JULIeT Users Manual Version 3.3

Java-based Ultra high energy Lepton IntEgral Transporter

Rie Ishibashi

Shigeru Yoshida

Mio Ono

Contents

List of Figures

List of Tables

Chapter 1

New Features in JULIeT version 3.x

1.1 Version 3.0

The new features added in the version 3.0 are:

1. JulietEventGenerator.java in the event package (see Chapter 11) is now capable of propagating neutrinos with the enhanced cross sections weighed by a factor that can be set by setNeutrinoInteractionWeight(). The corresponding change was made in InteractionsBase.java in the interaction package so that the path length of the neutrino can be sampled in a correct way even when the cross section is not necessarily small.

2. A new class, CELbeta.java, is now in the interaction package (Chapter 6). It gives the inelasticity coefficient $[cm^2/g]$ of muon as a function of energy.

3. PropagationMatrixFactory.java is a new member in the propagation package (Chapter 8). It provides an easy-to-use interface to read out the pre-calculated propagation matrix data.

4. The muon model package (Chapter 13) has been added. The atmospheric muon fluxes both at the earth surface and the deep earth including their propagation effects are calculated in the relevant classes in this package. The empirical model of the muon bundles with the parameters suggested by the data of the IceCube neutrino observatory provides the foundation. This package, together with the neutrino model package (Chapter 12) is used to calculate primary particle fluxes that enters into earth. The flux *after* the propagation in the earth is also calculated in the relevant classes in these packages using the propagation matrix in the propagation package (Chapter: 8).

5. The analysis package (Chapter 14) has been included. This package provides the tools to analyze the IceCube real and MC data using JULIeT libraries. The MC data generated by JULIeT stored in the data class I3Particle.java in the Particle package (Chapter: 5) are processed to calculate the IceCube event rate and sensitivity based on the propagation matrix data pre-calculated and stored by PropagationMatrix.java in the propagation package. Classes to make plots like zenith angle and NPE distributions in either 1 dimensional or 2 dimensional way are also available. The plotting capability is provided by the JAIDA software developed by SLAC (**http://java.freehep.org/).**

1.2 Version 3.3

The new features added in the version 3.3 are:

1. The Glashow resonance is added in the interaction package (see Chapter 6). This interaction is included in both the propagation package (Chapter 8) for numerical calculation and the event package (Chapter 11) for Monte-Carlo simulation.

2. The geometry can be shifted by a given amount in the MC data generation by JulietEventGenerator class (Section 11.3). The geometry package (Chapter 10) was changed to handle the geometry shift.

3. The random generator can now take a fixed seed number if necessary.

4. PropagatingNeutrinoFlux is now able to calculate fluxes of various neutrino models once it reads out relevant matrix data file.

5. Cross section of neutrino CC and NC interaction can be enhanced by a given factor in calculation of propagation matrix (Chapter 8), in order to study weak interaction model.

Chapter 2

Introduction

The JULIeT (Java-based Ultrahigh-energy Lepton IntEgral Transporter) is the Java-based Monte-Carlo/Numerical Simulation Package for calculation on transportation of extremely high energy(EHE) leptons in the earth. The JULIeT can run particles in energy range from 10^5[GeV] to 10^{12}[GeV], and trace the EHE lepton energy distribution during their travel through rock and/or ice. There are several methods for you to utilize this package. The easiest way would be the interactive mode to run the simulation. How to submit your run in the interactive way is described in Chapter3 together with a brief introduction to the JULIeT package. Details of the individual packages with the Java classes in the JULIeT package are then described in the following Chapters. You can find the details also in **http://www.ppl.s.chiba-u.jp/JULIeT/**.

2.1 Installation and Quick Run

Let us assume you install the JULIeT under a directory, ~/java_lib/. Unpacking the Jar-ball file there would give all the class files in ~/java_lib/classes/ and all the source files in ~/java_lib/sources/. There are some data files in ~/java_lib/data/ which can be used in some optional run the JULIeT provides.

An example of Compile Command:
at ~/java_lib/sources/iceCube/uhe/event/
```
% javac -sourcepath ~/java_lib/sources/  -classpath ~/java_lib/classes/
-d ~/java_lib/classes RunJuliet.java
```

An example of Execute Command:
```
java -Xms128m -Xmx256m -classpath ~/java_lib/classes/
iceCube.uhe.event.RunJuliet
```
(You can run the RunJuliet at any directory.)

You need '`-Xmx`' option to allocate larger memory for the `RunJuliet`. This example implies 256 Mbyte memory can be allocated in maximum. The results is saved in file you specified in F2K format so that you can "cat" it to have a look.

2.2 Structure of the JULIeT

The JULIeT is written in Java. Fig.2.1 shows structure of the packages. Each package has classes under the similar concepts. The representative packages among them are "event" and "propagation" both of which has its own method to run the simulation. The event package is a group of the classes to calculate the particle propagation with Monte-Carlo method. It has the interactive mode to run the simulation as we briefly explained in the previous section. The propagation package contains the classes to calculate

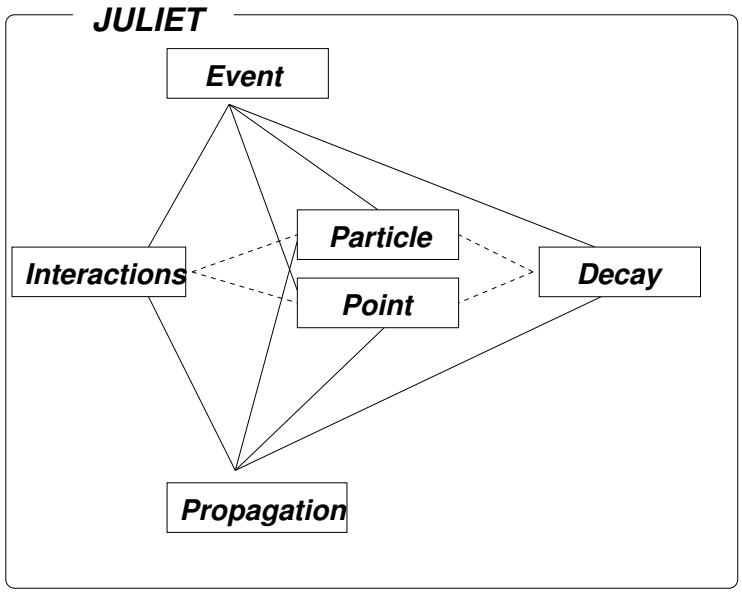

Figure 2.1: Package Structure of the JULIeT.

the particle propagation by numerically solving the relevant transport equations. The benefit is, needless to say, to save CPU time for calculating the propagation over long distances. The event and propagation packages rely on classes in all the other packages, such as the interaction package which is responsible for handling the particle interactions. The relation between the classes are shown in Fig. 2.2 for the event package, and Fig. 2.3 for the propagation package.

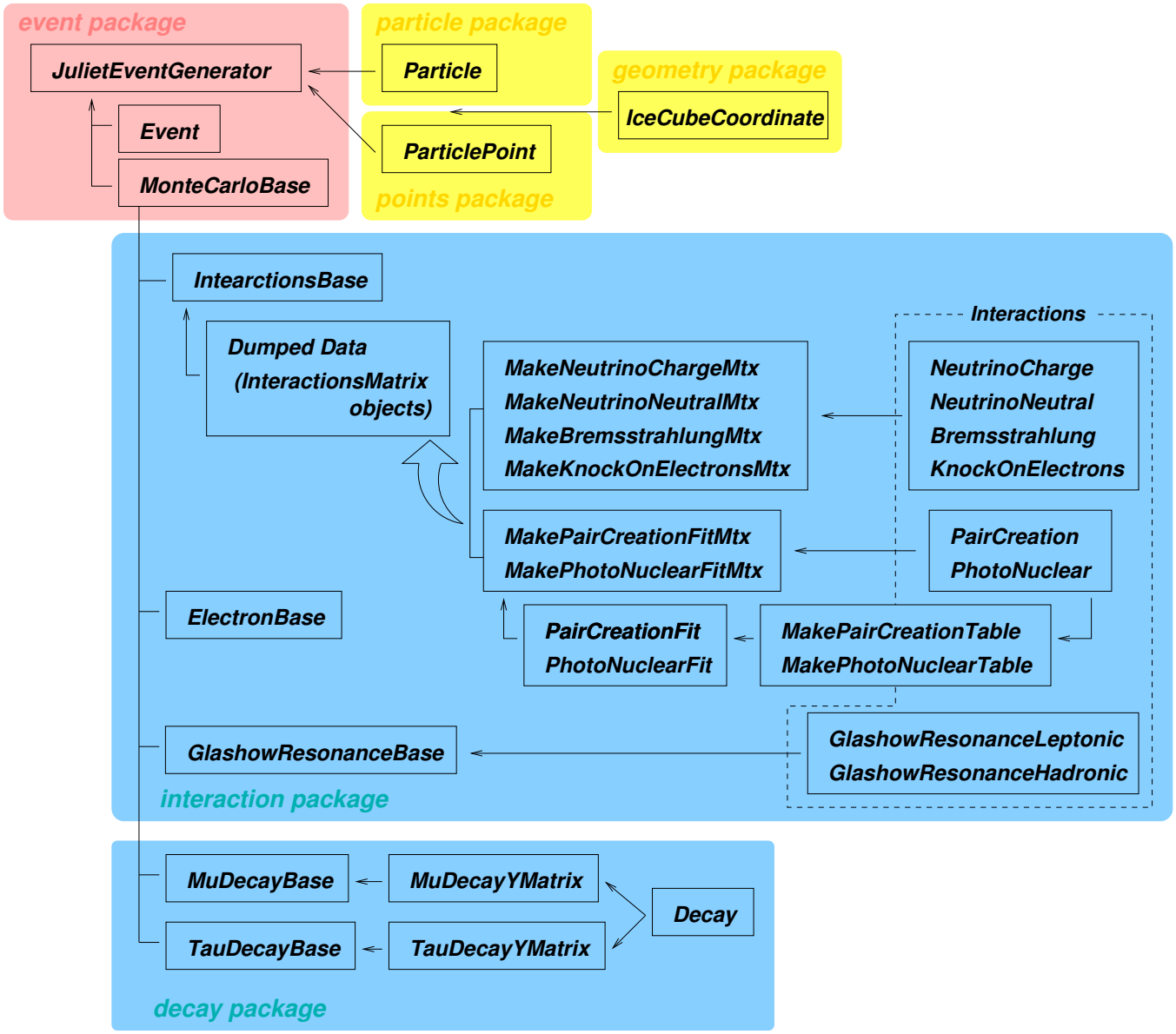

Figure 2.2: Class structure of the JULIeT from aspects of the event package. RunJuliet class has a main method example to initiate JulietEventGenerator for tracing neutrinos/leptons in the IceCube detection volume.

