# RadioScatter

Generated by Doxygen 1.8.13

# Contents

# Chapter 1

# Main Page

RADIO SCATTER

## 1.1 Introduction

this is the RadioScatter module. It has been made to run independently of, or within, GEANT4. It simulates RF scattering from points in 4 space, which may be ionization deposits left by particle showers produced in geant4. it is highly customizable. this documentation will be expanded.

## 1.2 Installation

### 1.2.1 Prerequisites

ROOT 6.08 or higher (`https://root.cern.ch/downloading-root`)

the module slac_rf is g4 source code that will produce histograms of voltages received in a scattering experiment. to run inside of GEANT4, you'll need to have that installed

GEANT4 (`https://geant4.web.cern.ch/geant4/support/download.shtml`) The GEANT4 software is fully independent of RadioScatter, and its license is available here: (`https://geant4.web.cern.ch/license/LICENSE.html`). Per that license: "This product includes software developed by Members of the Geant4 Collaboration ( http://cern.ch/geant4)."

you can use any monte-carlo program you want with RadioScatter, but this package includes some GEANT4 programs, built on their examples, that harness the full power of GEANT to make realistic rf scattering signals. if you want to use those, i'm gonna assume that you know how to use GEANT4 and ROOT.

### 1.2.2 Default installation

first be sure to have ROOT (as above) installed. there's lots of documentation on building those available elsewhere. and if you want to use it with GEANT, you'll need to have that installed too. then:

```
cd /your/favorite/source/dir  (like /usr/local or something)
git clone https://github.com/prchyr/RadioScatter.git
cd RadioScatter
./install.sh
```

this will install the header files and shared libraries (for root analysis) to the right places. it will put the libs inside of /usr/local/lib and the header files in /usr/local/include. It will make a build directory (inside the RadioScatter directory) called build, where the build products are placed before they are installed in the proper location.

### 1.2.3 Custom Installation

say you want your libraries and headers installed in some other place than /usr/local, you can set the RS_INSTA↩ L_DIR variable in your bashrc. then libraries will install to RS_INSTALL_DIR/lib and headers to RS_INSTALL_DI↩ R/include

```
cd /your/favorite/source/dir
git clone https://github.com/prchyr/RadioScatter.git
export RS_INSTALL_DIR=/your/preferred/install/dir
cd RadioScatter
./install.sh
```

tested on:

Ubuntu 16.04/gcc 5.4

Red Hat Enterprise server 6.9/gcc 4.8.5

Ubuntu 18.04/gcc 7.4

## 1.3 Usage

the theory of operation is outlined in this paper:

http://arxiv.org/abs/1710.02883

This document serves as the primary documentation, pulled from the source code.

how it works, breifly:

The RadioScatter module calculates rf scattering from an energy deposit in 4 space. It uses the position, energy deposited, region over which the energy was deposited, and the ionization energy of the material to get a number of free charges over which to perform the scatter, and a number density to get material effects (plasma screening, eave damping, etc.) it is most useful to have the deposits be from some monte-carlo generation, like GEANT4. you can probably fit it into any MC program with relative ease, or generate your own plasma cloud to scatter radio from.

### 1.3.1 Some Examples:

the examples directory has some use cases that don't involve geant4. to compile, be sure to link to the radio scatter library, i e -lRadioScatter, and also the root libraries, like so:

```
g++ -std=c++0x example.cc -o example `root-config --cflags --glibs --libs` -lRadioScatter
```

in the examples, there is a use case where a shower has been prepared using GEANT4 and then this is used in a standalone program to scatter radio, as an example of runnning radioscatter outside of a monte carlo program. various options are shown and explained in the example

### 1.3.2 GEANT4 usage

there is a GEANT4 module called slac_rf, so to install it, do

```
cd /your/GEANT/app/dir
mkdir slac_rf_build
cd slac_rf_build
cmake /path/to/RadioScatter/example/GEANT4/slac_rf
make -b -j4
```

which will make a build directory somewhere you want it to live, then install the GEANT4 simulation there linked to the RadioScatter libraries, once you've installed them as above.

how it works within GEANT4:

the Radioscatter.hh file is a header file that needs to be included in the GEANT4 source. there should be several different ways to use it, but one way is to globally declare the RadioScatter object in the main simulation .cc file of the simulation. it is then best to pass the constructed radio scatter object to the action initialization and the run action and stepping action files of the simulation. this will allow access to different methods that are useful at different points in the sumulation lifetime. for example, the data is filled during the stepping action phase, but the ROOT histograms are cosed at the end of the run action, through a built in GEANT4 method called EndOfRunAction.

