhe: For reading particle events from external files in standard formats.
    - (The actual list of available types must be determined from the ratpac-two source code.)
  - [options_string]: An optional string passed directly to the generator for its internal configuration. For the combo generator, this string might specify the sub-generators to use, e.g., gun:point:poisson (particle gun kinematics, point-like position, Poisson time distribution). Example: Add a combo generator using a particle gun, point source, and Poisson time distribution /generator/add combo gun:point:poisson The combo generator, as suggested by the example, indicates a modular and highly flexible approach to defining event characteristics. Users can effectively mix and match different components (e.g., particle type/energy from a "gun2" model, position from a "point" or "volume" model, time distribution from a "poisson" or "fixed_interval" model) to construct complex event profiles simply by changing parts of the options_string or subsequent configuration commands.
- /generator/vtx/set <particle_name_or_PDGcode> [energy_value][dx][dy][dz]:
- /generator/vtx/set <args>: Set the vertex generator state for the most recently added generator. Argument details depend on the generator type. For the built-in gun vertex generator, the format is /generator/vtx/set pname momx_MeV momy_MeV momz_MeV KE_MeV [polx poly polz mult]. An isotropic 2 MeV electron can be produced with /generator/vtx/set e- 0 0 0 2.0.
- /generator/pos/set <x> <y> <z> or /generator/pos/set <position_generator_name>: Sets the position for event generation. For the built-in point position generator the arguments are three numbers interpreted in millimetres.

- `/generator/rate/set <rate>`: Sets the event rate for the current generator in events per second. This string is passed directly to the time generator (e.g. `GLG4TimeGen_Uniform::SetState`).

Example sequence for generating isotropic 2 MeV positrons at the detector center, note that setting a generator to 0 0 0 defaults to isotropic emission: `/generator/add combo gun:point:poisson /generator/vtx/set e+ 0 0 0 2.0 /generator/pos/set 0 0 0 /generator/rate/set 1.0` Commands like `/generator/vtx/set`, `/generator/pos/set`, and `/generator/rate/set` apply to the generator most recently added with `/generator/add`. This contextual application is a common pattern in such command interfaces.

6.2. Multiple Generators

ratpac-two supports the use of multiple generators simultaneously. Events from different active generators will be interleaved according to their specified rates, and can even pile up if they occur in coincidence (i.e., within the same event window). This capability is essential for simulating realistic experimental conditions where multiple physics processes or background sources contribute to the collected data. It allows for comprehensive studies of signal-to-background ratios, pile-up effects, and coincidence detection schemes.

6.3. Table of Common Generator Configurations

Given the identified gap in existing documentation for generator commands, the following table provides conceptual examples of how common event sources might be configured. The precise command names and parameters must be validated against the ratpac-two source code.

Table 6.1: Common Event Generator Configurations via Macros (Conceptual)

Task/Ge Type	Key `/generator/add` Command	Subsequent Configuration Commands	Example Macro Snippet	Brief Explanation
Single Particle Gun	`/generator/add gun`	`/generator/vtx/set <pname> px_MeV py_MeV pz_MeV KE_MeV` `/generator/pos/set <x> <y> <z>` `/generator/rate/set <R>`	`/generator/add gun` `/generator/vtx/set e- 0 0 1 5.0` `/generator/pos/set 0 0 0`	Fires a 5 MeV electron from the origin along +z.
Volumetric Isotope Decay	`/generator/add isotope <isotope_nam`	`/generator/rate/set <R>`	`/generator/add isotope Bi214:ScintillatorVolume` `/generator/rate/set 10`	Simulates Bi214 decays uniformly within the volume.
Combo Generator (Point Source)	`/generator/add combo gun:point:po`	`/generator/vtx/set <pname> px_MeV py_MeV pz_MeV KE_MeV` `/generator/pos/set <x> <y> <z>` `/generator/rate/set <R>`	`/generator/add combo gun:point:poisson` `/generator/vtx/set e- 0 0 0 2.0` `/generator/pos/set 0 0 500` `/generator/rate/set 1`	Generates isotropic 2 MeV electrons at z=500 mm with Poisson timing.
Read from HEPEVT file	`/generator/add hepevt <file.hepevt>`	`/generator/rate/set <R>` (optional)	`/generator/add hepevt myevents.dat`	Reads particle events sequentially from the file.

These examples illustrate the intended flexibility. Users should consult any example macros provided with a specific generator for validated syntax.

7. Conclusion and Further Learning

Macro files are the cornerstone of user interaction with ratpac-two, offering a powerful and flexible plain-text interface to control nearly every aspect of the simulation environment. From defining the fundamental detector parameters via RATDB interactions to specifying the types of physics events to generate and orchestrating the complex chain of data processing steps, macros empower users to tailor ratpac-two to their specific research needs.

Given that ratpac-two documentation is an evolving landscape, users are encouraged to explore the following resources:

- This Guide: Keep this document as a reference for macro commands and concepts.
- ratpac-two GitHub Repository: The primary source for the code, and potentially updated documentation or examples within its /doc directory or elsewhere. The source code itself, particularly the Messenger classes, is the definitive reference for command definitions.

Experimentation is key. Start with simple macros, make incremental changes, and use verbosity commands extensively to observe the effects of your commands. As familiarity grows, so will the ability to harness the full potential of ratpac-two for advanced particle physics simulations.

1.1.4 Event Producers vs. Event Processors

Ratpac 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 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
```

Ratpac 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, 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"

/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.

1.1.5 Data Structure (RAT::DS)

The event data structure is a tree of information about a particular event. Event producers create an instance of the data structure for each event, and processors can then operate on this structure, transforming it as desired. A single event is that which is generated when a macro calls */generator/beamOn*; however, a processor is free to break this event into multiple sub-events.

The `ratpac-two` Data Structure

The `ratpac-two` data structure is defined in the *src/ds* directory and lists everything under the `RAT::DS` namespace. An instance of the data structure is defined in the `RAT::DS::Root` object. The data structure is tree-like, with each instance of `RAT::DS::Root` containing a list of `RAT::DS::MC` objects which contain the Monte Carlo truth information, and a list of `RAT::DS::EV` objects which contain the reconstructed event information (usually after going through a processor that simulates the detector response).

RAT::DS::MC

RAT::DS::EV

The `ratpac-two` ntuple structure

`ratpac-two` provides multiple output file formats, the simplest of which is called the "ntuple" output. The details of this output file format are described in *outntuple*. This output file format is generated by accessing the `RAT::DS` to fill flat `ROOT` TTrees, which are smaller and easier to work with than the full `RAT::DS`.

1.1.6 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 many things, such as: * 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.

Syntax

The RATDB syntax is effectively JSON with comments (JSONC). Each RATDB file is the simple concatenation of a series of JSON key-value pairs. C-style comment strings (ones that start with `//` or are surrounded by `/*` and `*/`) are also accepted.

However, it is worthy of note that the parser used in RAT accepts looser syntax than standard JSON. Most notably, it allows the keys of JSON key/value pairs to be unquoted, and it allows for trailing commas before the closing brace of an object.

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 scintillator), 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 run plane
- the default plane

The plane in which a RATDB entry is valid is determined by the `run_range` or `run_list` specified. `run_range` takes in a vector of length 2, which specifies the beginning and end of the run range. `run_list` takes in a non-empty vector that specifies the list of run numbers the RATDB entry is valid for. If both `run_range` and `run_list` are provided, **RATDB will prioritize ``run_range``. The user plane is denoted with run number -1, and the default plane is denoted with run number 0.** Syntax that uses `valid_begin` and `valid_end` are now officially deprecated. They should be replaced with `run_range` or `run_list`.

When an item is requested, RATDB will attempt to locate it in the user plane first, then the run 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 retrieve the data.

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.

Loading experiment-specific RATDB tables

RATDB also has the ability to load experiment-specific tables. These tables need to be placed under `$RATSHARE/ratdb/experiment_name/` (`$RATSHARE` is typically the RAT root/installation directory). To load these tables, one needs to set the field `DETECTOR.experiment` to the name of the directory. Typically, this can be done via the macro like:

```
/rat/db/set DETECTOR experiment "SNO"
```

Note that these experiment-specific tables are always loaded last, meaning that these tables will override any other tables with the same name. This can be very useful when an experiment wants to override the default values in certain tables, without worrying about telling a processor how to read a table with a non-default index.

However, other than this particular override behavior, any other collision in table name and index will currently result in completely undefined behavior. This is true between tables placed in RATPAC-two and a private experiment. **If an override is required, always place the overriding table in the experiment-specific directory.**

1.1.7 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
```

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.

led

```
/generator/add led POSITION:TIME
```

Creates a new LED generator using the position and time generators. The LED generator can simulate any number of independent LEDs, each with a predefined wavelength spectrum, intensity, timing, and angular distribution. The LED generator can be controlled through its corresponding ratdb file `LED.ratdb`. The parameters of the ratdb file are described in the following table:

table:

```
{
    "name": "LED"
    "index": "default",
    "run_range": [0, 0],

    "intensity": 100,
    // When using multiples LEDs in sequence, (x, y, z) are their positions.
    "x": [1.0],
    "y": [1.0],
    "z": [1.0],
    "wavelength": [100.0, 200.0, 300.0, 400.0, 500.0, 600.0],
    // "intensity_mode": {"single", "chain"}
    "intensity_mode": "single",
    // "time_mode": {"unif", "dist"}
    "time_mode": "unif",
    // "angle_mode": {"iso", "dist", "multidist"}
    "angle_mode": "iso",
    // "wl_mode": {"mono", "dist"}
    "wl_mode": "mono",

    // Timing distribution if "dist"
    "dist_time": [0.0, 25.0, 50.0],
    "dist_time_intensity": [1.0, 2.0, 1.0],
```

(continues on next page)

(continued from previous page)

```
    // Angular distribution if !iso -- radians?
    "dist_angle": [0.0, 1.0],
    "dist_angle_intensity": [1.0, 1.0],
    // Angular distributions if multidist
    "n_ang_dists": 1,
    "dist_angle0": [0.0, 2.0],
    "dist_angle_intensity0": [1.0, 1.0],

    // Wavelength distribution if !mono
    "dist_wl": [300.0, 800.0],
    "dist_wl_intensity": [1.0, 1.0],
}
```

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.

rootracker

The RooTracker generator is used to take event vertices generated by GENIE and simulate them in RAT. GENIE GHEP files should be converted using the `gntpc -f rootracker` tool. This generator can read in primary vertex information from any ROOT tree conforming to the GENIE RooTracker format, and use this to populate the MC primary and MC parent information. The functionality (and code) is similar to the VertexFile generator, which reads the same information from RAT ROOT trees.

The syntax is

```
/generator/add rootracker FILE_PATH[:POSITION][:TIME][:NEVENTS][:NOFFSET]
```

where `FILE_PATH` is the path to the RooTracker ROOT file. The `POSITION` and `TIME` arguments optionally specify a position and time generator, which should be configured after this top-level generator. If these generators are not specified, the position and time information from the input file is used. The generator overrides are available to, e.g., simulate GENIE interactions without a geometry (which will all be located at the origin) and distribute them throughout

a specified volume in the RAT geometry. If NEVENTS or NOFFSET are specified, this sets the maximum event index and starting point in file. Use /run/beamOn to set the number of events to process; if NEVENTS is less, the event loop will terminate when this limit is reached.