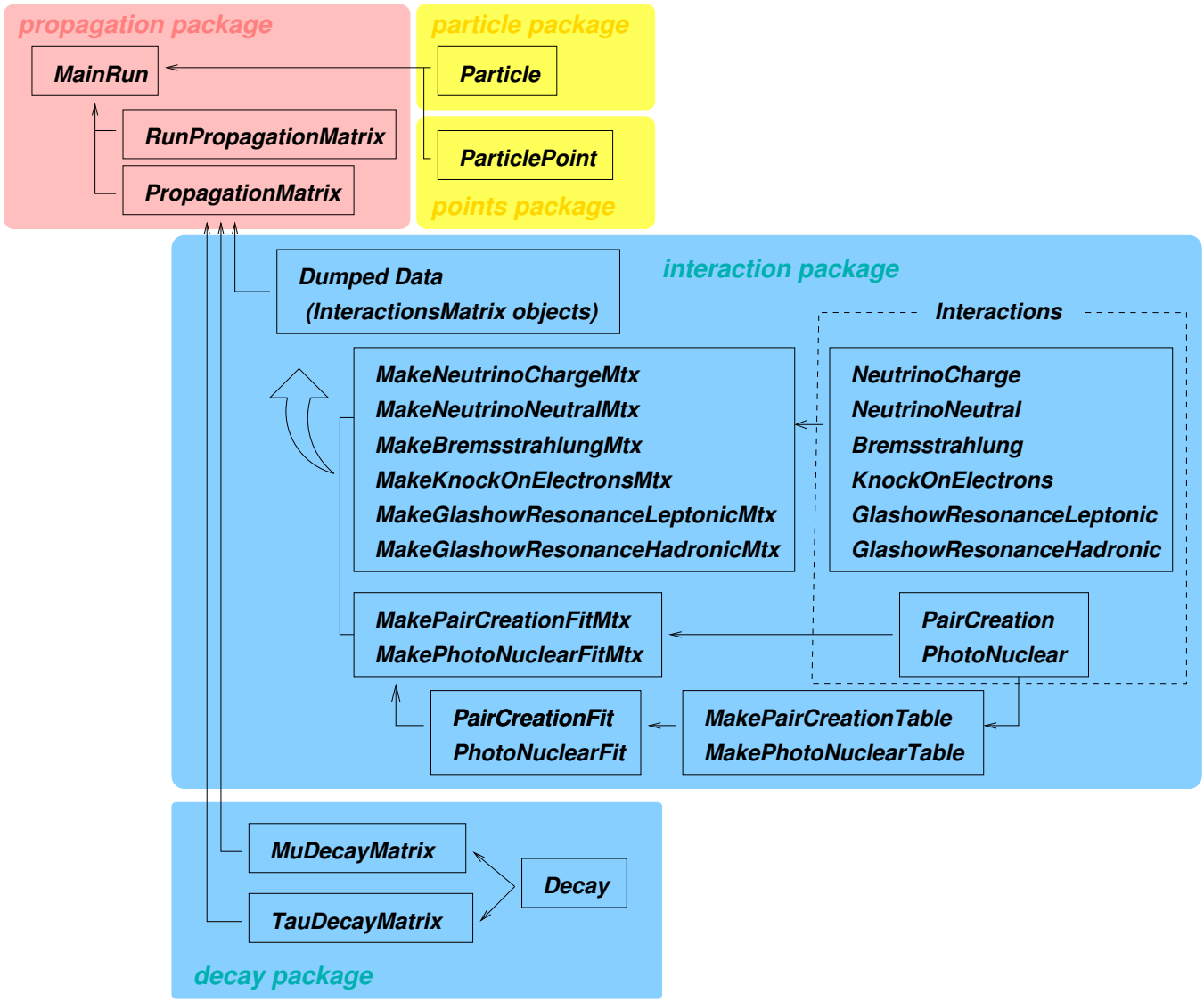

Figure 2.3: Class structure of the JULIeT from aspects of the Propagation package. MainRun class has a main program for tracing neutrinos/leptons in the earth.

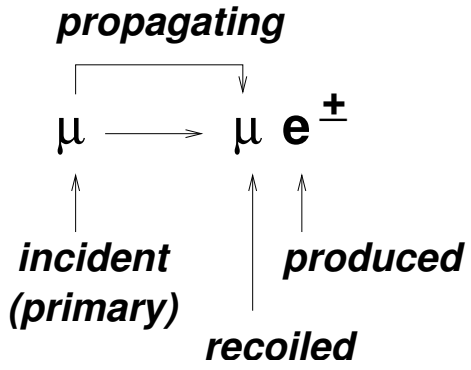

Figure 2.4: Naming convention of particles. This figure shows an example, in case of the pair creation from muon producing electron and positron.

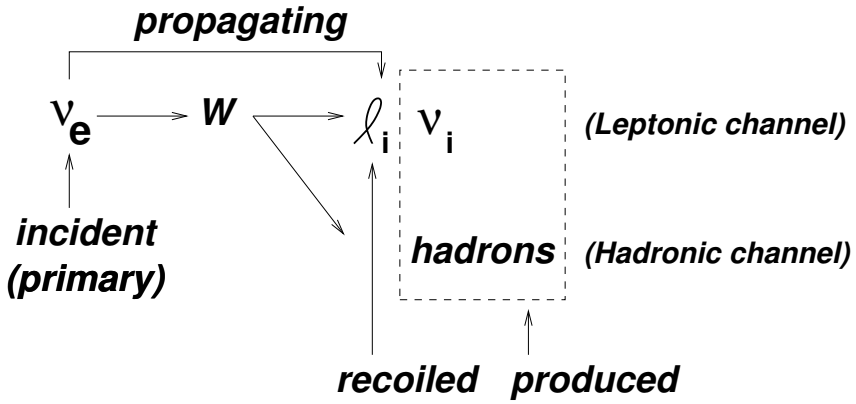

Figure 2.5: Naming convention of particles for the Glashow resonance occurring W boson decay with either leptonic or hadronic channel. The leptonic channel produces a neutrino and charged lepton with flavor of i, and the hadronic channel produces hadrons.

2.3 Convention and Notation

- Primary particles : ν_e, ν_μ, ν_τ, μ, τ
- Energy region [1] : primary particles from 10^5[GeV] to 10^{12}[GeV]
  produced particles from 10^2[GeV] to 10^{12}[GeV]
- Energy bins : 0.01 decade in log of [GeV] in the classes to make binning [2].
- Notation : index of the log energy bins of the primary particles $\cdots$ iLogE, iLogY [3]
  index of the log energy bins of the produced particles $\cdots$ jLogE, jLogY
- Accounting interaction/decay channels : See Table2.1.
- Naming of particles : Fig.2.4 shows the convention of the name of particles. The methods and parameters for every interaction except the Glashow resonance are named following the convention. The Glashow resonance introduces its own conversion as shown in fig.2.5. The hadronic channel of the interaction has no recoiled particle.

[1] They are the default values defined in Particle class (See section5.1).
[2] This is the default value defined in Particle class
[3] For example, differential cross section when a primary μ with iLogE produces e with jLogE via Bremsstrahlung is transferSigmaMatrix[iLogE][jLogE] .

Table 2.1: Allowed interaction/decay channels. Rows are primary particles, and columns are produced particles.

	ν_e	$\nu_m u$	$\nu_t au$	e/γ	μ	τ	hadron
ν_e	NC[a]/G[h]	G	G	CC[b]/G	G	G	NC/CC/G
ν_μ		NC			CC		NC/CC
ν_τ			NC			CC	NC/CC
μ	D[c]	CC/D		PC[d]/B[e]/K[f]/D	PC	PC	D/PN[g]/CC
τ	D	D	CC/D	PC/B/K/D	PC/D	P	D/PN/CC

[a]Neutral Current interaction [b]Charged Current interaction
[c]Decay [d]Pair Creation
[e]Bremsstrahlung [f]Knock-on Electron
[g]Photo-nuclear interaction [h]Glashow Resonance

- Source code naming : Depends on the package, but generally *Demo.java is a Demo program and Draw*.java is to write plots using "xfig-grafig"[4]

[4]xfig is downloadable from **http://www.xfig.org/**. The grafig package is available upon request to us. You can have a look at these codes to figure out how the relevant class works.

Chapter 3

How To Run The Simulation

This chapter describes how to run the JULIeT by using the event package. As mentioned in the Chapter 11, JulietEventGenerator.java in the event package runs the particles and calculates their energy profile with the Monte-Carlo method. The method using the propagation package, which has advantages in calculation of particles propagating over long distances, are described in Chapter 8.

JulietEventGenerator.java runs particles. Some examples using the interactive mode are provided in the following.

Below is an interactions list considered in this simulation. Users can add/remove the individual interaction/decay to/from this list.

- Charged current interaction
- Neutral current interaction
- Bremsstrahlung
- Knock-on Electron
- Pair Creation to e/mu/tau pair
- Photo-nuclear Interaction
- Glashow resonance
- Decay

You can switch on and off these interactions in an interactive manner when you instance JulietEventGenerator because its constructor takes care of the interaction selection.
Allowed regions: Energy region of primary particles from 10^5[GeV] to 10^{12}[GeV]
Energy region of produced particles from 10^1[GeV] to 10^{12}[GeV]

3.1 Download the class and the propagation matrix data files

The JULIeT class files can be generated by compiling the relevant Java source codes. However it is recommended to download the ready-to-use Jar ball from the JULIeT website http://www.ppl.phys.chiba-u.jp/JULIeT/ so that the precalculated interaction matrix data objects (See Sec. 6.2) are available without generating them with yourself by running the make classes.

Unpack the jar ball at your java directory. Let's call this directory "java_lib". Your JULIeT source files should be located under the directory java_lib/sources. Unpacking the jar ball generates the directory structure under java_lib/classes. The structure should be symmetric with those under java_lib/sources, which follows the Java standard style.

The precalculated propagation matrix data (Sec. 8.1) would be also suggested to download from the website. See Sec. 8.5 for the details.