there are also several commands available through the RSmessenger.hh/cc files which allow for manipulation of the radioscatter configuration through the GEANT4 macro files. things such as

/RS/setTxPos 5 0 3 m

and

/RS/setTxPower 100

/RS/setPlasmaLifetime 3

which can be used without the need to re-compile the GEANT4 source, making running simulations much more simple.

ok so to actually run the program,

```
cd /path/to/geant4/program/install/dir/
```

(default is RadioScatter/slac_build unless you provided an argument to install.sh)

```
./slac_rf -m run1.mac -f "/path/to/file/and/filename.root"
```

to set the root file name, (-f) which is optional. each run is stored in an entry in a tree. there is a default macro

run1.mac: -this will set the tx position and the energies and such, and then will simulate the received signal in 3 antennas

### 1.3.3 Some Use Cases

One use of RadioScatter is to generate a lot of events over some volume using a monte-carlo approach, and calculate information about each event. If you run within GEANT4 using one of the examples included, you simply do /RS/setFillByEvent 1 within your macro. This calls a function, radio->writeEvent() after each event. Coupled with setMakeSummary(1), this is a good way to get information that you can use to develop sensitivity curves, etc.

Another use of RadioScatter is to generate a statistically varied sample of events at the same vertex position, to achieve a high-resolution signal. Because RadioScatter calculates the scattering from each individual charge, we can't run over an actual 10^18 eV cascade, as it would take too long. running over a 100GeV cascade even begins to take a long time. but, you can run over a 1-10GeV cascade, which is fast, and then run multiple of them to get some statistical variation in the 4-positions of the ionization deposits. In the next section, we discuss how scaling works in RadioScatter to make this possible.

### 1.3.4 Scaling in RadioScatter

Because the radio signal is built up through calculation of scattering from the individual particles, the simulation time scales with energy. And, when running in GEANT4, there is an upper limit of about 100TeV for some of the physics processes. To simulate higher energies, RadioScatter employs empirical scaling of the density of the cascade, the length of the cascade, and the associated travel time of the lengthened cascade. We breifly discuss here how this is done.

There are two regimes in which scaling might be wanted.

1) for a collider experiment. instead of running 10^9 cascades to simulate a bunch of 1 billion primary particles, you might want to just run 1 cascade and scale the number density by 10^9 for computational efficiency.

2) for a UHE experiment. Instead of trying to simulate a 1PeV cascade, you'd simulate a 10GeV cascade and then set a target energy of 1PeV, and then scale both the number density and the physical scale of the cascade to the target energy.

#### 1.3.4.1 Running Independently of GEANT4 or other MC program

if running radioscatter on an independently generated energy deposition, scaling may not be required (for example, if you have generated a 3-d energy deposition profile for a cascade at 1 EeV, RadioScatter simply runs off of that). If, however, you use a lower energy cascade (10GeV, for example) and want to scale it up to a 1 EeV cascade, then scaling must be enabled. First, RadioScatter needs to know the energy of the primary particle of the cascade being simulated. so in this example, you'd set it to 10GeV (be sure to specify an energy unit). so,

setPrimaryEnergy(10∗TUtilRadioScatter::GeV);

then, to scale this up to the proper density, you set the target energy to 1 EeV which tells RadioScatter what the target energy is to make the calculations for scaling.

setTargetEnergy(1e9∗TUtilRadioScatter::GeV);

This essentially makes a scaling factor that each ionization deposity is multiplied by to increase the ionization densty to the target point (for details see arxiv:1710.02883). However, the profile of a 10 GeV particle and a 1 EeV particle are not the same, so the final step to make the scaling work is:

setScaleByEnergy(1);

which scales the cascade along the longitudinal dimension and the time dimension the appropriate amount for the given scaling.

Alternatively, if you are wanting to simulate a beam test in with there are a large number of primaries at a single energy, meaning you want to scale the density but not the physical/temporal extent of the cascade, you can use setNPrimaries but not scale with energy, like:

setNPrimaries(1e9); setScaleByEnergy(0);

#### 1.3.4.2 Running within GEANT4

inside of GEANT4 there is a different set of steps to set the correct scaling. It is important to remember that RadioScatter and GEANT4 are different programs running together, so there are some commands for one, and some commands for the other. when running in GEANT4, you need to tell GEANT4 the primary energy, not RadioScatter. so instead of setPrimaryEnergy() (RadioScatter command) you use the GEANT4 command in your macro,