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 multiplicity]
```

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).

The optional multiplicity is the number of primaries shot at once.

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. If the multiplicity is not specified, one primary will be shot.

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] by default, however several solar and stopped-pion spectra can be used instead. 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 by default; that is, it's the same neutrinos that cause both types of events.

To take the flux for elastic scattering from one of the solar spectra in *ratdb/SOLAR.db*, specify the process and neutrino flavor:

```
/generator/vtx/set  dir_x dir_y dir_z SOLAR:process:nu_flavor
```

To take the flux for elastic scattering from one of the stopped-pion spectra in *ratdb/STPI.db*, specify the timing profile and neutrino flavor:

```
/generator/vtx/set  dir_x dir_y dir_z STPI:timing_profile:nu_flavor
```

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.

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.

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",
"run_range": [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
}
```

Example:

```
/generator/vtx/set e- MICHEL
```

Produces Michel electron events drawn from the spectrum stored in the SPECTRUM[MICHEL] table. The spectrum is derived by integrating over cos(theta) using equation 56.3 in https://pdg.lbl.gov/2020/reviews/rpp2020-rev-muon-decay-params.pdf to get 3x**2 - 2x**3 where x = E_e / E_max and E_max = 52.8, and then normalizing to 1:

```
{
name: "SPECTRUM",
"index": "MICHEL",
"run_range": [0, 0],

// (Note that the energy spectrum has a maximum of 52.8 MeV)
"spec_e":     [ 0.00, 0.1, ..., 52.7, 52.8],
// (Note that first point is minimum of spectrum, last is maximum)
"spec_mag":   [ 0.00000000,0.00000041, ..., 0.03780678, 0.03780718],
}
```

Example:

```
/generator/vtx/set e- BOUND_MICHEL
```

Energy spectrum for michel electron from decay of mu- bound to oxygen from At. Data Nucl. Data Tables 54, 165 (1993):

```
{
name: "SPECTRUM",
"index": "BOUND_MICHEL",
"run_range": [0, 0],

"spec_e":     [ 0.0, 1.0, ..., 58.0, 59.0],
"spec_mag":   [0.00e+00, 4.27e-05, ..., 1.02e-04, 4.76e-05],
}
```

hepevt

```
/generator/vtx/set file_name
```

This generator takes a HEPEvt file as an input. The generator will read the file and generate events based on the information in the file. HEPEvt is a legacy file format typically used with FORTRAN-based generators, but is also supported by some modern generators such as MARLEY. The file should contain the event information in a specific format, which includes particle information such as PDG codes, momenta, and vertex positions.

Note that since this is only a vertex generator, it does not utilize the position and timing information provided in the HEPEvt file. Combination with a position and timing generator is still required.

Detailed documentation of the format is documented in detail by MARLEY here.

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.

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.

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).

Time generators

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

poisson

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

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

uniform

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

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

1.1.8 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.

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.

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_de	string (optional)	Name of sensitive detector if this volume should register hits. Limited to ‘’/mydet/pmt/inner”, ‘’/mydet/fibers” 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
- nestedtubearray - Array of three nested tubes. Useful to simulate optical fibers
- 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_ax	string (optional)	Axis along which to replicate volume: x, y, z
replica_sp	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_mod	string	Serves as the index for PMT, PMTCHARGE, and PMTTRANSIT tables giving the geometry, charge response, and time response models.
pos_tab	string	Specifies the table containing position (and direction) arrays specifying how to place PMTs
start_i	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)
orienta	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_	float[3] (optional)	Point (mm) in mother volume to aim all PMTs toward.
rescale	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.

NestedTubeArray Fields:

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 "Validation"
/rat/db/set DETECTOR geo_file "Validation/Valid.geo"
```

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

1.1.9 Physics Processes

The standard RAT simulation includes many standard GEANT4 physics processes.

Physics Lists

Add more documentation for the Physics List!

Commands that related to the physics list are listed under `/rat/physics`. You can find out what they do in `src/cmd/src/PhysicsListMessenger.cc`. Descriptions to these commands will be added here soon.

Remove a process from the physics list

If you want to remove a process from the current physics list (e.g. if you would like to stop simulating muon decays), you can modify the physics list in your main macro file (e.g. run.mac) as follows:

> /rat/physics/removeProcess mu+ Decay /rat/physics/removeProcess mu- Decay

Where the first argument is the G4 name of the particle, and the second is the name of the process you would like to remove.

Cerenkov Radiation

The Cerenkov class implemented in ratpac-two simulations is based on the original GEANT4 11.1.2 class, with a few key updates and added features. The base source file can be found within src/physics/src/ as RAT::G4CerenkovProcess, and is implemented within ThinnableG4Cerenkov.cc, which allows an effective thinning factor to be applied to the number of produced photons. The PMT efficiency is inversely scaled to account for the thinning factor (as long as the thinning factor does not increase the QE above 1.0). RAT::G4CerenkovProcess allows for the correct simulation of materials with non-monotonic refractive indices, which has been observed in noble liquids. GEANT4 could not handle these materials previously, which resulted in the original class under-predicting the Cerenkov photon yield.

There are a few commands users can use to 'tune' their simulation to match physical expectations. These are described below.

Command | Function | Default Value |
:——————————————- | :———————————————————————————- | :— |
/rat/physics/enableCerenkov | Controls whether we simulate Cerenkov photons (bool | true / false). | true |

/rat/physics/setCerenkovMaxNumPhotonsPerStep | Controls the maximum number of photons produced within a step (integer). | 1 |

/rat/physics/setCerenkovMaxBetaChangePerStep | Controls the maximum change in the phase velocity within a step (percentage). | 10.0 |

Dicebox

Add documentation for dicebox.

1.1.10 Optical Photon Processes

Cherenkov

To generate Cherenkov light, ratpac-two uses the `Geant4` G4Cerenkov class, with some small changes. The class in ratpac-two is called `ThinnableG4Cerenkov` as it primarily add the ability to 'thin' the number of photons that get propagated in ratpac-two. This option is provided in order to increase the speed of the simulation, and is discussed in more detail in *Photon Thinning*.

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:

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

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

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

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.

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`.

Quenching Models

Photon Thinning

Describe photon thinning here.

1.1.11 Event Processors

Event processors, described in *Event Producers vs. Event Processors*, are part of the event loop. They do not create new events, but instead receive events one-by-one and may either change the event by adding to or altering its contents. As an example, the data acquisition (DAQ) processors will receive an event and then, based on information such as the number of PMTs that detected light, decide whether the event caused the detector to trigger. All ratpac-two processors inherit the methods from the processor class, which provides the structure for how all processors run. For more details about these methods, how to use them, and how to write new processors, find details in the Programmer's guide: *Creating a Processor*. Below we describe the existing processors in ratpac-two.

Using a Processor From the Macro

The ratpac-two processors run as a block in the macro, instantiated after the `/run/initialize` line and prior to the generators. Below is an example of where several processors are run in a macro. Here we specify the last processor in the chain using the `proclast` syntax.

```
# Setup database parameters

/run/initialize

# DAQ processor
/rat/proc splitevdaq

# Count processor
/rat/proc count
/rat/procset update 100

# Quad fitter reconstruction
/rat/proc quadfitter

# Choose output file type
/rat/proclast outntuple

/generator/add combo gun:point:poisson

# ...
# Generator continued
```

The parameters for a processor are often highly configurable. This can be achieved by loading these parameters from the database, ratdb, described more in *The RAT Database (RATDB)*. In general, if the processors parameter is loaded from ratdb, we can change it from the macro using (using the PMT noise processor as an example):

```
/rat/db/set NOISEPROC noise_flag 1
```

Additionally, for processors, there are methods provided that allow the user to directly change parameters using the `procset` syntax (as shown already in the above example for the count processor):

```
/rat/proc splitevdaq
/rat/procset trigger_threshold 4.0
```

Cases where a parameter is tunable using `procset` are documented specifically for each processor below. For more details the Programmer's guide: *Creating a Processor* shows how this is achieved in the code using the 'Set' methods provided by the Processor class.

Count Processor

The count processor (located in `src/core`) 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 (3 triggered events)
```

This can be useful for quickly and roughly understanding what fraction of total simulated events (5) are causing detector triggers (3). In certain cases where the simulated events can create multiple triggered events, we may observe that in the count processor:

```
CountProc: Event 5 (10 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).

Prune Processor

The Prune processor is a processor for removing unwanted parts of the data structure to save space. The prune processor may be useful to call before the 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. The currently allowed entries are:
  - mc

- 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).

Python Processor

The python processor is a relatively unsupported feature of `ratpac-two`, but works for simple implementations of processors. There are several examples of python processors implemented in `python/ratproc/` that can be used to help develop a new processor. The base classes for the processors are provided in `python/ratproc/base.py`, which can be overloaded in dedicated python processors. The easiest example to follow is the python count processor, which provides the same functionality as the C++ version discussed in *Count Processor*, and is located in `python/ratproc/count.py`. The python count processor (the class is named `Count`) can be run from the macro using:

```
/rat/proc python
/rat/procset class "ratproc.Count(interval=10)"
```

PMT Processors

The PMT processors are described in *PMTs*.

DAQ Processors

The DAQ processors run the data aquisition model and are described in *DAQ Models*.

Reconstruction Processors

The reconstruction processors are described in *Reconstruction*.

Output Processors

The output processors are described in *Output Processors*.

1.1.12 Input Producers

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.

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

1.1.13 Photodetector Simulation

Several different photodetectors are available in *ratpac-two*.

PMTs

ratpac-two uses a custom PMT simulation.

For each photon that enters a PMT volume, there is a probability of creating a photoelectron (PE), which is equal to the product of the quantum efficiency, an efficiency correction factor, and an optical correction factor that accounts for the polarization and incidence angle of the photon (relative to the photocathode). The bulk of the code that calculates the optical correction factor and applies the overall efficiency can be found in `src/physics/src/GLG4PMTOpticalModel.cc`.

The quantum efficiency for each PMT is defined in `OPTICS_Photocathode.ratdb` and is specified for each PMT in `PMT.ratdb` using the index `photocathode_surface`. The efficiency correction factor can be specified in two different ways. First, a field in `PMT.ratdb` called `efficiency_correction` can be set to scale the efficiency of all PMTs of the specified type. The value of this scaling defaults to one. Second, an array of correction factors, called `efficiency_corr`, can be set in the `PMTINFO.ratdb` file for every individual PMT. This will scale the efficiency for each PMT separately.

Once it has been determined that a photoelectron should be created for a PMT, it is added to the `HitPMTCollection`. For each hit PMT, a `DS::MCPMT` object is created. The PMT may have detected more than one PE, in which case the `DS::MCPhoton` class (which can be accessed by the `GetMCPhoton()` method in the `MCPMT` class) keeps track of information for each `MCPE`.

The `DS::PMT` objects are created only after a trigger event has been issued (see DAQ documentation *DAQ Models*) and can include effects from the `DAQ` and `trigger`. Similarly, the `DS::DigitPMT` objects are created by the waveform analysis processors, which run over digitized waveforms, as described in *Waveform analysis*.

Geometries

Describe the different PMT geometries.

Toroidal

Describe the toroidal PMTs.

Revolution

Describe the revolution PMTs.

Cubic

Describe the cubic PMTs.

Cylindrical

Describe the cylindrical PMTs.

Charge and Time Response

The PMT charge and time response is set in `Gsim`, which 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 realistic charge and time 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 the model for the large-area Hamamatsu r11780 12-inch PMT (arbitrary choice).

`PMTCHARGE` fields:

- `charge` - "x" values of the charge PDF (arbitrary units)
- `charge_prob` - "y" values of the charge PDF (will be normalized)

`PMTTRANSIT` fields:

- `cable_delay` - constant offset applied to all PMTs of this model (nanoseconds)
- `time` - "x" values of the time PDF (nanoseconds)
- `time_prob` - "y" values of the time PDF (will be normalized)

Note that this `cable_delay` is applied to every single PE and can be used to shift the timing for all hits by a single value. This is separate from the per-PMT cable delays that can be applied, as described in *Channel Status*.

Dark Noise Processor

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. The dark-noise simulation is performed by the `NoiseProc` processor, with controllable parameters set in the `NoiseProc.ratdb` table.

The noise is included in the simulation after the physics event has been propagated (all particles followed to extinction, and PMT hits recorded). In order to include the PMT noise hits into the trigger simulation, the noise processor should be called prior to the DAQ processor. All noise hits are flagged and can be selected using the `isDarkHit` method in the `MCPhoton` class.

Control

The `noise_flag` in the `NoiseProc.ratdb` table sets the way in which the dark noise is simulated.

0: Global rate – all PMTs in the detector have the same dark-rate. The value of the global rate is set by the `default_noise_rate` parameter. The default behavior of the noise processor is to use this global setting with a default rate of 1 kHz.

1: Per PMT model rate – all PMTs of the same type have the same rate. This may be useful if the detector has different types of PMTs. In this case the noise rate is read from the `PMT.ratdb` table for the appropriate model.

2: Per PMT rate – every PMT in the detector have a different noise rate. This may be useful if the noise rates have been measured for every PMT and one wants to simulate these specific per-PMT rates. In this case the noise rates are read from the relevant `PMTINFO.ratdb` file, where an array called `noise_rate` can be defined. As an example, from the macro we can change the noise simulation to per PMT-model rates and change the rate for a specified model:

```
/rat/db/set NOISEPROC noise_flag 1
/rat/db/set PMT[r14688] noise_rate 5000.0
```

These parameters can also be set using `procset`. For example, to set the `default_noise_rate` we would do:

```
/rat/proc noise
/rat/procset rate 5000.0
```

Command:

```
/rat/proc noise
```

Parameters:

```
/rat/procset flag [value]
```

- [value] int - sets the value of the `noise_flag`.

```
/rat/procset rate [value]
```

- [value] double - sets the value of the `default_noise_rate`.

```
/rat/procset lookback [value]
/rat/procset lookforward [value]
/rat/procset maxtime [value]
```

- [values] doubles - sets the relevant parameters for the noise window, described further below.

Timing and charge distributions

Noise hits are generated uniformly in time, throughout a window defined by the `noise_lookback` and `noise_lookforward` parameters in the `NoiseProc.ratdb` table. The parameters are set by default to 1000 ns each, and are typically centered around the first true PMT hit-time in the event (in the case that there are no hits, the window is centered around zero). The value of `noise_maxtime` sets the timing cut-off for generating noise-hits in the case of long-lived particles in the MC.

The PMT charge distribution is sampled assuming the normal SPE charge distribution, as described in *Charge and Time Response*.

PMT Afterpulsing Processor

Details of PMT afterpulsing

PMT Pulse Generation

Details of the PMT pulse generation here.

PMT Encapsulation

PMT encapsulation is used for several reasons, such as to ensure compatibility with multiple detection media (e.g. air, water, doped water).

The encapsulation code was originally created for the BUTTON experiment, in which each of the 96 PMTs used are enclosed by two hemisphere domes that are sealed together by metal flanges and bolts.

The encapsulation code structure is based off the PMT construction structure, in which a instance is initialized depending on the construction type given.

When enabled, the encapsulation object is created first, followed the pmt object. The PMT is then placed inside the encapsulation before itself is placed in the mother volume given.

Enabling Encapsulation

Encapsulation by default is turned off. In a .geo file, it can be enabled by adding the following line inside the `inner_pmts` index entry:

```
encapsulation: 1,
```

With 0 being off. It can also be added in a macro with:

```
/rat/db/set GEO[inner_pmts] encapsulation 1
```

The other line that must be included inside the `inner_pmts` index entry is the model type:

```
encapsulation_model: "modelname",
```

Where "modelname" must match an index entry name in `ENCAPSULATION.ratdb`.

Encapsulation model information

Encapsulation models need to be added to `ENCAPSULATION.ratdb`, which is located in `ratpac/ratdb`. An entry can be called by using the `encapsulation_model:` command as mentioned above. Each entry provides all the important information that is needed to create the encapsulation objects:

- Construction type
- Enable and disable additional objects

- Object dimensions and materials
- Off-centre object placements

The construction type is needed to ensure the correct encapsulation construction is loaded. This represents the general shape of the encapsulation used. For any materials used, their properties should be defined in `MATERIALS.ratdb` and `OPTICS.ratdb`. Any values given such as dimensions and positions should be given in mm. Multiple entries can use the same construction type, but can vary on the objects and object properties used.

Adding a new Encapsulation construction

Initially, the only encapsulation construction is the "hemisphere" type, which encapsulates the PMT inside two hemispheres. An inner volume is then created in which the PMT can be placed.

When creating a new construction model (e.g. a box), the .cc file should contain three main functions:

- An initial function that is called to create an instance with the information from the given `ENCAPSULATION.ratdb` entry.
- A build function that creates and returns the encapsulation.
- A placement function.

A new encapsulation construction should make the build as customisable as possible. The important object information such as those stated above should be called from an `ENCAPSULATION.ratdb` entry.

To use a new construction type, the option must be added to `PMTEncapsulationConstruction.cc`. This file uses the construction type that is given in the called `ENCAPSULATION.ratdb` entry to initiate the associated encapsulation construction. For a working example please see `HemisphereEncapsulation.cc/hh` which uses the "hemisphere" construction type.

Placing PMT

If encapsulation is used, then is possible that the medium inside the encapsulation is different to the mother volume medium it would be placed in without encapsulation on. This can be changed in `PMTFactoryBase.cc` to ensure that the correct mother volume is used for the placement. If using the visualizer, the scene tree is useful to see if the PMT has been placed inside the correct volume.

PMT Offset

The encapsulation is placed using the PMT position(s) and direction(s) given, this means that by default the PMT is placed in the center of the encapsulation. An offset can be given in the `ENCAPSULATION.ratdb` entry so that the PMT is placed off-centre inside the encapsulation. This currently works for z-axis offsets (i.e move the PMT forwards/backwards).

PMT Concentrators

Document the PMT concentrators.

Magnetic Compensation

Technically there is code in `geo/src/pmt/PMTFactoryBase.cc` that can be enabled to attempt to change the PMT efficiency based on a specified external magnetic field; however, this is not supported code and is by default turned off.

LAPPDs

Describe LAPPDs here.

Optical Fibers

Describe Liquid-O style fiber simulations here.

Channel Status

Details of the channel status here.

1.1.14 DAQ Models

The DAQ processors are located in ratpac-two in the `src/daq/` directory. These processors are provided primarily as simple examples and helpful tools for producing triggered events, but will not accurately represent a realistic trigger system for a detector. In general, the DAQ processors can provide the below listed functionality (although the simple versions skip several of these steps):

1. Read information from the database, specified in `DAQ.ratdb`, for the DAQ settings. Some settings are also provided directly through `/rat/proc setting value`, as detailed individually for each processor below.
2. Get the MC information, primarily the true number of PMTs (`DS::MCPMT`), that detected light.
3. For that group of PMTs, build-up information about the event using the PMT hit-times and/or PMT charges. As an example, we may generate a hypothetical trigger signal pulse that we could then check against a threshold.
4. Issue a trigger decision about whether to create a triggered event. This is represented in ratpac-two as a `DS::EV` object.
5. Build up information about the event, such as the event count, the trigger time, etc. We also create new PMT objects (`DS::PMT`) that represent PMT hits within the triggered event. Several of the DAQ processors will loop through these PMTs to create information such as the total integrated charge for the event.
6. Based on whether it's enabled, we run the waveform digitization for the triggered event.

In principle, the DAQ code provided in ratpac-two is primarily for testing purposes and any experiment using ratpac-two would write their own custom DAQ code that could build from what is provided. If no DAQ is specified there will be no triggered events, and only the `DS::MC` related branches will be filled. In other words, all of the detector-related aspects of the data-structure (`DS::EV`, `DS::PMT`, etc.) are only filled if one of the DAQ processors is run.

Forced Trigger

The forced trigger processor is the simplest triggering scheme, which forces the detector to trigger for every MC event. A single EV is created for every MC event and the event structure is filled accordingly. There is no condition for issuing a trigger decision. This may be used for testing a random pulsed trigger, a beam trigger, or something similar.

Command:

```
/rat/proc forcedtrigger
```

Parameters: None

The digitization settings can be configured through the `DAQ.ratdb` table.

Simple DAQ

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 (i.e. the "frontEndTime" of the first photon), 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, such that the number of DAQ events will equal the number of MC events. This acts very similarly to the forced trigger processor, but will only fill the PMT branch if there is at least one hit.

Command:

```
/rat/proc simpledaq
```

Parameters: None

Split-EV DAQ

The SplitEVDaq processor achieves the most realistic of the data acquisition models by summing square trigger pulses together according to the hit-times of the PMTs. The trigger sum is compared against a configurable global trigger threshold, and events above threshold cause a detector trigger. SplitEVDaq also properly handles splitting events separated in time into separate triggered events, which is critical for simulating coincidence events such as IBDs. The parameters of the triggering are highly configurable and include the width of trigger pulses, the size of the trigger window, the size of the time-steps, etc.

Command:

```
/rat/proc splitevdaq
```

Parameters:

```
/rat/procset pulse_width "value"
/rat/procset trigger_window "value"
/rat/procset trigger_threshold "value"
/rat/procset trigger_lockout "value"
/rat/procset trigger_resolution "value"
/rat/procset pmt_lockout "value"
/rat/procset lookback "value"
/rat/procset max_hit_time "value"
/rat/procset max_hit_duration "value"
/rat/procset trigger_on_noise "0"|"1"
/rat/procset digitizer_name "digitizer"
/rat/procset digitize "true"|"false"
```

The digitization settings can also be configured through the `DAQ.ratdb` table.

1.1.15 Digitization

In ratpac-two, the user has the option to digitize waveforms for each photodetector channel according to a set of user defined settings. The settings are specified in `DIGITIZER.ratdb` and are separated for different digitizer types. Settings for two common CAEN digitizers, the V1730 and V1742, are provided as examples.

The digitization of PMT pulses is triggered by certain DAQ code once the specified conditions are met. As an example, the `forcedtrigger` processor, described in more detail in *DAQ Models*, will digitize hit channels for every MC event, given that the value in the appropriate `DAQ.ratdb` table is specified as: `digitize: True`. Also specified in the

DAQ.ratdb table is which type of digitizer to select (e.g., V1730 or V1742), with V1730 being the typical default model.

When the digitization is enabled, a PMT pulse will be generated for every mcPE, detailed in *PMT Pulse Generation*. A fixed window is selected and the PMT pulse, with Gaussian noise added, is sampled across that window. The voltage (or ADC counts) as a function of time (in steps according to the sampling rate) is used to create a digitized waveform for each PMT. This waveform is written to the DS::Digit object. The waveforms can also be added to the ntuple output by setting include_digitizerwaveforms: true in IO.ratdb.

These waveforms can be further processed in ratpac-two in order to produce additional information, such as the integrated charge or the time-over-threshold, by applying the waveform analysis. The waveform analysis will open the DS::Digit to extract the waveforms and will process those waveforms according to user-specified criteria. The output of the waveform analysis will fill the DS::DigitPMT object. More details on the waveform analysis are provided in *Waveform analysis*.

1.1.16 Waveform analysis

It is typical for modern detectors to digitize PMT pulses on waveform digitizers and readout entire waveforms for each PMT channel. These waveforms are then often processed offline to produce PMT hit-times and integrated charges, among other variables. ratpac-two both provides realistic simulation of the waveform digitizers, discussed in *Digitization*, but also a full chain of waveform analysis that can be used to extract information about the PMT pulses. As with other features, this existing waveform analysis code can easily be extended for an experiment's particular use-cases, and the provided waveform processors should be treated as helpful examples rather than finished products.

The waveform analysis processors are run over the digitized waveforms (the DS::Digit objects). These processors are only run if (a) a DAQ processor is run (b) digitization of the PMT waveforms is enabled (c) the waveform processing is enabled and (d) the waveform processor is run from the macro.

In order to enable the digitization of waveforms, discussed in *Digitization*, we adjust the DAQ.ratdb table and set digitize: True for the appropriate DAQ. As with all ratdb parameters, this can be achieved from the macro. This will cause waveforms for each MCPMT to be created, which can be further processed using the waveform analysis tools.

The waveform analysis is enabled by running the waveform analysis processor (e.g., /rat/proc WaveformPrep). The analysis processors reads from a ratdb table called DIGITIZER_ANALYSIS. In order to change the index of the table that is loaded, the user can select a new index using /rat/procset index_name. More details are provided below.

Base analysis

The primary waveform analysis processor is run from WaveformPrep, which is critical to run prior to any other waveform processing. By default, WaveformPrep loads the DIGITIZER_ANALYSIS table with no index, which contains settings related to the pedestal window, the integration window, the voltage threshold to consider a PMT pulse, etc. WaveformPrep calculates (among other things) the integrated charge around the PMT pulse, the time-over-threshold, the voltage-over-threshold, the pedestal in a prompt-window, the peak voltage, the total charge across the full window, and the constant-fraction discriminator timing. These variables are written to the DigitPMT.

Parameters in ratdb for Waveform Prep are defined below. These parameters can all be set using the standard /rat/procset variable_name value syntax.:

```
voltage_threshold
```

- the threshold over which to count the hit as a PMT pulse. Waveforms without samples that cross this threshold can be optionally suppressed.

```
zero_suppress
```

- optionally specify whether to keep the DigitPMT if the waveform never crosses threshold. By default set to 1, meaning that these below-threshold waveforms will be removed and not analyzed.

```
pedestal_window_low
pedestal_window_high
```

- the lower and upper edge of the window to calculate the baseline, in samples

```
constant_fraction
lookback
```

- parameters to calculate the timing using a constant fraction discriminator.

```
integration_window_low
integration_window_high
```

- the lower and upper edge of the window, centered around the peak of the PMT pulse, to integrate the charge, in samples.

```
sliding_window_width
sliding_window_thresh
```

- parameters to calculate a total charge by sliding an integration window across the waveform.

```
apply_cable_offset
```

- apply the cable offset from the PMT channel status. More details are provided in *Channel Status*.

For waveforms that cross the specified threshold, a new `DS` object called the `DigitPMT` is created. These objects represent the PMT properties as measured by the waveform analysis tools. For example, the `GetDigitizedTime` method returns the digitized time as measured by `WaveformPrep`, which applies a constant-fraction-discriminator to extract a single hit-time for each waveform. Because there are several different analysis methods that might calculate a PMT hit-time, the results are separated using the `WaveformAnalysisResult` tool that is further described below.

There are several additional waveform analysis proccesors described below, each of which is attempting to provide a precise measurement of the arrival time for single photoelectron hits. To run the full chain of waveform analysis from the macro:

```
# This is set true by default, so not
# strictly necessary
/rat/db/set DAQ[SplitEVDAQ] digitize true

/run/initialize

# The DAQ will automatically digitize hit
# PMTs for the triggered events
/rat/proc splitevdaq
/rat/procset trigger_threshold 5.0

# Always start with the waveform prep!
/rat/proc WaveformPrep
# If we have custom settings, we could
# create a new DIGITIZER_ANALYSIS table
# and load the setting here like:
# /rat/procset analyzer_name "custom_settings"
# Which can be done similarly for the
# below processors as well.
```

(continues on next page)

(continued from previous page)

```
# Then run the other waveform analysis
# processors
/rat/proc WaveformAnalysisLognormal
# This automatically loads the
# DIGITIZER_ANALYSIS table with an
# index of 'LognormalFit' unless we
# select something else using
# /rat/procset analyzer_name "custom_settings"

/rat/proc WaveformAnalysisGaussian

/rat/proc WaveformAnalysisSinc

/rat/proc WaveformAnalysisRAVEN
```

For all of these processors, there is a utility located in `util/src/` called `WaveformUtil.cc` that provides useful analysis tools. For example, there are public methods to convert ADC counts to voltage, identify the peak of the waveform and the corresponding sample, get the total number of threshold crossings, etc.

Common parameters

All waveform analysis processors inherit from `WaveformAnalyzerBase` and share the following parameters, which can be set using `/rat/procset`:

```
min_total_charge
max_total_charge
```

- Lower and upper bounds (in pC) on the digitized total charge of the waveform (`DigitPMT::GetDigitizedTotalCharge()`). If the total charge falls outside `[min_total_charge, max_total_charge]`, the analysis for that channel is skipped entirely. By default these are set to the most negative and most positive finite `double` values respectively, so all waveforms are analyzed unless a cut is explicitly specified.

For example, to restrict analysis to waveforms with total charge between -5 pC and 50 pC:

```
/rat/proc WaveformAnalysisLognormal
/rat/procset min_total_charge -5.0
/rat/procset max_total_charge 50.0
```

These cuts are applied per-channel, per-event, before `DoAnalysis()` is called, so they are a lightweight way to skip waveforms that are clearly noise or saturated without running the full analysis.

Lognormal fitting

Performs PMT waveform analysis using a lognormal distribution fit to extract timing and charge information from PMT pulses. This method fits a single lognormal function to the entire waveform around the peak region, seeded by digitTime.

The lognormal function used has the following form:

$$f(t) = \frac{Ae^{-\frac{(\ln((t-\theta)/m))^2}{2\sigma^2}}}{(t-\theta)\sigma\sqrt{2\pi}} \qquad x > \theta; m, \sigma > 0$$

where θ is the time offset parameter, m is the scale parameter, σ is the shape parameter (fixed during fitting), and the magnitude parameter a determines the pulse amplitude. The fitted time is calculated as $\theta + m$, and the charge is derived from the magnitude parameter.

The method can be configured using the following ratdb parameters:

Name	Description
`fit_window_low`	Time window before the digitized peak time to include in the fit, in ns.
`fit_window_high`	Time window after the digitized peak time to include in the fit, in ns.
`lognormal_shape`	The "sigma" parameter in the lognormal function controlling the pulse width.
`lognormal_scale`	The "m" parameter in the lognormal function controlling the pulse timing characteristics.

Gaussian fitting

Performs PMT waveform analysis using a Gaussian distribution fit to extract timing and charge information from PMT pulses. This method fits a single Gaussian function to the waveform around the peak region, seeded by digitTime.

The Gaussian function used has the following form:

$$g(t) = \frac{Ae^{-\frac{(t-\mu)^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}}$$

where μ is the mean (fitted time), σ is the standard deviation (fitted within specified bounds), and the magnitude parameter A determines the pulse amplitude. The fitted time is directly μ, and the charge is derived from the magnitude parameter.

The method can be configured using the following ratdb parameters:

Name	Description
`fit_window_low`	Time window before the digitized peak time to include in the fit, in ns.
`fit_window_high`	Time window after the digitized peak time to include in the fit, in ns.
`gaussian_width`	Initial value for the Gaussian width (sigma) parameter, in ns.
`gaussian_width_low`	Lower bound for the fitted Gaussian width parameter, in ns.
`gaussian_width_high`	Upper bound for the fitted Gaussian width parameter, in ns.

Sinc interpolation

Describe sinc interpolation.

Richardson-Lucy Direct De-modulation (LucyDDM)

Perform PMT waveform analysis using LucyDDM, a time-domain deconvolution algorithm that enhanced resolution compared to FFT-based convolution methods. This method is able to recosntruction multiple photoelectrons in a single PMT. The primary procedure is described in Sec. 3.3.2 of https://arxiv.org/abs/2112.06913. After deconvolution, peaks in the deconvolved waveform are identified, and a Gaussian fit is performed on each peak to extract time and charge information. Optionally, a likelihood-based NPE estimation can be performed on each resolved wave packet to further improve the charge and time resolution.

The method can be configured using the following ratdb parameters.

Name	Description
vpe_scale	The "m" parameter in the lognormal function that generates single PE waveform template.
vpe_shape	The "sigma" parameter in the lognormal function.
vpe_charge	The nominal charge of a single photoelectron in pC. A resolved wave packet with integral of 1 will have be assigned this amount of charge.
roi_threshold	The threshold (in mV) above which the deconvolution will be performed on. All samples below threshold will be set to effectively zero (small positive value for numerical stability).
max_iterations	Maximum number of LucyDDM iterations to run.
stopping_nll_d	Stop LucyDDM iterations if the negative log likelihood improvement is less than this value.
peak_height_th	All peaks below this peak height in the deconvolved waveform will not be considered.
charge_thresho	All wave packets with integrated charge below this threshold will not be considered.
min_peak_dista	If two resolved wave packets are closer than this distance (in ns), they will be merged and considered as one wave packet.
npe_estimate	If true, perform a NPE estimation on all resolved waveform packets using a likelihood on the integral of the packets.
npe_estimate_cl	Width of the single PE charge distribution (in pC) used in the NPE estimation likelihood.
npe_estimate_m	Maximum number of PEs to consider in the NPE estimation likelihood.

RAVEN

Performs PMT waveform analysis using RAVEN (Reverse Analysis of Voltage Events with Nonnegativity), a sparse non-negative least squares fitting algorithm that reconstructs multiple photoelectron times and charges from digitized waveforms. The underlying algorithm (rsNNLS) is described in Sec. 3.1.1 of https://www.sciencedirect.com/science/article/pii/S0925231211006370. The method builds a dictionary matrix of time-shifted single PE templates at sub-sample resolution, applies non-negative least squares to find optimal template weights, and iteratively removes low-significance components to improve sparsity. After weight merging, NPE estimation can be performed on resolved peaks. Two template types are supported: lognormal (asymmetric) and Gaussian (symmetric).

The method can be configured using the following ratdb parameters:

Name	Description
process_threshold_crossing	Enable region-based processing (0=no, 1=yes).
voltage_threshold	Voltage threshold for region detection, in mV.
region_padding	Number of samples to pad around threshold crossing regions.
raven_template_type	Template type: 0=lognormal, 1=gaussian.
lognormal_scale	Lognormal "m" parameter (used when raven_template_type=0).
lognormal_shape	Lognormal "sigma" parameter (used when raven_template_type=0).
gaussian_width	Gaussian sigma parameter (used when raven_template_type=1).
vpe_charge	Nominal charge of a single PE in pC.
upsampling_factor	Dictionary upsampling factor for sub-sample timing resolution.
max_iterations	Maximum iterative thresholding iterations.
nnls_tolerance	NNLS convergence tolerance.
weight_threshold	Minimum weight for component significance.
weight_merge_window	Time window (ns) for merging nearby weights. Set to 0 to disable.
npe_estimate	If true, perform NPE estimation on resolved wave packets.
npe_estimate_charge_width	Width of single PE charge distribution (in pC) for NPE estimation.
npe_estimate_max_pes	Maximum number of PEs to consider in NPE estimation.

WaveformAnalysisResult

The `WaveformAnalysisResult` class is a data structure that stores the output from waveform analysis processors. Each waveform analysis method creates a `WaveformAnalysisResult` object that is attached to the `DigitPMT` to store its specific analysis results. This design allows multiple analysis methods to be run on the same waveform, with each method's results stored separately and accessible independently.

The `WaveformAnalysisResult` object maintains three parallel arrays that are automatically sorted by time:

- **Times**: The reconstructed pulse times within the digitization window (no cable delay or trigger offset applied)
- **Charges**: The corresponding pulse charges, nominally in units of pC
- **Figures of Merit**: Additional analysis-specific metrics stored in a map structure

Key features of the `WaveformAnalysisResult` include:

Multiple PE Support: Unlike the basic `DigitPMT` analysis which typically identifies a single PE per waveform, `WaveformAnalysisResult` can store multiple reconstructed photoelectrons.

Automatic Time Ordering: When pulses are added using `AddPE()`, they are automatically inserted in the correct time-ordered position, ensuring that all arrays remain synchronized and sorted.

Flexible Figures of Merit: Each analysis method can store method-specific quality metrics (e.g., chi-squared, fit width, baseline) that are automatically synchronized with the time and charge arrays.

Time Offset Handling: The class supports time offset corrections to account for cable delays or trigger timing, which can be applied when retrieving results without affecting the stored raw timing.

The `WaveformAnalysisResult` objects are accessed from the `DigitPMT` using the method name as a key:

```
DS::WaveformAnalysisResult* lognormal_result = digitpmt->
↪GetOrCreateWaveformAnalysisResult("Lognormal");
DS::WaveformAnalysisResult* gaussian_result = digitpmt->
↪GetOrCreateWaveformAnalysisResult("Gaussian");

int n_pes = lognormal_result->getNPEs();

for (int i = 0; i < n_pes; i++) {
    double time = lognormal_result->getTime(i);
    double charge = lognormal_result->getCharge(i);
    double chi2 = lognormal_result->getFOM("chi2ndf", i);  // Method-specific FOM
}
```

This design enables comprehensive comparison between different waveform analysis methods and supports multi-PE reconstruction techniques.

1.1.17 Reconstruction

Here we describe the reconstruction processors.

Fitter Input Handler

Document the fitter handler.

Centroid Fitter

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

Quad Fitter

Quad fitter details.

Direction Center Fitter

The `fitdirectioncenter` processor reconstructs the direction of events as the average of the vectors from the event position to the selected PMT positions.

Command:

```
/rat/proc fitdirectioncenter
```

Parameters

No parameters are required aside from an event position. Several useful parameters can be set in macro, which allows the processor to be run multiple times with different settings in a single macro. Several figures of merit are available.

Detailed implementations are illustrated in macros/examples/fitdirectioncenter.mac In particular, there is an example to correct for the drive effect in reconstructed position. First, a position reconstruction is run, then a direction reconstruction, as usual. However, a second direction reconstruction is run and takes both the reconstructed position and direction as input to correct for the drive. The resulting position is then saved in the fitdirectioncenter FitResult.

Field	Type	Description
label	string	Additional string appended to "fitdirectioncenter"
position_fitter	string	Name of fitter providing position input
direction_fitter	string	Name of fitter providing direction for drive correction
pmt_type	int	PMT "type" to use. Multiple types can be used. Defaults to all types.
time_resid_low	double	Lower cut on time residuals in ns
time_resid_up	double	Upper cut on time residuals in ns
time_resid_frac_low	double	Lower cut on time residuals as a fraction in [0.0, 1.0)
time_resid_frac_up	double	Upper cut on time residuals as a fraction in (0.0, 1.0]
light_speed	double	Speed of light in material in mm/ns. Defaults to water.
event_position_x	double	Fixed position of event in mm
event_position_y	double	Fixed position of event in mm
event_position_z	double	Fixed position of event in mm
event_time	double	Fixed offset of time residuals in ns
event_drive	double	Fixed offset of position input in mm

Direction fit information in data structure

- figure of merit - num_PMT is the number of PMTs used in the reconstruction
- figure of merit - time_resid_low is the earliest time residual that passes the lower time residual cut
- figure of merit - time_resid_up is the latest time residual that passes the upper time residual cut
- figure of merit - angle_mean is the mean angle between reconstructed direction and photon path directions
- figure of merit - angle_stddev is the standard deviation of directions about the mean angle
- figure of merit - angle_octile7 is the seventh octile (87.5%) of angles between reconstructed direction and photon path directions
- figure of merit - angle_quartile3 is the third quartile (75%) of angles between reconstructed direction and photon path directions
- figure of merit - angle_median is the median (50%) of angles between reconstructed direction and photon path directions
- figure of merit - angle_quartile1 is the first quartile (25%) of angles between reconstructed direction and photon path directions
- figure of merit - angle_octile1 is the first octile (12.5%) of angles between reconstructed direction and photon path directions

---

Path Fitter

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: 1. Hit time residuals are minimized directly using *simulated-annealing* from a static seed. 2. PathFitter likelihood is minimized with *simulated-annealing* from stage 1's result. 3. 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_multi	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_ar	double	Area of photocathode mm^2
direct_time_fir	double	Time (ns) of first entry in `direct_time_prob`
direct_time_ste	double	Time step (ns) between entries in `direct_time_prob`
direct_time_pro	double[]	Probability (need not be normalized) of being “direct” light with a certain time residual
other_time_firs	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.

MiniSim

What does this do? Do we need this in RAT?

ClassifyChargeBalance

The classifychargebalance processor calculates the standard deviation divided by the mean of the charges on hit PMT channels.

Command:

```
/rat/proc classifychargebalance
```

Parameters: None

Classifier information in data structure

- name - chargebalance
- figures of merit - None

ClassifyTimes

The classifytimes processor calculates characteristic parameters of a hit time residual distribution. One parameter is the ratio of hits in a time window (typically narrow around prompt times) divided by the hits in another time window (often the full time window; i.e., all hits). Within another specified time window, the four central moments are calculated: mean, unbiased standard deviation, standardized unbiased skewness, and standardized unbiased excess kurtosis.

Command:

```
/rat/proc classifytimes
```

Parameters

No parameters are required though a position reconstruction should be run before (or a fixed position specified) and a light speed is needed to calculate time residuals. The light speed and a time window for the ratio are specified in CLASSIFIER.ratdb. Several useful parameters can be set in macro, which allows the processor to be run multiple times with different settings in a single macro. Detailed implementations are illustrated in macros/examples/classifytimes.mac.

Field	Type	Description
classifier_name	string	Defaults to "classifytimes".
position_fitter	string	Name of fitter providing position input.
pmt_type	int	PMT "type" to use. Multiple types can be used. Defaults to all types.
verbose	int	FOM save option. 1 saves num_PMT's. 2 also saves time_resid_low and time_resid_up
numer_time_resid_l	double	Lower cut on time residuals in ns. Used for ratio. Defaults to value in CLASSIFIER.ratdb.
numer_time_resid_u	double	Upper cut on time residuals in ns. Used for ratio. Defaults to value in CLASSIFIER.ratdb.
denom_time_resid_l	double	Lower cut on time residuals in ns. Used for ratio. Defaults to full range.
denom_time_resid_u	double	Upper cut on time residuals in ns. Used for ratio. Defaults to full range.
time_resid_low	double	Lower cut on time residuals in ns. Option for central moments.
time_resid_up	double	Upper cut on time residuals in ns. Option for central moments.
time_resid_frac_lo	double	Lower cut on time residuals as a fraction in [0.0, 1.0). Option for central moments.
time_resid_frac_up	double	Upper cut on time residuals as a fraction in (0.0, 1.0]. Option for central moments.
light_speed	double	Speed of light in material in mm/ns. Defaults to water value in CLASSIFIER.ratdb.
event_position_x	double	Fixed position of event in mm.
event_position_y	double	Fixed position of event in mm.
event_position_z	double	Fixed position of event in mm.
event_time	double	Fixed offset of time residuals in ns.

Classifier information in data structure

- name - classifytimes
- classifier result - ratio is the ratio of the numbers of PMTs selected by specified time windows
- classifier result - mean is the mean time within the time window for central moments
- classifier result - stddev is the unbiased standard deviation of times within the time window for central moments
- classifier result - skewness is the standardized unbiased skewness of times within the time window for central moments
- classifier result - kurtosis is the standardized unbiased excess kurtosis of times within the time window for central moments
- classifier result - num_PMT is the number of PMTs used in the central moment calculations
- classifier result - num_PMT_numer is the number of PMTs used in the numerator of the ratio
- classifier result - num_PMT_denom is the number of PMTs used in the denominator of the ratio
- classifier result - time_resid_low is the earliest time residual in the time window for central moments
- classifier result - time_resid_up is the latest time residual in the time window for central moments

FitTensor

Document this!

FitMimir

MIMIR is a general purpose event reconstruction framework that is meant to encapsulate a large number of minimization-based reconstruction strategies using a modular appraoch. It is designed to be flexible and extensible, allowing a fit to be described in components taht can be added both in RATPAC2 itself and in a downstram prirvate experiment.

Command