3.2 Photo-Nuclear Interaction

The JULIeT in current version uses two types of cross section of Photo-Nuclear interaction. One of them is calculated by ALLM parameterization [2, 3]. Another is calculated by Bezrukov-Bugaev(BB) parameterization with hard part [4]. When you run the JULIeT, you should

- cd /java_lib/classes/iceCube/uhe/classes(and souces)/, and copy(overwrite) one of the class(and source) files of
  ALLM/PhotoNuclear.class or BB/PhotoNuclear.class
- cd /java_lib/classes/iceCube/uhe/interactions/ice(and rock)/, copy(overwrite) one of the pre-calculated InteractionsMatrix, ALLM/muPhotoNuclearMtx(and tauPhotoNuclearMtx) or
  BB/muPhotoNuclearMtx(and tauPhotoNuclearMtx)
- in /java_lib/data/, remake the symbolic link by "`ln -s propMtx/ALLM(or BB)/ neutrino_earth`" if you use application programs in Propagation package.

3.3 Structure of JulietEventGenerator

The JulietEventGenerator.java runs particles (defined by the Particle package described in Chapter 5) in ice or rock (defined by the Points package). Calculations on particle interactions are made by InteractrionsMatrix which is pre-calculated and stored in disk.

The flow of executing particle run in the JulietEventGenerator.java is as follows.

1. Select the interaction channels and particles involved during the particle propagation. This is done inside the constructor calling the method configureJULIeT().
2. Determine the particle track geometry. The geometry is defined by IceCubeCoordinate.java in the JULIeT geometry package.
3. Configure the geometry by calling the method configurePropagationGeometry(). It converts the geometry to those defined by EarthCenterCoordinate.java where the center of Earth is its origin. We can consider the earth curvature and the rock density profile by doing so. The location of where the primary particle enters into earth is also calculated.
4. run the particle by calling runSingleEvent(). The interaction vertex locations and energies of the secondary produced particle initiating electromagnetic/hadronic cascades are stored in List of "locationIce3List" and "particleList", respectively. Any application program you write would be able to take these results out by using list iterator of these lists which the methods getLocationIce3Iterator() and getParticleIterator() provides. The JulietEventGenerator.java itself has method getListedEvents() to acquire the results stored in the lists.

Example of run An example of the main method to use JulietEventGenerator is found in RunJuliet.java in the same Event package.

cd ~/java_lib/classes/

```
% java -Xms128m -Xmx256m iceCube.uhe.event.RunJuliet

Particle Flavor -> 2              e:0 mu:1 tau:2
Particle Doublet -> 1             neutrino:0 charged lepton:1
Particle Energy [GeV] -> 1.0e10
Medium in Propagation ice(0) rock(1) -> 0
Random Generator has been generated
Charged Current Interactions yes(1)/no(0)?->0
```

```
Neutral Current Interactions yes(1)/no(0)?->0
Muon Bremsstrahlung yes(1)/no(0)?->1
Tau Bremsstrahlung yes(1)/no(0)?->1
Muon Knock-on Electrons yes(1)/no(0)?->1
Tau Knock-on Electrons yes(1)/no(0)?->1
Muon e+e- Pair Creation yes(1)/no(0)?->1
Tau e+e- Pair Creation yes(1)/no(0)?->1
Muon mu+mu- Pair Creation yes(1)/no(0)?->1
Tau mu+mu- Pair Creation yes(1)/no(0)?->1
Muon tau+tau- Pair Creation yes(1)/no(0)?->1
Tau tau+tau- Pair Creation yes(1)/no(0)?->1
Muon Photo-nuclear interactions yes(1)/no(0)?->1
Tau Photo-nuclear interactions yes(1)/no(0)?->1
Glashow Resonance yes(1)/no(0)?->1
Mu decay yes(1)/no(0)?->1
Tau Decay yes(1)/no(0)?->1
...
x [cm] in the ice3 coordinate->0.0
y [cm] in the ice3 coordinate->0.0
z [cm] in the ice3 coordinate->0.0
nadir angle [deg] in the ice3 coordinate-> 85.0
Azimuth angle [deg] in the ice3 coordinate-> 0.0
...
Name of File -> temp.data
Number of Event -> 10
Type of Event (0:single energy (full data) 1:multi energy (Matrix ) ) -> 0
```

Then the results is stored in "temp.data" by F2K format.

Have a look at the source code of JulietEventGenerator.java for understanding what it really does. It is fairly well written with comments. It is also explained in Section 11.3. The particle class to define particles and store their energies, the ParticlePoint class to handle the particle location along its track and acquire the propagation medium information, the J3Vector and J3Line classes for geometry, the IceCubeCoordinate and EarthCenterCoordinate to define the coordinate system, are main players in the generator, all of which is described in the following chapters.

Chapter 4

Package : iceCube.uhe.points

This package contains the classes related to 'definition of point' of particle location. It also takes care of the medium information where your particle is traveling.
The contained class is:

- ParticlePoint

4.1 ParticlePoint.java

This class contains the parameters and methods concerining the particle location and the propagation medium. Calculations on propagation of UHE particles in the rock/ice require the point vector that is provided by this class. It also provides the parameters of the medium such as the density which are used for the calculations of cross sections in the Interaction class.

The parameters provided in this class are:

`public final static double NA = 6.022e23` Avogadro's Number
`private int MaterialNumber` ice→0, rock→1[1]
`private final static double[] SurfaceDensity` Density of ice, rock [g/cm^3]
`private final static double[] IonizationPotential` Ionization potential [eV]
`public final static int[] NumberOfSpecies` Number of species of atom: ice→2(O and H), rock→1
`private final static double[][] AtomicNumber` Atomic number of O,H and 'rock'
`private final static double[][] Charge` Charge of atoms(O, H, and 'rock')
`private final static double[][] RadiationConstant` Radiation constant of atoms
`private final static int[][] NumberOfAtoms` Number of O, H, and 'rock'
`public final static double REarth` Radius of the earth [cm]
`private double IceRockBoundary` Radius of boundary between ice and rock [cm]
`private double R` Distance from the center of the earth [cm]
`private double alpha` Nadir angle [rad]
`private double theta` Zenith angle [rad]
`private double lAxis` Length of the trajectory [cm]
`private double AxisLength` Distance between incident point at the earth surface and emerging point at the earth surface.
`private double Xslant` Slant depth [g/cm^2]

Those parameters above are defined at the point where a particle emarges from underground. See Fig.4.1.

[1] Actually, the rocks are made of several atoms but we considered they are made of only one atom,'rock'.

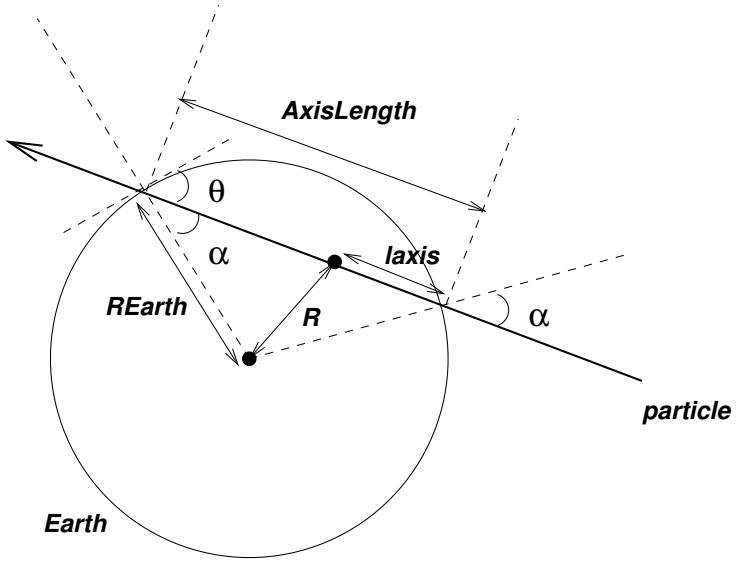

Figure 4.1: Definition of geometical parameters. The α is nadia angle, θ is zenith angle.

Constructor : `public ParticlePoint(double lAxis, double alpha, int MaterialNumber)`
Initialize the starting point and set the geometical parameters.

`public static boolean isValidNadir(double alpha)`
Chack if given nadia angle, `alpha`, is valid, i.e. $0 \leq 2\alpha \leq 2\pi$

`public double getRadiusFromEarthCenter(double lAxis)`
Calculate the radius from the earth center to the given particle location.

`public void setParticleLocation(double lAxis)`
`public double getParticleLocation()`
Set/Get the particle location along the trajectory, i.e. `lAxis`. If a propagating particle passed through and went out of the earth, warning messege will appeared.

`public double getAxisLength()`
Get the `AxisLength`.

`public void setSlantDepth(double Xslant)`
`public double getSlantDepth()`
Set/Get the `Xslant`.

`public double getMediumDensity()`
Get the density of the propagation medium. If the medium is rock, it returns the density depending on the radius from the earth center. While if the medium is ice, the density is considered as location independent.

`public void setIceRockBoundaryRadius(double r)`
`public double getIceRockBoundaryRadius()`
Set/Get the `IceRockBoundary`.

`public double getAtomicNumber(int iThSpecies)`
Get `AtomicNumber` corresponding to given `MaterialNumber` and species index of atom, i.e. `iThSpecies`.

`public double getCharge(int iThSpecies)`
Get `Charge` corresponding to given `MaterialNumber` and species index of atom, i.e. `iThSpecies`.

`public double getRadiation(int iThSpecies)`
Get `RadiationConstant` corresponding to given `MaterialNumber` and species index of atom, i.e. `iThSpecies`.

`public double getNumberOfAtoms(int iThSpecies)`
Get `NumberOfAtoms` corresponding to given `MaterialNumber` and species index of atom, i.e. `iThSpecies`.

`public double getIonizationE(int iThSpecies)`
Get `IonizationPotential` corresponding to given `MaterialNumber`.

`public void setMaterialNumber(int MaterialNumber), public int getMaterialNumber()`
Set/Get `MaterialNumber`. Ice→0, rock→1.

Chapter 5

Package : iceCube.uhe.particles

This package contains the classes to define particles and their property.
The contained classes are:

- Particle
- ParticleArray
- ParticleGroup
- ParticleInputStream
- ParticleOutputStream
- I3Particle
- I3ParticleInputStream
- I3ParticleOutputStream

5.1 Particle.java

This class defines particles by defining their mass, names, flavors, doublets, and lifetime. It also provides their energy. Table5.1 shows the difinition of particles.