/gps/energy 10GeV

this also sets the primaryParticleEnergy in GEANT, so you don't need to set that manually. You can, but it will be overwritten; this is to maintain internal consistsncy when running in G4 mode.

you can then set the target energy (RadioScatter command) as above, and then set the scaling (RadioScatter command) to 1 in your macro file like:

/RS/setTargetEnergy 1e9 GeV /RS/setScaleByEnergy 1

another thing you can do in GEANT4 is simulate more than one event. meaning, you can do

/run/beamOn(1)

or

/run/beamOn(n)

where n is any number of events that will be simulated with the set simulation parameters. so if you did

/gps/energy 10GeV /run/beamOn(10)

then it would simulate 10 events, each at 10GeV, at the same vertex position (set elsewhere). You have flexibility in how these events are saved. You can either set them to be saved as individual events, or you can have them averaged into a single event. this is done via

setFillByEvent(0) to have them averaged and setFillByEvent(1) to save each event individually.

when using setFillByEvent(0), the resultant radioscatter signal is scaled by 1/n, with n being the argument to /run/beamOn(n), such that the scatter you receive is the appropriate amplitude for the specified input energy. Why run like this? well, this increases the fideliity of the time-domain signal, by 'filling in' the scattering space randomly with different events, all of the same energy. This is different from using the nPrimaries scaling, which simply scales the number of ionization electrons at each point for a single cascade, because different events have different ionization electron geometry. you can think of it as the difference between increasing the brightness of a photo of a lightning strike and a long-exposure photo of lightning strikes. a single lightning strike has arms in random positions. to scale up the 'density' of the lightning, you could simply increase the brightness of that single strike by some value of n strikes. but in the case of numerous lightning strikes starting from the same place, the overall integrated brightness would increase by n strikes, but the random arms would slowly fill in the frame, as a time-integrated collection of the many strikes. This is essentially the difference between using setNPrimaries [increasing the brightness] and beamOn(n)[long exposure]. The resultant signal will have the same(ish) amplitude, because radar only cares about the total number of scattering centers, but the long-exposure method will be more robust in terms of signal fidelity, because the more filled-out the scattering suface is, the cleaner the reflected signal.

so why use nPrimaries ever? well, it's fast. ideally you'd just do /run/beamOn(1e8) and you're done, but this would take years to run. so it is up to the user to define what kind of signal fidelity is needed for a given simulation, and set all of these various parameters appropriately for their use case.

#### 1.3.4.3 General Guidance

for a general usage:

a 10 GeV input cascade results in decent signal fidelity. This is what has been used as the base energy for cascades simulated for RET-N. for T576, we used a 10-13.6GeV primary (depending on the actual beam energy) and then did /run/beamOn(10) to get a higher-fideilty signal. we found that beyond that, the time domain signal did not change in any noticeable way by increasing the argument to beamOn(), and only increased the comuptational time. [note: remember that the resultant RF signal is scaled by 1/n. so by doubling the number to beamOn you are not doubling the signal amplitude, just the signal fidelity, because the resultant amplitude is scaled back to reflect the added cascade(s)]. a 1GeV cascade will have fairly significant shot-to-shot differences, but is very fast [speed is linear in energy/particle number, 10GeV is 10x slower than 1GeV], so if number is your game and you can accept some error on the scattered signal, 1GeV may be sufficient. This can be appropriate for monte-carlo studies.

enjoy. questions can be sent to prohira dot 1 at osu dot edu

# Chapter 2

# Hierarchical Index

## 2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

# Chapter 3

# Data Structure Index

## 3.1 Data Structures

Here are the data structures with brief descriptions:

# Chapter 4

# Data Structure Documentation

## 4.1 RadioScatter Class Reference

This is the main workhorse of RadioScatter. It is a class that contains all of the functions needed to scatter radio from a particle cascade and store the results in ROOT files.

```
#include <RadioScatter.hh>
```

### Public Member Functions

- RadioScatter ()
- void makeOutputFile (TString filename)

    *creates the output file.*
- void makeOutputTextFile (char ∗filename)

    *unused*
- void writeToTextFile ()

    *unused*
- void setMakeSummary (double val)

    *flag saying to make a useful summary file.*
- int setNTx (double n)

    *set the number of transmitters.*
- int setNRx (double n)

    *set the number of receivers. this can be any arbitrary number.*
- void setTxPos (double xin, double yin, double zin, int index=0)

    *set the transmitter position. the index goes from 0 to ntx-1*
- void setRxPos (double xin, double yin, double zin, int index=0)

    *set the receiver position. the index goes from 0 to nrx-1*
- void setTxPos (TVector3 in, int index)