```
/rat/proc mimir
```

Concepts

Comoponents

The MIMIR framework consists of a number of **components** that can be put together into a full reconstruction recipe. There are three types of components:

- **CostFunction**: A function that the fit aims to minimize. This is typically a likelihood function or a similar function that produces a numerical value that represents the goodness of the current fit.
- **Optimizer**: An engine that can minimize a given cost function (e.g. minuit, minuit2, NLOPT).
- **FitStrategy**: A recipe that controls what is done in a fit. A fit strategy typically instantiates and utilizes the above components during a fit. It is worthy of note that FitStrategies can also consists of other FitStrategies, making them flexible and extensible.

ParamSet

All MIMIR components manipulates a parameter set (`ParamSet`). This is a structure that consists of the 7 possible paramers that could be fitted for a event: The position of the event (xyz), the time of the event (t), the direction of the event (theta, phi), and the energy of the event (E). For each of these fields, `ParamSet` keeps track of the current value (either for seeding or in the middle of a fit) of the field, the left and right bounds of the value, as well as the status of the field. By default, the fields are set to have bounds that are effectively infinite: the positional and time coordinates have bounds at `std::numerical_limits`, the directional componets have bounds at `[0, pi]` and `[-pi, pi]`, and energy has bounds at `[0, std::numerical_limits]`.

The fields can take on the following statuses:

- **INACTIVE**: The field does not participate in the fit nor the cost function. It will be ignored completely.
- **FIXED**: The field does not participate in the fit, but is relevant for evaluating the cost function. It will be passed to the cost function but its value will not be modified by the optimizer.
- **ACTIVE**: The field is currently being fitted, it will be passed to the cost function and its value will be modified by the optimizer.

Top-level Configuration

All components of MIMIR are cinfigured via entries to the RATDB. All MIMIR-related configuration blocks have the tablename prefix of `MIMIR_`. At the very top level, the processor instance needs to be pointed to a configuration for a `FitStrategy` configuration block. This can be done in the macro via `procset` comamands. For example, to instruct the fitter to use the strategy `FitStep` with the configuration type `PositionTime_PMTTypeTimeResidual`, one would use:

```
/rat/procset strategy "FitStep[PositionTime_PMTTypeTimeResidual]"
```

If no such `procset` command is given, MIMIR will fall back to the strategy specified in the RATDB. The following RATDB block does the same thing as above:

```
{
  "name": "FIT_MIMIR",
  "index": "",
  "strategy": "FitStep",
  "strategy_config": "PositionTime_PMTTypeTimeResidual",
}
```

Writing a MIMIR component

MIMIR components are all templated classes that requires the override of several functions. The components can be added in either RATPAC2 itsslef or a downstream experiment.

All components require the following to be done:

- override `bool configure(ratdblinkptr db_link)`: this function receives a ratdb configuration block and correctly instatiates the component.
- register the component with the mimir framework by calling the following preprocessor macro: `mimir_register_type(componettype, classname, "humanreadableclassname")`, where `componenttype` is either `fitstrategy`, `cost`, or `optimizer` in the `rat::mimir` namespace.

Each type of component has the following additional requirements:

- **Cost**: Override the following functions:
  - `double operator()(const ParamSet& params)`: takes in a ParamSet and returns the value of the cost function for that set of parameters.
- **Optimizer**: Override the following function:
  - `void minimizeimpl(std::function<double(const paramset&)> cost, paramset&params)`: given a callable function *cost*, modifies *params* such that *cost* is minimized. note that the internal templating of *optimizer* allows this minimization routine to be used for both minimization and maximization via `optimizer::minimize` and `optimizer::maximize`.
- **FitStrategy**:
  - Override the following functions:
    - `void Execute(ParamSet &params)`: takes in the `params` as a set of seeds and bounds, perform the fit, and writes ther eseult back to `params`.
  - During initialization, the FitStrategy also should: - correctly identifies the optimizers and costs associates with the strategy, look up the correct configuration bloccks from RATDB, and instantiate the components.

The Component Factory

To instantiate a MIMIR component, one should utilize the MIMIR component factory, which provides many convenience functions for creating components based on their type and configuration. The factory can be used as follows:

```
Factory<Cost>::GetInstance().make_and_configure(name, index);
```

Where the complating typename `Cost` can be replaced with `Optimizer` or `FitStrategy` to create the corresponding component. The `name` and `index` are the type name and configuration index for the component, respectively. The

factory will create a component with typename `name` and use the RATDB entry of `MIMIR_name[index]` for configuration.

Available Strategies

FitStep

`FitStep` is the simplest fit strategies that can be used with MIMIR. It takes in an optimizer and a cost function, and runs the optimizer to minimize the cost function.

Field	Type	Description
optimizer_name	string	Name of the optimizer type to use.
optimizer_config	string	Configuration to use for the optimizer.
cost_name	string	Name of the cost type to use.
cost_config	string	Configuration to use for the cost function.
position_time_statu	int or int[4]	Status codes for `[x, y, z, t]`, `0 = INIACTIVE, 1 = ACTIVE, 2 = FIXED`.
direction_status	int or int[2]	Status codes for `[theta, phi]`
energy_status	int or int[1]	Status codes for `E`
x_bound	double[2]	Bounds for the field `x`. Also works for all other fields: y, z, t, theta, phi, energy.

FitSteps

Field	Type	Description
steps	string[]	A list of `FitStep` configurations to run in order.

Available Optimizers

RootOptimizer

`RootOptimizer` is a wrapper around the `ROOT::Math::Minimizer` class. See the official documentation for details.

Field	Type	Description
minimizer_type	string	Passed to `ROOT::Math::Factory::CreateMinimizer` as `minimizerType`.
algo_type	string	Passed to `Root::Math::Factory::CreateMinimizer` as `algoType`.
max_function_ca	int	Specifies the max number of calls to the cost function.
max_iterations	int	Specifies the maximum number of iterations to run the minimizer.
tolerance	double	Absolute tolerance for the minimizer. Detailed use may vary depending on the minimzier and algorithm used.
print_level	int	Verbosity level for the minimizer. 0 is silent, 1 is normal, 2 is verbose.

NLOPTOptimizer

`NLOPTOptimizer` is a wrapper around the `nlopt::opt` class. See the official documentation for details.

This optimizer only supports **gradient-free** (derivative-free) algorithms. Gradient-based algorithms (those starting with `LD_` or `GD_`) are not supported and will cause the configuration to fail with a clear error message.

Field	Type	Description
algo_type	strir	NLopt algorithm name (e.g., LN_COBYLA, LN_NELDERMEAD, LN_SBPLX). Must be a gradient-free algorithm (LN_* or GN_*). See NLopt algorithms for a full list.
max_functio	int	Maximum number of objective function evaluations allowed.
tolerance	doubl	Relative tolerance on the optimization parameters (xtol_rel). The optimizer stops when the change in all parameters is less than this tolerance.

Available Costs

PMTTypeTimeResidualPDF

Evaluates a 1D time residual PDF as a negative log likelihood.

Note that the received histograms will be noramlized such that the integral in the range specified by binning is 1.0. The negative natural logirithms of the bin heights are then calculated and evaluated via a cubic spline. When the computed time reisudal is out of range for the current hypothesis, the PDF will evaluate to the left or right edge of the binning.

PMTTypeCosAlphaPDF

Evaluates a 1D CosAlpha PDF as a negative log likelihood.

1.1.18 Output Processors

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 is called "ds". The full RAT data structure is described in *The ratpac-two Data Structure*.

Command:

```
/rat/proc outroot
```

Parameters:

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

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

outntuple

The OutNtuple processor writes events to disk as a flat ROOT Tree, and are thus much smaller and easier to work with than the files output by the outroot processor. There is also significant flexibility in what information is stored to the file, as detailed below. To run this processor and write the ntuple files out:

Command:

```
/rat/proc outntuple
```

Parameters:

```
/rat/procset file "filename"

/rat/procset include_mcparticles 1
```

(continues on next page)

(continued from previous page)

```
/rat/procset include_tracking 1
/rat/procset include_pmthits 1
/rat/procset include_nestedtubehits 1
/rat/procset include_untriggered_events 1
/rat/procset include_mchits 1
/rat/procset include_digitizerwaveforms 1
/rat/procset include_digitizerhits 1
/rat/procset include_digitizerfits 1
/rat/procset waveform_fitters ["Lognormal","Gaussian","Sinc","LucyDDM","RAVEN"]
/rat/procset waveform_fitter_FOM_FITTERNAME ["FOM1","FOM2"]
/rat/procset event_fitters ["quadfitter","fitcentroid","fitdirectioncenter","mimir"]
/rat/procset event_fitter_FOM_FITTERNAME ["FOM1","FOM2"]
```

- `filename` (required, string) Sets output filename. File will be deleted if it already exists.
- `include_*` (optional, int) Sets whether the ntuple structure will be extended to include more variables, as detailed below. By default the following, based on the entries in IO.ratdb, the following are set to 0 by default: `include_tracking`, `include_mcparticles`, and `include_digitizerwaveforms` and the rest are set to 1 by default (i.e., the associated variables aare included in the ntuple file, as detailed below).
- `waveform_fitters` (optional, vector<string>) Waveform analysis algorithm results to include in the ntuple. See below for naming of the specific variables.
- `waveform_fitter_FOM_FITTERNAME` (optional, vector<string>) The figure of merit to include for each waveform fitter. See below for naming of the specific variables.
- `event_fitters` (optional, vector<string>) Event reconstruction algorithm results to include in the ntuple. See below for naming of the specific variables.
- `event_fitter_FOM_FITTERNAME` (optional, vector<string>) The figure of merit to include for each event fitter. See below for naming of the specific variables. FOM can be specified by the base name of the reconstruction algorithm (e.g., `event_fitter_FOM_quadfitter`) or by the specific instances of each algorithm (e.g. `event_fitter_FOM_fitdirectioncenter__0_quad`).

Similarly to the outroot file, one can pass the filename using the "-o" flag by running the macro as:

```
rat example_macro.mac -o output.root
```

The ntuple contains flat `TTrees` called `meta` and `output`, and optionally a tree called `waveform`. The `output` branch contains data for each event, and is structured to hold both the "true" simulated quantities as well as the "detector" quantities, that are filled after a data acquisition processor are run. As an example, we see in the below table that the `output` branch contains entries for `mcx`, the true position of the simulated event, as well as `triggerTime`, the time of the event trigger. This can lead to confusion because not all simulated events will trigger the detector. Importantly, if we do not run any DAQ simulation, none of the events will be saved unless `include_untriggered_events` is set to true. If you're entirely interested in studying variables generated prior to the data aquisition (e.g., particle position, number of Cherenkov photons produced, the true number of detected photoelectrons, etc.), be certain to set `include_untriggered_events` to true in the macro.

If we do run the DAQ simulation, only information for the triggered events will be saved to the `output` branch, unless `include_untriggered_events` is set to true, in which case all events will be saved to the `output` branch. Furthermore, for triggered event we should expect to have differences between the simulated and detector quantities. For example, the `mcPMTID` (only filled when `mchits` is set true), the true set of PMTs that detected light, the `hitPMTID` (only filled when `pmthits` is set true), the set of PMTs that detected light after applying the data aquisition simulation, and the `digitPMTID`, the set of PMTs that detected light after applying the data aquisition and waveform analysis simulations, can all be different length vectors (the `hitPMTID` and `digitPMTID` will always be a subset of `mcPMTID`).

The `meta` branch of the output file should only have a single entry and contains information relevant for all simulated

events, such as the run-number, the run-time, the PMT positions, and the total number of simulated events. The data-structure for the "default" `meta` tree, that are always written to the file, are:

Name	Type	Description
runID	int	The run number, from the RUN table.
runType	ulong64	The run type, from the RUN table.
runTime	ulong64	The unix start time of the run.
dsentries	int	The total number of simulated events.
macro	string	The macro name.
pmtType	vector<int>	The list of PMT types.
pmtID	vector<int>	The list of PMT IDs.
pmtChannel	vector<int>	The list of PMT electronic channels.
pmtIsOnline	vector<bool>	The list of whether each PMT is online.
pmtCableOffset	vector<double>	The list of calibration cable offsets per PMT.
pmtChargeScale	vector<double>	The list of calibration charge scales per PMT.
pmtPulseWidthScale	vector<double>	The list of calibration pulse width scales per PMT.
pmtX	vector<double>	The list of PMT x positions.
pmtY	vector<double>	The list of PMT y positions.
pmtZ	vector<double>	The list of PMT z positions.
pmtU	vector<double>	The list of PMT x directions.
pmtV	vector<double>	The list of PMT y directions.
pmtW	vector<double>	The list of PMT z directions.
digitizerWindowSize	uint32	The length of the digitizer window.
digitizerSampleRate_GHz	double	The sampling rate of the digitizer, in GHz.
digitizerDynamicRange_mV	double	The dynamic range of the digitizer, in mV.
digitizerResolution_mVPerADC	double	The resolution of the digitizer, in mV/ADC.

The data-structure for the `output` tree is as follows. First, the "default" variables that are always written to the ntuple are detailed below. Note the distinction between simulated event and detector event, described in *DAQ Models*. The word "true" is used to indicate simulated quantities that are not affected by the detector response.

Name	Type	Description
mcpdg	int	Particle data code for highest energy particle.
mcx	double	True x position of the highest energy particle.
mcy	double	True y position of the highest energy particle.
mcz	double	True z position of the highest energy particle.
mcu	double	True x direction of the highest energy particle.
mcv	double	True y direction of the highest energy particle.
mcw	double	True z direction of the highest energy particle.
mcke	double	True kinetic energy of the highest energy particle.
mct	double	True time, relative to the start of the simulation, of the highest energy particle.
scintEdep	double	True total energy deposited in the scintillator (0 if no scintillator).
scintEdepQuenched	double	True total quenched energy deposited in the scintillator.
scintPhotons	int	True total number of scintillation photons produced.
remPhotons	int	True total number of re-emitted photons produced.
cherPhotons	int	True total number of Cherenkov photons produced.
mcid	int	The simulated event ID.
mcparticlecount	int	The true total number of simulated particles.
mcnhits	int	The true total number of PMTs that detected light.
mcpecount	int	The true total number of detector photoelectrons.
evid	int	The detector event ID.
subev	int	The ID of the event within a single simulated event.
nhits	int	The total number of PMTs that detected light in the detector event.
triggerTime	double	The trigger time of the detector event, relative to the start of the simulation.
timestamp	double	The UTC time of the detector event.
timeSinceLastTrigger_us	double	The time since the last triggered event, in microseconds.
event_cleaning_word	ulong64	The list of event cleaning cuts that failed.

If `include_mcparticles` is set then we additionally add the following information to the `output` branch of the ntuple. These are filled from the `DS::MCParticle` branch. This provides a method for looking at all simulated particles, rather than just the first.

Name	Type	Description
mcpdgs	vector<int>	Particle data code for all particles.
mcxs	vector<double>	True x position of all particles.
mcys	vector<double>	True y position of all particles.
mczs	vector<double>	True z position of all particles.
mcus	vector<double>	True x direction of all particles.
mcvs	vector<double>	True y direction of all particles.
mcws	vector<double>	True z direction of all particles.
mckes	vector<double>	True kinetic energy of all particles.
mcts	vector<double>	True time of each particle, relative to the start of the simulation.

If `include_pmthits` is set then we additionally add the following information to the `output` branch of the ntuple. Note that a DAQ system must also run in order for these variables to be filled, see *DAQ Models* for more details. The `hitPMT` variables are filled from the `RAT::DS::PMT`, described in *The ratpac-two Data Structure*.

Name	Type	Description
hitPMTID	vector<int>	The unique ID of each of the PMTs that detected light.
hitPMTTime	vector<double>	The hit-time of the first PE at each PMT.
hitPMTCharge	vector<double>	The charge for each PMT that detected light.

If include_digitizerhits is set then we additionally add the following information to the output branch of the ntuple. Note that a DAQ system and waveform analysis processor must also run in order for these variables to be filled, see *DAQ Models* and *Waveform analysis* for more details. The digitPMT variables are filled from the RAT::DS::DigitPMT, described in *The ratpac-two Data Structure*.

Name	Type	Description
digitPMTID	vector<int>	The unique ID of each of the PMT waveform that crossed threshold.
digitNhits	int	The total number of PMT waveforms that crossed threshold.
digitNhitsCleaned	int	The total number of PMT waveforms that passed all enabled hit cleaning processors and crossed threshold.
digitTime	vector<double>	The hit-time extracted from each PMT waveform.
digitCharge	vector<double>	The charge extracted from each PMT waveform.
digitNCrossings	vector<int>	The total number of times each PMT waveform crossed threshold.
digitTimeOverThresh	vector<double>	The total time each PMT waveform spent above threshold.
digitVoltageOverThr	vector<double>	The integrated voltage over threshold for each PMT.
digitPeak	vector<double>	The peak voltage of each PMT waveform.
digitLocalTriggerTi	vector<double>	Convenience variable to add per-PMT calibration timing.
digitReconNPEs	vector<int>	The total number of PEs per PMT, as estimated by a PE counting method.

If include_digitizerfits is set then we additionally add the following information to the output branch of the ntuple. Note that a DAQ system and waveform analysis processor must also run in order for these variables to be filled, see *DAQ Models* and *Waveform analysis* for more details. In particular, these are filled by a specific waveform analysis algorithm, such as the lognormal fits described in *Lognormal fitting*. The fitter will append its name to the variable name (e.g., fit_pmtid_lognormal). These are filled from the DS::WaveformAnalysisResult branch.

Name	Type	Description
fit_pmtid_+{fitter_name}	vector<int>	The unique ID of each of the PMT waveform that was fit.
fit_time_+{fitter_name}	vector<double>	The time extracted from each PMT waveform fit.
fit_charge_+{fitter_name}	vector<double>	The charge extracted from each PMT waveform fit.
fit_FOM_+{fitter_name}_+{fom_nam	vector<double>	The figure of merit extracted from each PMT waveform fit.

If include_nestedtubehits is set then we additionally add the following information to the output branch of the

ntuple. These "nested tubes" are intended for use with liquid-O style fiber optics detectors. These are filled from the `DS::MCNestedTube` branch.

Name	Type	Description
`mcnNTs`	int	Total number of nested tubes that detected light.
`mcnNThits`	int	Total number of PEs detector by nested tubes.
`mcNTid`	vector<int>	The unique ID of each true nested tube that detected light.
`mcNThittime`	vector<double>	The true time of each PE detected by a nested tube.
`mcNThitx`	vector<double>	The true x position for each PE detected by a nested tube.
`mcNThity`	vector<double>	The true y position for each PE detected by a nested tube.
`mcNThitz`	vector<double>	The true z position for each PE detected by a nested tube.
`ntId`	vector<int>	The unique ID for each detector nested tube that detected light.

If `include_nestedtubehits` is set then we additionally add the following information to the `meta` branch of the ntuple.

Name	Type	Description
`ntX`	vector<double>	The x position of the nested tubes.
`ntY`	vector<double>	The y position of the nested tubes.
`ntZ`	vector<double>	The z position of the nested tubes.
`ntU`	vector<double>	The x direction of the nested tubes.
`ntV`	vector<double>	The y direction of the nested tubes.
`ntW`	vector<double>	The z direction of the nested tubes.

If `include_mchits` is set then we additionally add the following information to the `output` branch of the ntuple. The `mcPMT` variables are filled from the `RAT::DS::MCPMT` branch, whereas the `mcPE` variables are filled from the `RAT::DS::MCPhoton` branch.

Name	Type	Description
`mcPMTID`	vector<int>	The unique IDs of the true hit PMTs.
`mcPMTNPE`	vector<int>	The true number of PEs for each hit PMT.
`mcPMTCharge`	vector<double>	The true charge deposited on each PMT.
`mcPEPMTID`	vector<int>	The unique ID of the PMT that detected each PE.
`mcPEHitTime`	vector<double>	The true detection time of each PE.
`mcPEFrontEndTime`	vector<double>	The true detection time, smeared by the sensor response, of each PE.
`mcPEProcess`	vector<string>	The process that created the photon that created the PE.
`mcPEx`	vector<double>	The true x position of the PE.
`mcPEy`	vector<double>	The true y position of the PE.
`mcPEz`	vector<double>	The true z position of the PE.
`mcPECharge`	vector<double>	The true charge of each PE.

If `include_tracking` is set then we additionally add the following information to the `output` branch of the ntuple. These variables are filled from the `RAT::DS::MCTrack` and `RAT::DS::MCTrackStep` branches. The variables below are mostly 2D vectors. The inner vector is the set of steps along the particle track (`RAT::DS::MCTrackStep`) and the outer vector is the set of tracks along the particle trajectory (`RAT:DS::MCTrack`). In other words, each track can have many steps, each of which as an associated position, momentum, process, and volume. As a reminder, `/tracking/storeTrajectory 1` must also be set in the macro in order to save the tracking information.

Name	Type	Description
trackPDG	vector<int>	The PDG code of the particle associated with this track.
trackPosX	vector<vector<double>>	The starting x position of each of the steps along the particle track.
trackPosY	vector<vector<double>>	The starting y position of each of the steps along the particle track.
trackPosZ	vector<vector<double>>	The starting z position of each of the steps along the particle track.
trackMomX	vector<vector<double>>	The starting x momentum of each of the steps along the particle track.
trackMomY	vector<vector<double>>	The starting y momentum of each of the steps along the particle track.
trackMomZ	vector<vector<double>>	The starting z momentum of each of the steps along the particle track.
trackKE	vector<vector<double>>	The kinetic energy of each of the steps along the particle track.
trackTime	vector<vector<double>>	The time, relative to the start of the simulation, of the particle steps.
trackProcess	vector<vector<int>>	The ID of the process that created the step.
trackVolume	vector<vector<int>>	The ID of the detector volume the step started in.

If include_tracking is set then we additionally add the following information to the meta branch of the ntuple.

Name	Type	Description
processCodeMap	map<string, int>	A map from process name to process ID.
volumeCodeMap	map<string, int>	A map from volume name to volume ID.

If include_digitizerwaveforms is set then we create a new branch in the ntuple called waveform that stores the full digitized waveforms. Note this will significantly increase the size of the files. The waveform branch will contain the following variables:

Name	Type	Description
evid	int	The event ID, repeated in the output tree.
waveform_pmtid	vector<int>	The unique PMT ID associated with the waveform.
inWindowPulseTimes	vector<double>	The list of MCPE front-end times that fall inside the waveform window.
inWindowPulseCharges	vector<double>	The list of MCPE charges that fall inside the waveform window.
waveform	vector<ushort>	The digitized waveform, per PMT.

If event_fitters specify that event reconstruction algorithm results should be included in the ntuple, then we add the following variables to the output branch of the ntuple. These are filled from the DS::EventFitResult branch. All fitter instances are labeled by the "full name" of the fitter instance, which is the name of the fitter type + the instance name of the fitter separated by double underscores (e.g., quadfitter__instance1). The variables are as follows:

Name	Type	Description
x_fitter__FULLNAME	double	X coordinate of the reconstructed event vertex.
y_fitter__FULLNAME	double	Y coordinate of the reconstructed event vertex.
z_fitter__FULLNAME	double	Z coordinate of the reconstructed event vertex.
u_fitter__FULLNAME	double	X component of the reconstructed event direction.
v_fitter__FULLNAME	double	Y component of the reconstructed event direction.
w_fitter__FULLNAME	double	Z component of the reconstructed event direction.
energy_fitter__FULLNAME	double	Reconstructed event energy.
time_fitter__FULLNAME	double	Reconstructed event time.
validposition_fitter__FULLNAME	bool	Whether the reconstructed event position is valid.
validdirection_fitter__FULLNAME	bool	Whether the reconstructed event direction is valid.
validenergy_fitter__FULLNAME	bool	Whether the reconstructed event energy is valid.
validtime_fitter__FULLNAME	bool	Whether the reconstructed event time is valid.
FOMNAME_fitter__FULLNAME	double	The figure of merit for the event fit.

OutNet

NOTE: This processor is not currently supported. The below documentation is outdated, but may provide some useful information.

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.

1.1.19 Off-line Analysis in ROOT

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.

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);
```