The parameters provided in this class are:

`private double energy` Particle energy [GeV]
`private double logEnergy` Log(particle energy [GeV])
`private double mass` Particle mass [GeV]
`private double lifetime` Particle life time [sec]
`private int flavor` Particle flavor
`private int doublet` Particle doublet
`public final static int NumberOfFlavor` Number of valiations of flavor

Table 5.1: Difinition of particles.
The particles defined by their flavors and doublets.

		Flavor			
		0	1	2	3
Doublet	0	ν_e	ν_μ	ν_τ	hadron(π^0)
	1	e	μ	τ	hadron(π^+)

`public final static int NumberOfDoublet` Number of valiations of doublet
`public final static double[][] particleMasses`
Particle mass corresponding to their flavor and doublet
`public final static double[][] particleLifeTimes`
Particle life time corresponding to their flavor and doublet

Variables to handle the energy distribution of the particles, dN/dlogE, are:
(dN/dlogE form a matrix used in calculation of the transport equation for particles' propgation in rock/ice.)
`private static int DimensionOflogEnergyMatrix` Dimension of matrix which elements are dN/dlogE
`private static double logEnergyMinimum` Minimum value of log-energy
`private static double deltaLogE` Bin width of dN/dlogE matrix
`private double[] logEnergyMatrix` Array which indexes imply log-energy
The energy E and the intenger index 'iLogE' is related as log E = logEnergyMinumum + deltaLogE $\times$ (double)iLogE.

Constructor : `public Particle(int initialFlavor, int initialDoublet)`
Initialize flavor and doublet. Default initial energy(1EeV = 10^9[GeV]) is set.

Constructor : `public Particle(int initialFlavor, int initialDoublet, double initialEnergy)`
Check the given flavor, doublet and energy. Set the particle mass and lifetime corresponding to given flavor and doublet.

`public static boolean isValidFlavor(int initialFlavor)`
Check the given flavor is valid. The `initialFlavor` has to be 0$\sim$3.

`public static boolean isValidDoublet(int initialDoublet)`
Check the given doublet is valid. The `initialDoublet` has to be 0 or 1.

`public static boolean isValidEnergy(double initialEnergy)`
Check the given energy, `initialEnergy`, is valid, i.e. particle energy must be greater than its own mass.

`public int getFlavor()`
Get the flavor of the particle.

`public int getDoublet()`
Get the doublet of the particle.

`public doublet getMass()`
Get the mass of the particle.

`public double getLifeTime()`
Get the life time of the particle.

`public double getEnergy()`
Get the energy of the particle.

`public double getLogEnergy()`
Get the log-energy of the particle.

`public void putEnergy(double newEnergy)`
Put energy to the particle only if `newEnergy` is valid.

`public void putLogEnergy(double newLogEnergy)`
Put log-energy to the particle only if `newLogEnergy` is valid.

`public static String particleName(int flavor, int doublet)`
Get the particle name.

`public static int getDimensionOfLogEnergyMatrix()`
`public void putDimensionOfLogEnergyMatrix(int dimension)`
Get/Put `DimensionOflogEnergyMatrix`.

`public static double getLogEnergyMinimum()`
`public void putLogEnergyMinimum(double newLogEnergy)`
Get/Put `logEnergyMinimum`.

`public static double getDeltaLogEnergy()`
`public void putDeltaLogEnergy(double newDeltaLogE)`
Get/Put `deltaLogE`.

`public void generateLogEnergyMatrix()`
Generate array object, `logEnergyMatrix[]`, which length is given with `DimensionOflogEnergyMatrix`.

`public double getLogEnergyMatrix(int ilogE)`
Get the `ilogE`th element of `logEnergyMatrix[]`.

`public void putLogEnergyMatrix(double logE, double matrixElement)`
`public void putLogEnergyMatrix(double logE, double matrixElement)`
Put the `matrixElement` to the `ilogE`th element of `logEnergyMatrix[]`.

5.2 I3Particle.java

This class overrides Particle class with additional methods to store the particle trajectory in both earth-center coordinate (Section 9.5) and the IceCube local coordinate (Section 10.1), the reconstruected energy, and the IceCube detector outputs like number of hit OMs, and NPEs. It also stores the flux weights, the log-differential flux of neutrinos and/or atmospheric muons to responsible for this particle at the IceCube depth. The weighting operation is performed with the class in the analysis package.

I3Particle class has been introduced as an IceCube event data class. The classes in the analysis package reads out this class to run various analysis.

The I/O using the Java's serialization capability are provided I3ParticleInputStream and I3ParticleOutputStream classes. A demo program is I3ParticleStreamDemo.java.

Chapter 6

Package : iceCube.uhe.interactions

This package contains the classes to define interactions and make dumped matrices which elements consist of differential cross sections as a function of log-energy of primary and produced particles.
The contained classes[1] are:

- Interactions
- InteractionsMatrix
- GlashowResonanceHadronicMatrix
- InteractionsMatrixInput
- InteractionsMatrixOutput
- Bremsstrahlung
- MakeBremsstrahlungMtx
- PairCreation
- PairCreationFit
- MakePairCreationTable
- MakePairCreationFitMtx
- PhotoNuclear
- NeurtinoCharge
- GlashowResonanceLeptonic
- GlashowResonanceHadronic
- InteractionsBase
- ElectronBase
- GlashowResonanceBase
- CELbeta

[1] Some of the contained classes are omitted as they are similar to each other(See Fig.2.2).

6.1 Interactions.java

This is an abstract class to provide several method concerning particle interactions. The Pair creation, Bremsstrahlung, Photo nuclear interactions, and the Weak interactions involving UHE leptons and neutrinos propagating underground rock and ice are calculated in its daughter classes.

The parameters provided in this class are:

`static final double Alpha` Fine Structure Constant
`static final double Re` Classical electron radius [cm]
`static final double LambdaE` Compton wavelength [cm]
`static final double E` Base of natural logarithm
`static final double roundOfError` Round off error
`int producedFlavor` Flavor of the produced particle
`double energy` The incident particle energy [GeV]
`double producedMass` Mass of produced particle [GeV]
`double mass` Mass of incident particle [GeV]
`double energyCut` Energy to define the integral range of y=$dE_{produced}/dE_{incident}$
`double[] parameters` Parameter for interface `getFunction()`
`double a` Parameter for interface `getFunction()`
`Particle p` Particle object involved with this interaction
`ParticlePoint s` ParticlePoint object involved with this interaction

Constructor : `public Interactions(Particle p, ParticlePoint s, int flavor)`
Register the Particle and ParticlePoint objects which are necessary because the cross section depends on the particle property and the propagation medium like Z and A(atomic number). It also check if a given particle can be involved in this interaction. The given `flavor` should be the flavor of the produced particle.

`public void setIncidentParticleEnergy(double energy)`
`public void setIncidentParticleEnergy(int iLogE)`
Set the incident particle energy [GeV]. The default value has been set in the constructor. In order to set a different value of `energy/iLogE`, it is need to call this method.

`public double getIncidentParticleEnergy()`
Get the energy of the incident particle.

`abstract double getDSigmaDy(double y)`
This is an abstract method which is implimented in each daughter class, e.g. PairCreation. Get differential cross section, $d\sigma/dy$, where y is inelasticity parameter, $y = 1 - E_{recoil}/E_{incident}$.

`public double getDSigmaDz(double z)`
Get differential cross section, $d\sigma/dz$. $z = E_{recoil}/E_{incident}$: inelasticity parameter.

`abstract boolean isValidInelasticity(double y)`
This is an abstract method which is implimented in each daughter class, e.g. PairCreation. It checks the range of the given inelasticity, `y`.

`abstract double getYmin()`
`abstract double getYmax()`
Get the minimum/maximum of the inelasticity, `y`.

`public void setEnergyCut(double cutEnergy)`
`public double getEnergyCut()`
Set/Get the parameter of energy-cut in integration to obtain the total cross section.

`abstract boolean isValidParticle(Particle p)`
This is an abstract method which is implimented in each daughter class, e.g. PairCreation. It checks the particle kind involved with each interaction.

`public double getSigma()`
Get total cross section [cm^2] with integration of differential cross section.

`public double integralDSigmaDy(double lowerY, double upperY)`
Integrate $d\sigma/dy$ over a given range to obtain a partial cross section.

`public double integralDSigmaDz(double lowerZ, double upperZ)`
Integrate $d\sigma/dz$ over a given range to obtain a partial cross section.

`public double getYDSigmaDy(double lowerY, double upperY)`
Integrate $y \times d\sigma/dy$ over a given range to obtain the inelasticity distribution. The energy transfer probability would also require this value.

`public double getYDSigmaDZ(double lowerY, double upperY)`
Integrate $z \times d\sigma/dz$ over a given range to obtain the inelasticity distribution. The energy transfer probability would also require this value.

`public double getFunction(int functionIndex, double[] parameters, double x)`
Method for interface `Function`[2] to the utility methods such as the Romberg Integration code that is desinged for a genereal function in form of $f(x)$. The `functionIndex` defines $f(x)$:
`functionIndex` 1 $\Rightarrow f(x) = d\sigma/dy$
`functionIndex` 2 $\Rightarrow f(x) = d\sigma/dz$ $\quad z = y\text{-}1$
`functionIndex` 3 $\Rightarrow f(x) = y \times d\sigma/dy$
`functionIndex` 4 $\Rightarrow f(x) = z \times d\sigma/dz$
By calling this interface, the methods like `getSigma()` or `getYDSigmaDy()` in this class can make integrals with the Numerical Recipes package to derive the cross sections and so on.

`abstract String interactionName()`
Get the name of interaction.

`public void showIntegralErrorMessage(double lowerY, double upperY)`
Error message utility.

6.2 InteractionsMatrix.java

6.2.1 What is the InteractionsMatrix object?

This class makes matrices with elements of differential cross section for a given interaction. The elements of matrix are calculated by the methods supplied by the Interaction class (See section6.1). The generated matrices by this class are directly used by the propagation package (Section 8.1), or makes the Interaction-Base matrices (Section 6.16) for sampling the recoiling energy/determination of interaction points with Monte-Carlo method in the event package (Chapter 11).

The parameters provided in this class are:
`Interactions interactions` Interactions object
`double[][] transferMtx` Matrix whose elements are differential cross section, i.e. $d\sigma/dy$.
`double[][] transferAMtx` Matrix whose elements are differential cross section, i.e. $d\sigma/dz$.
`double[] sigmaMtx` Matrix whose elements are total cross section [cm^2]

[2] Function is an Interface to get value of a given function in a given class. It is defined in `~ javalib/sources/numRecipes`, the directoy of the Numerical Recipes package, as

```
public interface Function {
      double getFunction(int functionIndex, double[] parameters, double x);
}
```

`double[] inelasticityMtx` Matrix whose elements are inelasticity
`int dimension` Dimension of the matrix defined in Particle class.
`double delta` Bin width of the matrix defined in Particle class.