    *set the tx position using an TVector3 object for a specific index*
- void setTxPos (TVector3 in)

    *set the tx position using an TVector3 object, but with built-in indexing (e.g. each time this is called the next tx will be set up to ntx-1);*
- void setRxPos (TVector3 in, int index)

    *set the rx position using an TVector3 object*
- void setRxPos (TVector3 in)

*set the rx position using an TVector3 object, but with built-in indexing (e.g. each time this is called the next tx will be set up to nrx-1);*

- void setTxFreq (double f)

  *set the transmitter frequency*
- void setTxVoltage (double v)

  *set the transmitter voltage. can be superceded by setting power or vice versa (V)*
- void setTxPower (double p)

  *set the tx power (W)*
- void setNPrimaries (double n)

  *set the number of primaries.*
- void setInelasticity (double y=1.)

  *set the inelasticity (y) of this event.*
- void setReceiverGain (double gain)

  *set the receiver gain in dB*
- void setRxGain (double gain)

  *set the receiver gain in dB*
- void setTransmitterGain (double gain)

  *set the transmitter gain in dB*
- void setTxGain (double gain)

  *set the transmitter gain in dB*
- void setPrimaryEnergy (double e)

  *set the energy of the primary.*
- void setPrimaryPosition (TVector3 p)

  *set the position of the primary. useful for several calculations*
- void setPrimaryDirection (TVector3 d)

  *set the direction of the primary.*
- void setTargetEnergy (double e)

  *not used*
- void setCrossSection (double val)

  *not used*
- void setWeight (double val)

  *set the weight of the event (if used);*
- int setScaleByEnergy (double val)

  *use this flag to scale the shower by some amount.*
- int scaleByEnergy ()

  *sets the scaling by energy.*
- void setPlasmaLifetime (double l)

  *set the plasma lifetime in nanoseconds*
- void setPolarization (const char ∗p)

  *set the antenna polarization. very primitive now: vertical=(0,1,0), horizontal=(0,0,1). TODO: fix this.*
- void setTxVals (double f, double power, double gain)

  *not really useful. set them individually instead.*
- void setRxVals (double s, double gain)

  *not really useful. set them individually instead.*
- void **setSimulationParameters** (double n, char ∗tx_rx_pol, double relative_index_of_refraction, int flag)
- void setIndexOfRefraction (double iof)

  *This sets the index of refraction for the medium. assumes TX and RX are in this same medium.*
- void setRelativeIndexOfRefraction (double iof)

  *this is n1/n2 for n1>n2. used for refraction calculations when the tx and rx are in different media. DO NOT USE FOR TX AND RX IN SAME MEDIA, FOR THAT USE setIndexOfRefraction*
- void setCalculateUsingAttnLength (double val=0.)

*set to calculate the RF propagation with attenuation losses*

- void setRecordWindowLength (double nanoseconds)

  *set the length of the save window in ns*
- void setRxSampleRate (double rate)

  *set the sample rate of the receiver in GS/s*
- void setTxInterfaceDistX (double dist, int index=0)

  *same*
- void setRxInterfaceDistX (double dist, int index=0)

  *only used for refraction calculations when tx and rx are in different media.*
- void setShowCWFlag (double i)

  *set this flag to show the pure CW from transmitter to receiver. default is to have it off.*
- void setTxOnTime (double on)

  *set the time the transmitter is on. useful for pulsed CW, but otherwise don't mess with it and the program assumes constant CW.*
- void setTxOffTime (double off)

  *set the time the tx is off. see above*
- void setFillByEvent (double i)

  *when running in GEANT4, save individual events (1) or average over all events in a run (0)*
- void setFillParticleInfo (double i)

  *save the tuples in GEANT4 that are filled with information about the particle tracks and steps. slow and will eat big memory so be careful.*
- void setParticleInfoFilename (char ∗filename)

  *set the filename for the above information.*
- TVector3 getTxPos (int index=0)

  *get the transmitter position*
- TVector3 getRxPos (int index=0)

  *get the receiver position*
- double getFreq ()

  *get the transmitter frequency*
- double getTxGain (int index, double angle)

  *get the transmitter gain at a certain angle (not implemented)*
- double getRxGain (int index, double angle)

  *get the receiver gain at a certain angle (not implemented)*
- double makeRays (TLorentzVector point, double e, double l, double e_i)

  *the main function to do the actual scattering.*
- double makeRays (TLorentzVector point, double e, double l, double e_i, TH1F ∗hist)

  *optional to include a histogram to fill.*
- void printEventStats ()