(continues on next page)

(continued from previous page)

```
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.

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.

1.1.20 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
```

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, " which are entirely unrelated!) A process represents some sort of interaction, like ionization, absorption, etc. While some of the process names are straightforward enough, like "eIoni" and "eBrem," others like "Attenuation" are misleading (implements both absorption '"and"' Rayleigh scattering). As a result of the interaction: * 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 GLG4sim code after all other processes have been run, but only if the track segment is contained in a scintillator region. The post-process does two things to the current track: 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.

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
  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
-------------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
 # |          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.

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.

1.1.21 Creating and Running *rattest* Tests

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 *$RATSHARE/test/*, with the functional tests in *$RATSHARE/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 *$RATSHARE/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.py* script takes the following options:

```
usage: rattest.py [-h] [-u] [-m] [-r] [-t] [--make-template TEMPLATE] input [input ...]

positional arguments:
  input                 RAT test(s) to run. Must be a directory or directories.

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.
  --make-template TEMPLATE
                        Write a template rattest to current directory for you to edit.␣
↪Supplied name is used for .mac and .C files.
```

Existing RAT Tests

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

Automated rattests

Every time a PR is submitted to ratpac-two, the rattests in $RATROOT/test/full (FIXME currently just acrylic_attentuation) are ran via GitHub actions using a Github hosted runner. PRs that do not pass all of the rattests will not be merged in, unless the reasons for the test failures are intended. If a test begins to fail because of an intended change, the *standard.root* file should be updated.

The workflow file that controls the running of the rattests can be found at $RATSHARE/.github/workflows/rattests.yml. To add a test, one must add another job to the workflow file using the same format used by the other rattests.

Downstream forks and experiment repositories are encouraged to create their own rattests using the same workflow setup and the *rattest.py* script.

Writing a RAT Test

1. Create a new folder in *$RATSHARE/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",
"run_range": [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 defaults to the ones in *$RATSHARE/ratdb*.

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

...
```

(continues on next page)

(continued from previous page)

```
/run/beamOn 500
```

You can also create a custom rat "experiment" in the test directory. This experiment can include any custom ratdb tables you want. You can tell rat to use this experiment by adding the line:

```
/rat/db/set DETECTOR experiment "cylinder"
```

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 with the same name as the macro ie *acrylic_attentuation(std::string event_file, std::string outfile)*. *rattest* will automatically fill in the function arguments when it calls the root macro.

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 acrylic_attenuation(std::string event_filename, std::string out_filename)
{
  TFile *event_file = new TFile(event_filename.c_str(),"READ");
  TTree *T = (TTree*)event_file->Get("T");
  TFile *out_file = new TFile(out_filename.c_str(),"RECREATE");

  TH1F *acr_attn_100 = new TH1F("acr_attn_100", "Photon track length (100 nm)", 50, 0,␣
↪50);
  acr_attn_100->SetXTitle("Track length (mm)");
  acr_attn_100->SetYTitle("Count");
  T->Draw("mc.track.GetLastMCTrackStep()->length>>acr_attn_100","TMath::Abs(1.
↪23997279736421566e-03/(mc.track.GetLastMCTrackStep()->ke)-100)<10","goff");
  //acr_attn_100->Fit("expo");
  //acr_attn_100->Draw("goff");
  acr_attn_100->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*.

1.1.22 Visualization with OpenGL-Qt

Visualizations using opengl allow graphics to be rendered and adjusted real-time in a relatively performant way. One can visualize the whole detector or slices of the detector with particle tracks to debug / demonstrate the detector.

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

/vis/open OGLSQt
/vis/scene/create
/vis/scene/add/trajectories #additionally can add rich and/or smooth
/tracking/storeTrajectory 1
/tracking/FillPointCont 1
/vis/scene/add/volume
/vis/scene/add/hits
/vis/sceneHandler/attach

/vis/viewer/set/upVector 0.0 0.0 1.0
/vis/viewer/set/viewpointThetaPhi 90 180
/vis/viewer/zoomTo 20
/vis/viewer/set/style s

## Cut a plane through the detector
/vis/viewer/addCutawayPlane -100 0 0 cm 1 0 0
```

Running rat

In order to keep rat from exiting the moment the macro completes, place rat into interactive mode. This can either be done standalone:

```
rat --vis
```

Or even in combination with a list of macros::
rat vis.mac –vis

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

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]

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)
```

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.

Examples

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

1.1.24 Tools

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.

1.1.25 Random Numbers

All of Ratpac 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 Ratpac. With this seeding scheme, two Ratpac 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 Ratpac, version 1.0
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.

Indices and tables

- genindex
- modindex
- search

1.2 Programmer’s Guide

Table of Contents

1.2.1 Extending the Data Structure

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.

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!

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; }
```

(continues on next page)

(continued from previous page)

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

  /// 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.

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.

1.2.2 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.

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
```

(continues on next page)

(continued from previous page)

```
  // link object. You can leave off the index if you just want
  // empty index ""
}

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().

1.2.3 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.

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.

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.

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");
```

1.2.4 Creating a Processor

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

Historically 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". Currently, not all processors follow this naming scheme. 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. Processors live in several of the ratpac-two subdirectories and are primarily organized by topic. For example, the reconstruction processors all live in `src/fit`.

In the header file, you find the new processor class (in this example, we call it `NewProc`) initialized as part of the *RAT* namespace as follows:

```
// Header guards
#ifndef __RAT_NewProc__
#define __RAT_NewProc__

#include <RAT/Processor.hh>

```

(continues on next page)

(continued from previous page)

```
namespace RAT {

class NewProc : public Processor {
 public:
  // Constructor
  NewProc();

  // Destructor
  virtual ~NewProc();
```

Then from the constructor we set the processor name that gets called from the macro:

```
NewProc::NewProc() : Processor("new_processor") {}
```

From the `BeginOfRun` method we can load any database paramters, setup variables, and prepare the processor. This runs once at the beginning of the run, prior to any events being processed. As an example, for the noise processors loads parameters from the database in `BeginOfRun`:

```
void NoiseProc::BeginOfRun(DS::Run *run) {
  DBLinkPtr lnoise = DB::Get()->GetLink("NOISEPROC");

  fNoiseFlag = lnoise->GetI("noise_flag");
  fDefaultNoiseRate = lnoise->GetD("default_noise_rate");
  ...

  DS::PMTInfo *pmtinfo = run->GetPMTInfo();
  UpdatePMTModels(pmtinfo);
  ...
}
```

In `BeginOfRun` the run object is passed, which allows users to grab the `DS::PMTInfo` or `DS::ChannelStatus`, as per the example above. In this example, the noise processor uses the `PMTInfo` to generate necessary information prior to the processor running.

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 (MC) information primarily, or need to consider all the detector events as a group, you should overload the `DSEvent` method:

```
Processor::Result DSEvent(DS::ROOT &ds);
```

As an example, the DAQ processors use the MC information to generate information about the event and decide whether to issue a trigger and create a detector event. Therefore the DAQ processors all overload the `DSEvent` method:

```
Processor::Result SplitEVDAQProc::DSEvent(DS::Root *ds);
```

If you are writing a processor that is primarily interested in detector (triggered) events, it may be easier to instead overload the `Event` method instead:

```
Processor::Result Event(DS::Root *ds, DS::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:

```
for (int i = 0; i < ds->GetEVCount(); i++) {
   DS::EV *ev = ds->GetEV(i);
   // Process events here
}
```

You should only overload `DSEvent` or `Event`, but NOT BOTH.

Next 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 probably put them into a RATDB table. Users can override RATDB values using the */rat/db/set* command in their macro files. If you do want parameters that can be set using */rat/procset*, 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.

Finally, the `EndOfRun` method is invoked once after all of the events have been processed, and can be used to clean-up variables or print summary statistics.

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 are implementing one of the parameter methods, you should use this general pattern:

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

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 macros. Include the header for your processor at the top, then find the relevant code in the constructor that looks like:

```
// Create processor allocator table
AppendProcessor<OutROOTProc>();
AppendProcessor<OutNtupleProc>();
AppendProcessor<NoiseProc>();
AppendProcessor<CountProc>();
```

and append your new processor.

1.2.5 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.

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
```

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>);
```

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>);
```

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>);
```

1.2.6 Extra Utilities

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) { };
```

(continues on next page)

(continued from previous page)

```
    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);

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

1.2.7 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.

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.

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.

Data Structure Classes

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

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.

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 ...
}
```

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.

Indices and tables

- genindex
- modindex
- search

1.3 Ratpac-two

1.3.1 Related Pages

1.3.2 Namespaces

Namespace List

Namespace Members

Namespace Members

Namespace Members

Namespace Members

1.3.3 Classes

Class List

Class Index

Class Hierarchy

Class Members

All

Class Members

Class Members

Class Members

1.3.4 Files

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search