Constructor : `public InteractionsMatrix(Interactions interactions)`
Initialize Interactions object and generate matrices.

`public void setTransferMatrix(int iLogE, int jLogE)`
Calculate a element of the `transferMtx[][]`. The element is $d\sigma/dy$ which is the differential cross section for a given `iLogE` and `jLogE`, where `iLogE` is the index of log-energy of incident particle and `jLogE` is the index of produced particle. If jLogE>iLogE, the element must be 0.0, because of energy-conservation. The element is integrated from `logY-0.5*delta` to `logY+0.5*delta` where $\log Y = \log E_{recoil} - \log E_{incident}$. Fig.6.1 shows the schematic construction of the matrix. This is more accurate way than simply calculating $d\sigma/d\log E$. Also calculate a element of the `transferAMtx[][]`. The element indicates $d\sigma/dz$ of the recoiled lepton which energy is $(1-y)\times E_{incident}$.

`public double getTransferMatrix(int iLogE, int jLogE)`
Get the element of the `transferMtx[][]` calculated by `setTransferMatrix()`.

`public double getLeptonTransferMatrix(int iLogE, int jLogE)`
Get the element of the `transferAMtx[][]` calculated by `setTransferMatrix()`.

`public void setSigmaMatrix(int iLogE)`
Calculate the total cross section matrix(`sigmaMtx[]`) and $y\times d\sigma/dy$ (`inelasticityMtx[]` , corresponding to given index of the incident energy, `iLogE`.

`public double getSigmaMatrix(int iLogE)`
Get the element of the `sigmaMtx[]`.

`public double getInelasticityMatrix(int iLogE)`
Get the element of the `inelasticityMtx[]`.

`public boolean isValidIndex(int iLogE)`
Check if the index of the matrix is valid.

`public int getFlavor()`
Get the flavor of the particle propagating.

`public int getDoublet()`
Get the doublet of the particle propagating.

`public int getProducedFlavor()`
Get the flavor of the produced particle.

6.2.2 Make the InteractionsMatrix objects.

The InteractionsMatrix class is `Serializable`, *i.e.* the generated objects with calculated transfer matrices can be stored in the disk. As you see in the following sections, `Make+''Interaction-name''+Mtx.java` has a main method to generate the `InteractionsMatrix` objects, calculate the matrices, and store the objects into the directory.

The suggested directory where you put the generated `InteractionsMatrix` objects are

~/javalib/classes/iceCube/uhe/interactions/rock/

for the cross sections in the rock medium,

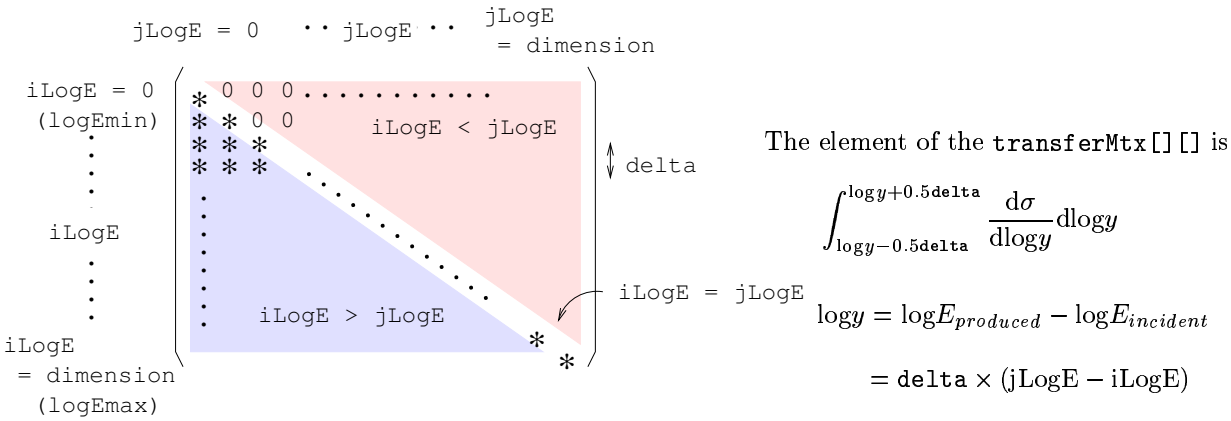

Figure 6.1: Structure of matrix produced by `setTransferMatrix()`.

~/javalib/classes/iceCube/uhe/interactions/ice/

for those in the ice. These are where the propagation package and the event package assumes the `InteractionMatrix` objects are saved. The distribution of the JULIeT package has already included the object files in those directories.

`DrawInteractionsMtx.java` gives you a simple example how to read out the stored `InteractrionsMatrix` object and call its methods.

6.3 GlashowResonanceHadronicMatrix.java

This class has a super class, `InteractionsMatrix` (See section 6.2) and makes the transfer matrix shown in Fig. 6.1 for the hadronic channel of the Glashow resonance interaction. The interaction has a characteristic matrix which is a diagonal matrix with total cross section, because all incident energy is transfered to produced hadrons. The differential cross section for this interaction, therefore, should be $d\sigma/dy = \sigma \times \delta(1-y)$ where $y = E_{incident}/E_{produced}$.

`public void setTransferMatrix(int iLogE, int jLogE)`
Calculate a element of the `transferMtx[][]`. The element is zero if jLogE≠iLogE, or total cross section for a given index of incident energy, `ilogE`, if jLogE = iLogE. Also calculate a element of the `transferAMtx[][]`. The element indicates $d\sigma/dz$ of the recoiled lepton. All the elements are zero because of no incident particle in hadronic channel of the Glashow resonance interaction.

`public void setSigmaMatrix(int iLogE)`
Calculate the total cross section matrix(`sigmaMtx[]`) and y×$d\sigma$/dy (`inelasticityMtx[]` , corresponding to given index of the incident energy, `iLogE`. The inelasticity for this interaction equals to total cross section.

6.4 InteractionsMatrixInput.java

The serialized InteractionsMatrix object is read out from Inputstream. For example,

```
FileInputStream in = new FileInputStream(fileName);
InteractionsMatrix intMtx = InteractionsMatrixInput.inputInteractionsMatrix(in);
```

`public static InteractionsMatrix inputInteractionsMatrix(InputStream in) throws IOException`
Read object serialized by `InteractionsMatrixOutput`. Return the `InteractionsMatrix` object which you read out from the file you specified.

6.5 InteractionsMatrixOutput.java

The InteractionsMatrix object is serialized and written out to OutputStream. For example,

```
FileOutputStream out = new FileOutputStream(fileName);
InteractionsMatrixOutput.outputInteractionsMatrix(intMtx, out);
```

`public static void outputInteractionsMatrix(InteractionsMatrix intMtx, OutputStream out) throws IOException`

Write out the given InteractionsMatrix object (`intMtx` in the above exmaple) to given OutputStream, `out`.

6.6 Bremsstrahlung.java

This class has a super class, `Interactions`. The Bremsstrahlung involving UHE charged leptons propagating undergound rock and ice are calculated in this class.

The parameters provided in this class are:

`private double massRatio` Mass ratio, $m_{produced}/m_{incident}$.

Constructor : `public Bremsstrahlung(Particle p, ParticlePoint s)`

Register the Particle and ParticlePoint objects. They are necessary because the cross section depends on the particle property and the propagating medium like Z and A(atomic number). The Bremsstrahlung produces electron, therefore produced flavor is set to 0, *i.e.* calling `super(p, s, 0)`. The `massRatio` will be used as a scaling factor corresponding to the kind of incident particle.

`public double getDSigmaDy(double y)`

Calculate differential cross section, $d\sigma/dy$ for given `y`. The variable, `chargeFactor` is needed because the cross section depends on the medium like Z and A(atomic number).

`public double getChargeFactor(double y, int ithSpecies)`

Calculate the term which depends on the charge of the propagation medium.

`public boolean isValidInelasticity(double y)`

Checking the range of the given inelasticity,`y` .

`public double getYmin()`

Get the minimum of y allowed for Bremsstrahlung.

`public double getYmax()`

Get the maximum of y allowed for Bremsstrahlung. Assuming that Pure Hydrogen as upper bound.

`public double getYmaxCharge(int ithSpecies)`

Get the maximum of y allowed for Bremsstrahlung, taken account for the charge of medium.

`public boolean isValidParticle(Particle p)`

Checking the particle kind involved with Bremsstrahlung. Only μ and τ but not neutrinos and pions are allowed to be involved with the Bremsstrahlung.

`public String interactionName()`

Get the name of interaction and incident particle.

6.7 MakeBremsstrahlungMtx.java

This class makes the `InteractrionsMatrix` object of Bremsstrahlung, and write out the object to `InteractionsMatrixOutput`. As expained in Section 6.2, this is among the category of Make+"Interaction-name"+Mtx.java and to create the object and store it into the class directory for the event and propagation package.

The parameters provided in main method are:

`String fileName`	Name of the output file you want to make
`int flavor`	Flavor of the incident particle
`int doublet`	Doublet of the incident particle
`int material`	Rock→1, ice→0
`double energy`	Energy of the incident particle [GeV]
`double epsilon`	
`double energyCut`	Energy cut for the integration to obtain the total cross section [GeV]

`public static void main(String[] args) throws IOException`

The main method to make the energy transfer matrix of Bremsstrahlung. Users have to input the name of the output-file and the flavor of the incident particle, e.g. you run `java -classpath /javalib/classes/ iceCube.uhe.interactions.MakeBremsstrahlungMtx out.dat 1` at ~/javalib/classes. Particle and ParticlePoint objects are generated for Bremsstrahlung object. InteractionsMatrix object, `bremssMtx` is generated and total/differential cross sections are calculated by its methods. Finaliy, the `bremssMtx` is written out to output stream.

6.8 PairCreation.java

This class has a super class, `Interactions`. The PairCreation involving UHE leptons and neutrinos propagating undergound rock and ice are calculated in this class.