  *print out some event statistics. not used much.*
- std::vector< std::vector< TH1F ∗ > > scaleHist (float num_events)

  *scale the histogram (when averaging over several events)*
- int writeRun (float num_events=1., int debug=0)

  *write a full run.*
- int writeEvent (int debug=0)

  *write a single event.*
- int makeSummary (TFile ∗f)

  *make a useful summary file from the full RadioScatterEvent*
- void close ()

  *close*

## Data Fields

- RadioScatterEvent **event**
- TString **output_file_name**
- const char ∗ pol = (char∗)"horizontal"

  *default, also set as default in set_simulation_paramaters*
- TString **polarization** ="horizontal"
- double **x_offset** =0∗m
- double **z_offset** = 2.∗m
- double y_offset = 5.∗m

  *unused*
- double tcs = .655e-24

  *thompson cross section*
- double collisionalCrossSection = 3.e-13

  *mm^-3, from NIST, converted in to mm^-3*
- double n_primaries = 1

  *set this based on number of events in run*
- double tx_voltage = 1.

  *V.*
- double zscale =1.

  *the longitudinal scale factor*
- double tscale =1.

  *the time scale factor*
- double impedance = 50

  *ohms*
- double tx_gain =3.

  *transmitter gain default*
- double rx_gain =3.

  *receiver gain default*
- double step_length =1

  *default g4 steplength*
- double E_i =.000038

  *default electron ion pair energy*
- double e_charge_cgs = 4.803e-10

  *statcoloumbs*
- double e_mass_cgs = 9.109e-28

  *g*
- double k_b = 1.38e-23

  *j/k*
- double c_light_mns =c_light/m

  *c light in m/ns*
- double plasma_const = 4.∗pi∗e_charge_cgs∗e_charge_cgs/e_mass_cgs

  *e^2/(4pi epislon0 m c^2), in units of m*
- double **e_radius** =classic_electr_radius/m
- double nu_col = 0

  *collisional frequency*
- double half_window = 300

  *number of nanoseconds in 1/2 of the record window. can be changed;*
- int useAttnLengthFlag =0

  *use attenuation length?*
- double attnLength =1400.

*average length for upper 1.5km at pole. is changeable*

- double **frequency** =1
- double **period**
- double **lambda**
- double **k**
- double **omega**
- double phase0 =0.

  *these are changed based on the gain, so the defaults here are meaningless.*
- double **rxEffectiveHeight** =1.
- double **txEffectiveHeight** =1.
- double **txFactor** =1.
- double rxFactor =1.

  *receiver stuff*
- double **samplerate** =10
- double **samplingperiod** =.1
- double **start_time** =0
- double end_time =1000

  *plasma lifetime, set to the samplingperiod by default*
- double lifetime =.1

  *timing variables for pulsed CW*
- double **txp**
- double **tx_on** =-999999999.
- double **tx_off** =999999999.
- int includeCW_flag =0

  *misc constants*
- std::vector< double > **amplitudes**
- std::vector< double > **timeofarrival**
- std::vector< double > **phases**
- std::vector< double > **field**
- std::vector< double > plasma

  *variables for refraction manipulation*
- double **k_r**
- double **mag1**
- double mag2

  *, tof, txphase, kx;*
- double c_light_r =c_light

  *distance from the antennas to the interface, must be set by user*
- std::vector< double > **tx_interface_dist** {1}
- std::vector< double > **rx_interface_dist** {1}
- double n_rel =1.5

  *relative index of refraction, calculated to always be >1.*
- TH1F ∗ **fft_hist**
- TH1F ∗ **power_hist**
- TH1F ∗ **t_h** = new TH1F("eventHist", "eventHist", 100, 0, 10)
- TH1F ∗ **re_h** = new TH1F("reHist", "reHist", 100, 0, 10)
- TH1F ∗ **im_h** = new TH1F("imHist", "imHist", 100, 0, 10)
- TGraph ∗ **event_gr** = new TGraph()
- std::vector< std::vector< TH1F ∗ > > **time_hist**
- std::vector< std::vector< TH1F ∗ > > **re_hist**
- std::vector< std::vector< TH1F ∗ > > **im_hist**
- std::vector< std::vector< TGraph ∗ > > **event_graph**
- TH1F ∗ **testHist0**
- TH1F ∗ testHist1