The parameters provided in this class are:

`private double massRatio` Mass ratio, $m_{produced}/m_{incident}$.
`private double producedScale`
`private double[] para`

Constructor : `public PairCreation(Particle p, ParticlePoint s, int flavor)`

Register the Particle and ParticlePoint objects. They are necessary because the cross section depends on the particle property and the propagation medium like Z and A(atomic number). The PairCreation produces e/μ/τ, so given `flavor` should be 0/1/2. The `massRatio` will be used for a correcting factor corresponding to the kind of incident particle. The `producedScale` is the ratio of electron mass, $e^{\pm}$, to produced charged leptons mass, $e^{\pm}$, $\mu^{\pm}$ and $\tau^{\pm}$.

`public double getDSigmaDyDrho(double y, double rho)`

Calculate differential cross section, dσ/dy for given `y`. The y is an inelasticity parameter, y= $1-E_{recoil}/E_{incident}$. The variable, `chargeFactor` is needed because the cross section depends on the medium like Z and A(atomic number). The assymetry factor is defined by rho = $(E^+ - E^-)/(E^+ + E^-)$.

`public double getDSigmaDy(double y)`

Calculate differential cross section, dσ/dy for given `y`. The variable, `chargeFactor` is needed because the cross section depends on the medium like Z and A(atomic number).

`public double getDSigmaDyPlus(double yPlus)`

Calculate differential cross section, dσ/dy for given `yPlus`=$E_{produced\ e^+}/E_{incident}$

`public double integralDSigmaDyPlus(double lowerY, double upperY)`

Integrate dσ/dyPlus over a given range to obtain a partial cross section.

`public double getYPlusDSigmaDyPlus(double lowerY, double upperY)`

Integrate yPlus×dσ/dyPlus over a given range to obtain the inelasticity distribution. The energy transfer probability would also require this value.

`public double getFunction(int functionIndex, double[] parameters, double x)`
Override Method for interface, `Function`. to the utility methods such as the Romberg Integration code that is desinged for a genereal function in form of f(x). The `functionIndex` defines f(x):

`functionIndex` 1 $\Rightarrow$ f(x) = dσ/dy
`functionIndex` 2 $\Rightarrow$ f(x) = dσ/dz z = y-1
`functionIndex` 3 $\Rightarrow$ f(x) = y$\times$dσ/dy
`functionIndex` 4 $\Rightarrow$ f(x) = z$\times$dσ/dz
`functionIndex` 5 $\Rightarrow$ f(x) = dn/drho Assymetry term
`functionIndex` 6 $\Rightarrow$ f(x) = dσ/dy-drho$times$2/(1+rho)
`functionIndex` 7 $\Rightarrow$ f(x) = dσ/dyPlus
`functionIndex` 8 $\Rightarrow$ f(x) = yPlus$\times$dσ/dyPlus

`public double getAsymmetryTerm(double rho, double y, int ithSpecies)`
Calculate the term on the asymmetry factor of the pair creation. The assymetry factor is defined by rho = ($E^+ - E^-$)/($E^+ + E^-$).

`public double getScreenFactor(int ithSpecies)`
Calculate the factor due to the screening effect on an atomic electron.

`public boolean isValidInelasticity(double y)`
Checking the range of the given inelasticity y.

`public boolean isValidInelasticityPlus(double yPlus)`
Checking the range of the given inelasticity yPlus.

`public double getYmin()`
Get the minimum of y allowed for pair creation.

`public double getYmax()`
Get the maximum of y allowed for pair creation. Assuming that Pure Hydrogen as upper bound.

`public double getYmaxCharge(int ithSpecies)`
Get the maximum of y allowed for pair creation. taken account for the charge of medium.

`public double getYPlusMin()`
Get the minimum of yPlus allowed for pair creation.

`public double getYPlusMax()`
Get the maximum of yPlus allowed for pair creation.

`public boolean isValidParticle(Particle p)`
Checking the particle kind involved with pair creation. Only μ and τ but not neutrinos and pions are allowed to be involved with the pair creation.

`public String interactionName()`
Get the name of interaction and incident particle.

6.9 PairCreationFit.java

This class reads the numerical table generated by MakePairCreationTable.java and calculates the relevant cross sections with numericaly interpolating the table.

6.9.1 InteractionName+Fit class

`PairCreationFit.java` is a class in the categoty of "Interaction Name"+Fit class. The pair creation cross section is very complicated and its calculation from the analytical integration is an extremely time-consuming job. We, therefore, pre-calculate a table of the differential cross section for a few hundreds

of the points in the phase space. "Interaction Name"+Fit class numerically interpolate the pre-calcuated table to obtain the differential and total cross sections.

The pre-calculated tables are generated by `Make+``Interaction Name''+Table.java` and saved in a file. It is highly recommended to put the table in

~/javalib/classes/iceCube/uhe/interactions/rock/

for the cross sections in the rock medium,

~/javalib/classes/iceCube/uhe/interactions/ice/

for those in the ice.

The distribution of the JULIeT jar ball has already contained the pre-calculated tables in the directories above under the name of "*.dat".

The Pair Creation, and the Photonuclear interaction are using this algorithm *i.e.*, PairCreationFit.java and PhotoNuclearFit.java exist in the package.

6.9.2 PairCreationFit class

It reads out the pre-calculated table in

~/javalib/classes/iceCube/uhe/interactions/rock/

or

~/javalib/classes/iceCube/uhe/interactions/ice/

depending on the medium which is defined by the `ParticlePoint` object in the Constructor. The table file names are: muToePairCreation.dat, muTomuPairCreation.dat, tauToePairCreation.dat, tauTomuPair-Creation.dat, tauTotauPairCreation.dat.

The parameters provided in main method are:

`private double[] para`
`private double[] logEArray` Choosed energy array for incident energy
`private double[] logDyArray` Choosed energy array for produced energy
`private double[][] yDsigmaArray` Table of the differential cross sections
`private static final double ln10 = Math.log(10.0)` Common logarithm

`Constructor : public PairCreationFit(Particle p, ParticlePoint s, int Flavor) throws IOException`
Register the Particle and ParticlePoint objects. It also reads dumped data files calculated by MakePairCreationTable.java.

`public double getDSigmaDy(double y)`
Get the differential cross section, $d\sigma/dy$ [cm^2]. y = 1 − E$_r$*ecoile*/E*incident*, inelasticity parameter.

`public boolean isValidInelasticity(double y)`
Check the range of the given inelasticity y that is determined in an individual interaction channel.

`public double getYmin()`
Get the minimum of allowed y for Pair Creation.

`public double getYmax()`
Get the maximum of allowed y for Pair Creation.

`public boolean isValidParticle(Particle p)`
Checking the particle kind involved with this interaction. Only μ and τ but not neutrinos and pions are allowed to be involved with the Pair Creation.

`public String interactionName()`
Get the name of interaction and incident particle.

6.10 MakePairCreationTable.java

This class makes the pre-calculated table of Pair Creation Interactions. The generated data will be used for `PairCreationFit` object.

The parameters provided in main method are:

`String fileName`	Name of the output file you want to make
`int flavor`	Flavor of the incident particle
`int producedFlavor`	Flavor of the produced particles
`int doublet`	Doublet of the incident particle
`int material`	Rock→1, ice→0
`double energy`	Energy of the incident particle [GeV]
`double epsilon`	
`double ln10 = Math.log(10.0)`	Common logarithm
`double[] logEArray`	Choosed energy array for incident energy
`double[] logDyArray`	Choosed energy array for produced energy
`double[][] yDsigmaArray`	Table of the differential cross sections

`public static void main(String[] args) throws IOException`

This is the main method to make the table of differential cross section. Users have to input the name of the output-file, the flavor of the incident and produced particle, e.g. `java -classpath /javalib/classes/ iceCube.uhe.interactions.MakePairCreationTable out.dat 1 0`
Particle and ParticlePoint objects are generated to generate PairCreation object, `pairCMtx`. The differential cross sections are calculated on 120×30 points. Finaliy, the `pairCMtx` is written out to output stream.

6.11 MakePairCreationFitMtx.java

This class makes the energy transfer matrix of `PairCreationFit`, and writes out the object to `InteractionsMatrixOutput`. There is also MakePairCreationMtx class which makes the energy transfer matrix using PairCreation class by calculating all the elements. Because it takes an enormous amount of time to calculate, the event and propagation packages are using the `InteractionMtx` objects of `PairCreationFit`.

The parameters provided in main method are:

`String fileName`	Name of the output file you want to make
`int flavor`	Flavor of the incident particle
`int recFlavor`	Flavor of the produced particle
`int doublet`	Doublet of the incident particle
`int material`	Rock→1, ice→0
`double energy`	Energy of the incident particle [GeV]
`double epsilon`	
`double energyCut`	Energy cut for integration to obtain the total cross section [GeV]
`double ln10 = Math.log(10.0)`	Common logarithm

`public static void main(String[] args) throws IOException`

This is the main method to make the energy transfer matrix of `PairCreationFit`. Users have to input the name of the output-file and the flavor of the incident and produced particle, e.g. `java -classpath /javalib/classes/ iceCube.uhe.interactions.MakePairCreationFitMtx out.dat 1 0`
Particle and ParticlePoint objects are generated to generate PairCreationFit object. InteractionsMatrix object, `pairCMtx` is generated and total/differential cross sections are calculated by its methods. Finaliy, the `pairCMtx` is written out to output stream.

6.12 PhotoNuclear.java

The JULIeT in current version uses two types of cross section of Photo-Nuclear interaction. One of them is calculated by ALLM parameterization [2, 3]. Another is calculated by Bezrukov-Bugaev(BB) parameterization with hard part [4]. When you run the JULIeT, you should choose one of them (See 3.2).

The calculation by ALLM takes long time. In order to save CPU time, `MakePhotoNuclearTable class` calculates differential cross section on some points. Then `MakePhotoNuclearFitMtx class` interpolate the table like `MakePairCreationFitMtx`. While the calculation by BB parameterization does not take long time, so `MakePhotoNuclearMtx class` can calculate a matrix directly like `MakeBremsstrahlungMtx`.

6.13 NeutrinoCharge.java

This class provides the cross sections of the neutrino charged current interactions. Alothough its name does not include the word of 'Fit', it is based on the "Interaction Name"+Fit class: It reads out the pre-caluclated table, "nucch5.dat" in

~/javalib/classes/iceCube/uhe/interactions/.

Because the neutrino cross section is given by that per *nucleon*, the cross section is independent on the madium, ice or rock.

MakeNeutrinoChargeMtx.java generates the `InteractionsMatrix` object for NeutrinoCharge class. The JULIeT distribution contains the `InteraxtionsMatrix` objects as NeutrinoChargeMtx in

~/javalib/classes/iceCube/uhe/interactions/ice/

and

~/javalib/classes/iceCube/uhe/interactions/rock/

The neutral current reaction is taken care of by NeutrinoNeutral.java. The pre-calculated table is "nunch5.dat".