*public:*

- int **ntx** =1
- int **nrx** =1
- std::vector< TLorentzVector > tx {1}

  *transmitters, allow for multiple*
- std::vector< TLorentzVector > rx {1}

  *recievers, allow for multiple*
- int **TX_ITERATOR** =0
- int **RX_ITERATOR** =0
- int **FILL_BY_EVENT** =1
- int **FILL_PARTICLE_INFO** =0
- TString **PARTICLE_INFO_FILENAME** =""
- int **MAKE_SUMMARY_FILE** =0
- int **TX_GAIN_SET** =0
- int **RX_GAIN_SET** =0
- int **TX_FREQ_SET** =0
- int **RX_FREQ_SET** =0
- int **NPRIMARIES_SET** =0
- int TARGET_ENERGY_SET =0

  *indicates that the number of primaries has been set.*
- int PRIMARY_ENERGY_SET =0

  *inicates that a target energy for scaling has been set.*
- int SCALE_BY_ENERGY =0

  *indicates that the primary particle energy is known to radioscatter*
- int ENERGY_SCALING_SET =0

  *sets the flag to scale by energy.*
- int REAL_DATA =0

  *indicates that the energy scaling has been set.*

### 4.1.1 Detailed Description

a word about units.

GEANT uses mm and ns as the length and time, but for RF stuff things are best defined in m. so for length calculations as they pertain to RF fields, the lengths are converted into meters.

velocity in geant is mm/ns, so for things like phase calculations we employ these native units.

we use volts for the fields. we use watts for power units.

if you set a distance for radioscatter, always multiply by the unit, like 100$*$m for 100 meters. for RF units like volts and watts, just use the default RadioScatter (e.g. just provide the number)

### 4.1.2 Constructor & Destructor Documentation

#### 4.1.2.1 RadioScatter()

```
RadioScatter::RadioScatter ( )
```

default constructor.

### 4.1.3 Member Function Documentation

#### 4.1.3.1 makeOutputFile()

```
void RadioScatter::makeOutputFile (
             TString filename )
```

this is a mandatory call that needs to come before the others. it sets the output file and makes a RadioScatterEvent object that is filled with all of the outpus.

#### 4.1.3.2 makeRays()

```
double RadioScatter::makeRays (
             TLorentzVector point,
             double e,
             double l,
             double e_i )
```

this fuction are all that you need to call (for each point you want to scatter radio from.)

just give it a 4 vector, the energy deposited in a region, one length of that region, and the ionization energy of the material. It calculates the volume (using the given length) into which the energy has been deposited to calculate an ionization density. This density is used to inform macroscopic parameters of the scattering, plasma screening effects, and so on.

**Parameters**

| | |
|---|---|
| *point* | 4 vector indiciting the x, y, z, t position of this energy deposit |
| *e* | the deposited energy. |
| *l* | a characteristic length of the energy deposition. if the energy deposition is a dE/dx, then this is the length over which x is integrated to give an energy. |
| *e_i* | the ionization energy of the material. |

#### 4.1.3.3 scaleByEnergy()

```
int RadioScatter::scaleByEnergy ( )
```

this function actually does the scaling. prior to it being called, it makes sure that all of the information is there, namely, the energy of the actual primary particle that makes the cascade, and the number of primaries, either via nprimaries or targetenergy.

#### 4.1.3.4 setNPrimaries()

```
void RadioScatter::setNPrimaries (
             double n )
```

this is essentially a scaling factor used to achieve higher-energy showers than GEANT can provide in a reasonable time. for a 10PeV shower, for example, you could simulate a 10GeV primary and then set nPrimaries to 1e7, to get the equivalent density of ta 10PeV shower. to get the longitudinal profile correct, you'd need to setScaleByEnergy = 1 below.

default is 1.

#### 4.1.3.5 setNTx()

```
int RadioScatter::setNTx (
            double n )
```

sets the number of transmitters. currently only 1 is allowed, but we've included framework to add more in future.

#### 4.1.3.6 setPrimaryEnergy()

```
void RadioScatter::setPrimaryEnergy (
            double e )
```

this is important when running over some arbitrary shower input file, as there is no way for radioscatter to know the energy of the primary. it is used for output files and stuff and for scaling factors etc.

#### 4.1.3.7 setScaleByEnergy()

```
int RadioScatter::setScaleByEnergy (
            double val )
```

use this flag to scale the shower by some amount. it then scales the shower accordingly in the longitudinal direction.

#### 4.1.3.8 setTargetEnergy()

```
void RadioScatter::setTargetEnergy (
            double e )
```

not used

#### 4.1.3.9 setTxInterfaceDistX()

```
void RadioScatter::setTxInterfaceDistX (
            double dist,
            int index = 0 )
```

only used for refraction calculations when tx and rx are in different media.