6.14 GlashowResonanceLeptonic.java

This class has a super class, `Interactions`.

The GlashowResonanceLeptonic provides the cross sections of the Glashow resonance reaction with W into the leptonic decay, $\bar{\nu_e} + e^- \rightarrow W \rightarrow \nu_l + l^-$, where l is lepton flavors of e, μ and τ. The approximation that

1) Masses of the produced leptons are negligible

2) Energy of incoming anti electron neutrino is by far higher than the electron mass

has been introduced to calculate the differential cross section. Because the JULIeT uses neutrino cross sections per *nucleon*, the cross section of the Glashow resonance, which is given by per *electron*, should be multiplied by number of electrons per nucleon. The inelasiticity parameter y is here defined as y = 1 - $E^l/E^{\bar{\nu_e}} = E^{\nu_l}/E^{\bar{\nu_e}}$. The Monte Carlo simulation does not propagate the produced ν_l for some reasons. The `JulietEventGenerator` (See section 11.3) is able to handle only propagation of recoiled particles, l^-'s, and moreover, the produced neutrinos with energy of smaller than $\sim 10^7$ GeV have mean free path with longer than $\sim 10^4$ km which is much larger than detector size.

The parameters provided in this class are:

`protected boolean isPerNucleon` Flag:true→cross section is given as per target nucleon.

`protected double chargePerNucleon`
Number of electrons per nucleon calculated in the constructor.

`public static final double G_F = 1.16639e-5` Fermi coupling constant.

`public static final double hbar_c = 1.97327e-14`
Conversion constant in the natural unit [cm GeV].

`public static final double massW = 80.22` Mass of W [GeV].
`public static final double gammaW = 2.12` Decay width of W [GeV].
`public static final double massE` Mass of electron [GeV].

Constructor : `public GlashowResonanceLeptonic(ParticlePoint s, int flavor)`
Register ParticlePoint objects and the produced flavor; e→0. μ →1 and τ →2. They are necessary because the cross section per *electron* depends on the propagating medium like Z and A(atomic number) and this interaction has three channels of produced flavor. The incident particle of the Glashow resonance is anti electron neutrino, therefore the constructor calls `super(new Particle(0,0), s, 0)`.

`public boolean isValidParticle(Particle p)`
Checking the incident particle kind involved with the Glashow resonance. Only electron neutrino is allowed to be involved with the Glashow resonance.

`public void calculateCrossSectionAsPerNucleon()`
Calculate the differential cross section as the one per *nucleon*. For instance in the case of ice, the number of electrons per nuclei is approximately $(2 \times 1 + 1 \times 8)$ corresponding to H_2O is multiplied to the cross section.

`public void calculateCrossSectionAsPerElectron()`
Calculate the differential cross section as the one per *electron*. This is the original cross section for the Glashow resonance.

`public double getDSigmaDy(double y)`
Calculate differential cross section, $d\sigma/dy$ for given `y`.

`public double getSigma(double y)`
Calculate total cross section for given `y`.

`public boolean isValidInelasticity(double y)`
Checking the range of the given inelasticity, `y` .

`public double getYmin()`
Get the minimum of y allowed for the Glashow resonance.

`public double getYmax()`
Get the maximum of y allowed for the Glashow resonance.

`protected double areaFactorByWeakCoupling(double invariant_s)`
Calculate the area given by the weak coupling constant for a given Lorentz invariant energy squared, $A_W = G_F^2 s/pi$, where s is the Lorentzian.

`protected double wResonance(double invariant_s)`
Calculate the dimension less W resonance function, $f_W(s) = M_W^4 \ / \ (s - M_W^2)^2 + M_W^2\Gamma_W^2$, where M_W, Γ_W are Mass of W and decay width of W.

`public String interactionName()`
Get the name of interaction and incident particle.

6.15 GlashowResonanceHadronic.java

This class has a super class, `GlashowResonanceLeptonic` (See section 6.14).

The GlashowResonanceHadronic provides the cross sections of the Glashow resonance reaction with W into the hadronic decay, $\bar{\nu_e} + e^- \rightarrow W \rightarrow$ hadrons. The super class GlashowResonanceLeptonic class is used in most of the calculation. The inelasiticity parameter y is defined as $y = E^{hadrons}/E^{\bar{\nu_e}}$. and is fixed to be always 1 because all the final states are hadrons that generates cascades at once.

The parameters provided in this class are:

`protected final double ratioOfDecayWidth = 6.523` Ratio of decay width of W between hadronic and leptonic channel, *i.e.* Γ_W^{hadron} / Γ_W^{muon} = Γ_W^{hadron} / Γ_W^{tau}.

Constructor : `public GlashowResonanceLeptonic(ParticlePoint s)`

Register ParticlePoint objects. They are necessary because the cross section per *electron* depends on the propagating medium like Z and A(atomic number) and this interaction has three channels of produced flavor. The hadronic channel of the Glashow resonance produces hadron, therefore produced flavor is set to 3, *i.e.* calling `super(s, 3)`.

`public double getDSigmaDy(double y)`

Calculate differential cross section, dσ/dy for given `y`. Because `y` is fixed to be always 1, the differential cross section is defined as dσ/dy = $\sigma \times \delta$(1 - y).

`public double getSigma(double y)`

Calculate total cross section for given `y`.

`public String interactionName()`

Get the name of interaction and incident particle.

6.16 InteractionsBase.java

In order that RunManager and Event classes can handle all of the interactions/decays under the same footing, this class is extended from its super class,MonteCarloBase[3]. The InteractionsBase is one of the "Base" classes for interactions contained in interactions package.

The Base classes essentially provide probability tables to sample the interaction(or decay) length and produced energy with a Monte Carlo method (See Section 11.1).

For an `InteractionsMatrix` object given in the Constructor, this class calculates the probability tables. It also supplies the method to return the path length and produced energy for a given random number. The randum number is sampled by `RandumGenerator` object in the Numerical Recipes package, which is a part of the JULIeT.

The parameters provided in this class are:

`private InteractionsMatrix interactMtx` InteractionsMatrix obejct
`private double logEnergyMinimum` Minimum log energy of incident particles
`private static final double initialLogEnergyProducedMinimum` Initial value of minimum log energy of produced particles
`private double logEnergyProducedMinimum` Minimum log energy of produced particles
`public static final int neutrinoFactor` In order to save CPU time, neutrino cross section is emhanced by this factor
`private int dim` Dimension of dumped matrix
`private int expandedDim` Dimension of corrected matrix
`private double[][] cumulativeTable` Table of cumulative differential cross section.

Constructor : `public InteractionsBase(InteractionsMatrix intMtx)`

Initialize InteractionsMatrix object, `intMtx`. The `cumulativeTable` corresponding to given `intMtx` is calculated.

`public static double getLogEnergyProducedMinimum()`

Get the `initialLogEnergyProducedMinimum`.

`private void setCumulativeTable(InteractionsMatrix interactMtx)`

In order to use Monte Carlo method, the cumulative table of differential cross section is needed.

[3]RunManager, Event and MonteCarloBase are classes of event package.

The dimension of the dumped matrix which is made by MakeBremsstrahlungMtx.java etc. is given by `dim`. In the default, it means 10^6[GeV] to 10^{12}[GeV]. But it is not enough for produced energy because particles may deposit smaller energy when an interaction occured. Therefore ,in this method, differential cross section matrix is expanded to 10^2[GeV] to 10^{12}[GeV], which dimension is given by `expandedDim`. In order to expand the matrix, we introduce an approximation that the recoiling energy distribution follows the energy-scaling law. *i.e.*, the behavior of **fraction** of energy loss is independent of primary energy, which is valid in EHE range. The correction table for energies below 10^6[GeV] is made using dumped data corresponding to initial energy of 10^{12}[GeV], and it is normalized to total cross section with relevant primary energy. Then the cumulative table whose dimension is dim-expandedDim is made. Their elements are normalized by total cross section.

`public double getPathLength(int iLogE, RandomGenerator rand)`
`public double getPathLength(double logEnergy, RandomGenerator rand)`
Get the pathlength of the interaction. The pathlength has a fluctuation around its meanfreepath. A random number[4] generated by given `RandomGenerator` is used for deciding pathlength. This method is called by Event.java, and the rest of the calculation is done in Event.java (See `getPhysicalPathLength()` in Section11.2).

`public double getNeutrinoPathLength(int iLogE, RandomGenerator rand)`
`public double getNeutrinoPathLength(double logEnergy, RandomGenerator rand)`
Get the pathlength of the neutrino interaction. The pathlength has a fluctuation around its mean-freepath.
A random number, `r` generated by given `RandomGenerator` is used for deciding pathlength. This method is called when the primary particle is neutrino. The cross section is enhanced by `neutrinoFactor` to save CPU time. This method is called by Event.java, and the rest of the calculation is done in Event.java (See `getPhysicalPathLength()` in Section11.2).

`public double getProducedEnergy(int iLogE, RandomGenerator rand)`
`public double getProducedEnergy(double logEnergy, RandomGenerator rand)`
Get produced log energy by generated random number sorting cumulativeTable which normalized to 1. In order to make the energy have continuous value, another random number is added.

`public int getPropFlavor()`
Get the flavor of the particle propagateing.

`public int getPropDoublet()`
Get the doublet of the particle propagateing.

`public int getProducedFlavor()`
Get the flavor of the produced particle.

`public String getInteractionName()`
Get the name of the interaction.

`public int getTypeOfInteraction()`
Get type of the interaction. Interaction→0, Decay→1.

6.17 ElectronBase.java

The ElectronBase is one of the "Base" classes contained in the Interactions package.
An electron once generated by ν_e charged current interaction is subject to immediate electromagnetic cascades. The Event class under the current version is not able to simulate cascade features themselves. Instead, it simply records primary energy of the electron (i.e. primary energy of the emg cascades) and its generated location, and put an end to the particle tracing. In order to do so, this class provides a hypothetical "electron-to-electron interaction" where all the primary energy is channeled into "produced"

[4]The `rand.GetRandomDouble()`, `r`, returns a random number from 0 to 1.

electron with pathlength of 0. By calling this class right after an electron is generated by interactions such as ν_e charged current interactions, all the energy is deposited at the same location and the event sees its end.