### 4.1.4 Field Documentation

#### 4.1.4.1 attnLength

```
double RadioScatter::attnLength =1400.
```

misc rf constants

#### 4.1.4.2 c_light_mns

```
double RadioScatter::c_light_mns =c_light/m
```

4∗pi e^2/m_e

#### 4.1.4.3 includeCW_flag

```
int RadioScatter::includeCW_flag =0
```

whether to simulate the direct signal as well.

#### 4.1.4.4 n_rel

```
double RadioScatter::n_rel =1.5
```

some histograms

#### 4.1.4.5 rx

```
std::vector<TLorentzVector> RadioScatter::rx {1}
```

flags

The documentation for this class was generated from the following files:

- include/RadioScatter.hh
- src/RadioScatter.cc

## 4.2 RadioScatterEvent Class Reference

This is the storage class for a RadioScatter object, called an event. an event is a single scatter from a cascade, as detected in all of the receivers.

```
#include <RadioScatterEvent.hh>
```

## Public Member Functions

- TH1F ∗ **getComplexEnvelope** (int txindex, int rxindex, double cutoff=0)
- TGraph **getLowpassFiltered** (int txindex, int rxindex, double cutoff)
- TGraph ∗ **getGraph** (int txindex, int rxindex)
- TH1F ∗ **getSpectrum** (int txindex, int rxindex, bool dbflag=false)
- void **spectrogram** (int txindex, int rxindex, Int_t binsize=128, Int_t overlap=32)
- int plotEvent (int txindex, int rxindex, double noise_flag=0, int show_geom=0, int bins=256, int overlap=128, int logFlag=2)
- int plotEventNotebook (int txindex, int rxindex, double noise_flag=0, int show_geom=0, int bins=64, int overlap=8, int logFlag=2)
- int **reset** ()
- double **thermalNoiseRMS** ()
- double **chirpSlope** ()
- double **startFreq** ()
- double **stopFreq** ()
- double **sineSubtract** (int txindex, int rxindex, double rangestart=0, double rangeend=240, double p0=.02, double p1=1., double p2=0.)
- int **backgroundSubtract** (int txindex, int rxindex, TH1F ∗bSubHist)
- TH1F ∗ **makeBackgroundSubtractHist** (int txindex, int rxindex, TString bfile)
- double **peakFreq** (int txindex, int rxindex)
- double **bandWidth** ()
- double **peakV** (int txindex, int rxindex)
- double **effectiveCrossSection** (int txindex, int rxindex)
- double **rms** (int txindex, int rxindex)
- double **duration** (int txindex, int rxindex, double highThreshRatio=.3, double lowThreshRatio=.1)
- double integratedPower (int txindex, int rxindex)

  *the duration of a pulse. expressed in terms of a threshold as a ratio of the peak voltage. so here the high threshold is .3∗peakV, and the low threshold is .1∗peakV.*
- double **integratedPower** (int txindex, int rxindex, double tlow, double thigh, double dcoffset=0.)
- double **integratedPowerAroundPeak** (int txindex, int rxindex, double window=100)
- double **integratedVoltage** (int txindex, int rxindex)
- double **integratedVoltage** (int txindex, int rxindex, double tlow, double thigh, double dcoffset=0.)
- double **peakPowerMW** (int txindex, int rxindex)
- double **peakPowerW** (int txindex, int rxindex)
- double **pathLengthM** (int txindex, int rxindex)
- double **pathLengthMM** (int txindex, int rxindex)
- double **primaryParticleEnergy** ()
- int **triggered** (double thresh, int n_antennas=1)
- int **nTriggered** (double thresh)
- int **trigSingle** (double thresh, int ant=0)
- int **buildMap** ()
- TLorentzVector **pointUsingMap** ()
- **ClassDef** (RadioScatterEvent, 5)

## Data Fields

- TVector3 **direction** =TVector3(0,0,1)
- TVector3 **position** =TVector3(1, 1, 1)
- std::vector< TLorentzVector > **tx**
- std::vector< TLorentzVector > **rx**
- double **primaryEnergy** =1∗GeV
- double **targetEnergy** =1∗GeV
- double inelasticity =1.

*1GeV*

- double **weight** =1.
- double **sampleRate**
- double **nPrimaries** =0
- double **txVoltage** =0
- double **txPowerW** =0
- double **freq** =0
- double **txGain** =1.
- double **rxGain** =1.
- std::vector< std::vector< double > > **beta**
- std::vector< std::vector< double > > **delta**
- std::vector< std::vector< double > > **doppler**
- double **totNScatterers** =0
- std::vector< std::vector< TH1F ∗ > > **eventHist**
- std::vector< std::vector< TH1F ∗ > > **reHist**
- std::vector< std::vector< TH1F ∗ > > **imHist**
- std::vector< TH1F ∗ > **testHist**
- std::vector< std::vector< TGraph ∗ > > **eventGraph**
- int **SINE_SUBTRACT** =0
- int **ntx** =1
- int **nrx** =1

### 4.2.1 Detailed Description

The RadioScatterEvent is a TObject, meaning that it is easily stored in a ROOT tree and is plottable from ttree->Draw(). It has many member variables that store information about the cascade, the geometry of the receiver(s) and transmitter(s), and some basic variables about the events that can be easiliy plotted. But it also stores the raw waveforms captured in each receiver, which can be used in analysis.

### 4.2.2 Member Function Documentation

#### 4.2.2.1 plotEvent()

```
int RadioScatterEvent::plotEvent (
             int txindex,
             int rxindex,
             double noise_flag = 0,
             int show_geom = 0,
             int bins = 256,
             int overlap = 128,
             int logFlag = 2 )
```

TCanvas ∗c = new TCanvas("plotEvent", "plotEvent", 800, 400);

#### 4.2.2.2 plotEventNotebook()

```
int RadioScatterEvent::plotEventNotebook (
             int txindex,
             int rxindex,
             double noise_flag = 0,
             int show_geom = 0,
             int bins = 64,
             int overlap = 8,
             int logFlag = 2 )
```

TCanvas *c = new TCanvas("plotEvent", "plotEvent", 800, 400);

The documentation for this class was generated from the following files:

- include/RadioScatterEvent.hh
- src/RadioScatterEvent.cc

## 4.3 RSEventSummary Class Reference

This is a storage class of summary variables of a RadioScatterEvent, that are useful for plotting and calculations.

```
#include <RSEventSummary.hh>
```

### Public Member Functions

- **RSEventSummary** (int ntransmitters=1, int nreceivers=1)
- int **triggered** (double thresh, int n_antennas=1)
- int **nTriggered** (double thresh)
- int **trigSingle** (double thresh, int txx=0, int rxx=0)
- **ClassDef** (RSEventSummary, 1)

### Data Fields

- TVector3 **direction**
- TVector3 **position**
- vector< TLorentzVector > **tx** {1}
- vector< TLorentzVector > **rx** {1}
- double **primaryEnergyG4** =0
- double **sampleRate** =0
- double **nPrimaries** =0
- double **txVoltageV** =0
- double **txPowerW** =0
- double **freq** =0
- double **weight** =1.
- double **totNScatterers** =0
- double **primaryParticleEnergy** =0
- double **inelasticity** =1
- int **ntx** =1
- int **nrx** =1

- vector< vector< double > > **peakFreq**
- vector< vector< double > > **peakV**
- vector< vector< double > > **effectiveCrossSection**
- vector< vector< double > > **rms**
- vector< vector< double > > **duration**
- vector< vector< double > > **integratedPower**
- vector< vector< double > > **peakPowerW**
- vector< vector< double > > **pathLengthM**
- std::vector< std::vector< double > > **beta**
- std::vector< std::vector< double > > **delta**
- std::vector< std::vector< double > > **doppler**

### 4.3.1 Detailed Description

The summary class is useful for things like effective volume calculations, where a large number of events are run, and the storage space for the individual waveform data for each event and each antenna would be prohibitive. This allows for some useful variables to be calculated for each event to be stored in this object for analysis.

An RSEventSummary is created when setMakeSummary(1) is called from a radioscatter object

The documentation for this class was generated from the following files:

- include/RSEventSummary.hh
- src/RSEventSummary.cc

## 4.4 TUtilRadioScatter::TVec1D< T > Class Template Reference

### Public Member Functions

- **TVec1D** (int N)
- **TVec1D** (int N, T const &init)
- void **push_back** (T const &elem)
- T **size** ()
- T & **operator[ ]** (int index)
- void **clear** ()
- **ClassDefNV** (TVec1D, 1)

The documentation for this class was generated from the following file:

- include/TUtilRadioScatter.hh

## 4.5 TUtilRadioScatter::TVec2D< T > Class Template Reference

### Public Member Functions

- **TVec2D** (int N)
- **TVec2D** (int N, int M)
- **TVec2D** (int N, int M, T const &init)
- TUtilRadioScatter::TVec1D< T > & **operator[ ]** (const int index)
- void **clear** ()
- **ClassDef** (TVec2D, 1)

The documentation for this class was generated from the following file:

- include/TUtilRadioScatter.hh