The parameters provided in this class are:

`Particle p` Particle object which has to be electron

`private static final double initialLogEnergyProducedMinimum` Initial value of minimum log energy of produced particles

`private double logEnergyProducedMinimum` Minimum log energy of produced particles

Constructor : `public ElectronBase(Particle p)`

Require that given Particle object, `p` is electron.

`public static double getLogEnergyProducedMinimum()`

Get the `initialLogEnergyProducedMinimum`.

`private void setCumulativeTable(InteractionsMatrix interactMtx)`

This is the dummy method. It does not have to be called.

`public double getPathLength(int iLogE, RandomGenerator rand)`
`public double getPathLength(double logEnergy, RandomGenerator rand)`

Get the pathlength of the interaction. Electron deposits its energy to "produced" electron without propagating, so pathlength is always 0.0.

`public double getNeutrinoPathLength(int iLogE, RandomGenerator rand)`
`public double getNeutrinoPathLength(double logEnergy, RandomGenerator rand)`

These are dummy methods.

`public double getProductEnergy(int iLogE, RandomGenerator rand)`
`public double getProductEnergy(double logEnergy, RandomGenerator rand)`

Get produced log energy. This method always returns same as incident energy.

`public int getPropFlavor()`

Get the flavor of the particle propagateing.

`public int getPropDoublet()`

Get the doublet of the particle propagateing.

`public int getProducedFlavor()`

Get the flavor of the produced particle

`public String getInteractionName()`

Get the name of the interaction.

`public int getTypeOfInteraction()`

Get type of the interaction. Interaction→0, Decay→1.

6.18 GlashowResonanceBase.java

The GlashowResonanceBase is one of the "Base" classes for the Glashow resonance contained in interactions package.

This class supplies the methods to return the path length and produced energy for a given random number. This class directly uses the `Interactions` object given in the Constructor, while the other `InteractionsBase` classes use an `InteractionsMatrix` objects. Since it is easy to calculate numerically the differential cross section and total cross section, which are the parameters to determine the path length and produced energy. The random number is sampled by `RandumGenerator` object in the Numerical Recipes package, which is a part of the JULIeT.

The parameters provided in this class are:

`private Interactions interactions`
: Interactions object. `GlashowResonanceLeptonic` or `GlashowResonanceHadronic`

`private ParticlePoint point` ParticlePoint object.

`private int materialNumber` Rock→1, ice→0

`private final double antiNuERatio = 0.5`
: Ratio of anti electron neutrino.
It is necessary because only $\bar{\nu_e}$ but not ν_e is allowed to be involved with the Glashow resonance.

`public static final int neutrinoFactor`
: In order to save CPU time, neutrino cross section is enhanced by this factor.

`private final int propFlavor = 0` Flavor of incident particle *i.e.* ν_e is 0.

`private final int propDoublet = 0` Doublet of incident particle *i.e.* ν_e is 0.

`private int producedFlavor` Flavor of produced particle.

Constructor : `public GlashowResonanceBase(int flavor, int mediumID)`
: Generate Interactions object corresponding to a given produced flavor, `flavor`, and material number, `mediumID`. The object is the `GlashowResonanceLeptonic` if the given flavor is e, μ or τ, or the `GlashowResonanceLeptonic` if hadron.

`public double getPathLength(int iLogE, RandomGenerator rand)`
`public double getPathLength(double logEnergy, RandomGenerator rand)`
: Get the path length of the interaction. The path length has a fluctuation around its mean free path. A random number[5] generated by given `RandomGenerator` is used for deciding path length. This method is called by Event.java, and the rest of the calculation is done in Event.java (See `getPhysicalPathLength()` in Section11.2).

`public double getNeutrinoPathLength(int iLogE, RandomGenerator rand)`
`public double getNeutrinoPathLength(double logEnergy, RandomGenerator rand)`
: Get the path length of the neutrino interaction. The path length has a fluctuation around its mean free path.
A random number, `r` generated by given `RandomGenerator` is used for deciding path length. This method is called when the primary particle is neutrino. The cross section is enhanced by `neutrinoFactor` to save CPU time. This method is called by Event.java, and the rest of the calculation is done in Event.java (See `getPhysicalPathLength()` in Section11.2).

`public double getProducedEnergy(int iLogE, RandomGenerator rand)`
`public double getProducedEnergy(double logEnergy, RandomGenerator rand)`
: Get produced log energy by generated random number. The energy is calculated numerically because the differential cross section for the Glashow resonance is a simple function of inelasticity parameter, `y`.

`public int getPropFlavor()`
: Get the flavor of the particle propagating.

`public int getPropDoublet()`
: Get the doublet of the particle propagating.

`public int getProducedFlavor()`
`public int setProducedFlavor()` Get/set the flavor of the produced particle.

`public String getInteractionName()`
: Get the name of the interaction.

`public int getTypeOfInteraction()`
: Get type of the interaction. Interaction→0, Decay→1.

[5] The `rand.GetRandomDouble()`, `r`, returns a random number from 0 to 1.

6.19 CELbeta.java

The energy loss profile during the particle propagation is fully able to be calculated in the propagation package described in Chapter 8. The quick and handy calculation can be done, however, using the Continuous Energy Loss (CEL) approximation:

$$-\frac{dE}{dX} = \alpha + \beta E \tag{6.1}$$

In the very high energy regime ($\gg 10^3 GeV$), the ionization term, α, is negligible. Therefore the inelasticity parameter β given by the relevant differential cross sections determines the profile in this approximation. CELbeta.java provides β as a function of muon energy. It is based on its internal numerical table calculated using InteractionMatrix.java (Sec.6.2). In the moment, it involves muon propagation in ice only.

`public static double getBeta(double logEnergy)`
Returns β [cm^2/g GeV] for log10(muon energy [GeV]).

Chapter 7

Package : iceCube.uhe.decay

This package contains the classes to handle decay processes and make decay matrices which elements are differential decay rate. The `DecayMatrix` objects are similar with the `InteractionsMatrix` in the interaction package. The contained classes are:

- Decay
- MuDecayYMatrix
- MuDecayMatrix
- MuDecayBase
- TauDecayYMatrix
- TauDecayMatrix
- TauDecayBase

7.1 Decay.java

Particle decay such as $\tau \rightarrow \mu \rightarrow$e is handled in this class. The lifetimes of particles are provided by the Particle class(See 5.1).

The parameters provided in this class are:

`public static final double rMuPI` The squared mass ratio $(m_\mu/m_\pi)^2$
`public static final double rPiTau` The squared mass ratio $(m_\pi/m_\tau)^2$
`public static final double rRhoTau` The squared mass ratio $(m_\rho/m_\tau)^2$
`public static final double rA1Tau` The squared mass ratio $(m_{A1}/m_\tau)^2$
`public static final double rXTau` The squared mass ratio $(m_X/m_\tau)^2$
m_X is the 'averaged mass' of produced hadrons
`public static final double BRatioTau2Leptons` Branching ratio of Leptonic tau decay
`public static final double BRatioTau2Pi` Branching ratio of tau decay puroducing pi's
`public static final double BRatioTau2Rho` Branching ratio of tau decay puroducing rho's
`public static final double BRatioTau2A1` Branching ratio of tau decay puroducing a1's
`public static final double BRatioTau2X` Branching ratio of tau decay puroducing others
`Particle p` Particle object involved with the decay
`double energy` Energy of the particle involved with the decay [GeV]
`double tau` Lifetime of the particle [sec]

Constructor : `public Decay(Particle p)`
Register given Particle object, `p`. The particle has to be μ or τ or π.

`public boolean isValidParticle(Particle p)`
Check the particle kind involved with a given decay channel. Only μs, τs and πs but not electrons and neutrinos are allowed to be involved for the obvious reasons.

`public static double getWeakDecayProbToW(double y, double parity)`
Calculate the Weak Decay probability per inelasticity, `y`. The energy of decay products is y$\times$E$_{decay}$. The product originated in the charged current from the decaying particle is taken care of (See Fig.7.1).

`public static double getWeakDecayProbFromW(double y, double parity)`
Calculate the Weak Decay probability per inelasticity, `y`. The energy of decay products is y$\times$E$_{decay}$. The produced neutrino originated in the charged current on the another side is taken care of (See Fig.7.1).

`public static double integralWeakDecayProbToW(double lowerY, double upperY, double parity)`
Integration of the Weak decay probability from `lowerY` to `upperY`. The integration is analyticaly made in this method. Handle the particles of the charged current on the another side.

`public static double integralWeakDecayProbFromW(double lowerY, double upperY, double parity)`
Integration of the Weak decay probability from `lowerY` to `upperY`. The integration is analyticaly made in this method.

`public static double getYmin()`
Get the minimum of inelasticity, `y`.

`public static double getYmax()`
Get the maximum of inelasticity, `y`.

`public static double getTauHadronDecayProbToW(double y, double massRatio)`
Calculate the τ to hadron decay probability per inelasticity, `y`. The energy of decay product (in this case ν_τ) is y$\times$E$_\tau$. (See Fig.7.2)

`public static double getTauHadronDecayProbFromW(double y, double massRatio)`
Calculate the τ to hadron Decay probability per inelasticity, `y`. The energy of decay product (in this case hadron) is y$\times$E$_\tau$. (See Fig.7.2)

`public static double getYmax(double massRatio)`
Get the maximum of inelasticity, `y` for the τ to hadron Decay.

`public static double integralTauHadronDecayProbToW(double lowerY, double upperY, double massRatio)`
Integration of the τ to hadron decay probability from lowerY to upperY. The integration is analyticaly made in this method.

`public static double integralTauHadronDecayProbFromW(double lowerY, double upperY, double massRatio)`
Integration of the τ to hadron decay probability from lowerY to upperY.

7.2 MuDecayYMatrix.java

This class makes the matrix of the energy transfer dN/dlogY for the μ decays. The matrix elements are calculated by the methods supplied by the Decay class. The bin width is defined in the Particle class. Actually each matrix element dN/dLogY, where logY = logE$_{produced}$ $-$ logE$_\mu$, is calculated by the integralation of dN/dY from logY $-$ 0.5$\times$ bin width to logY + 0.5$\times$bin width. The transfer matrix to the ν_μ is acquired by the method `getMuToNuMuDecayMatrix()`, that to the ν_e by `getMuToNuEMatrix()` that to the other charged leptons such as electron by `getMuToEDecayMatrix()`.

This object and the similar object `MuDecayMatrix` (see Section 7.4) play the same roles as the `InteractionsMtx` objects do in the event/propagation package. The biggest difference is, the decey matrix element is calculated whenever the object is generated while `InteractionsMtx` is usually pre-calculated and stored in form of files to be read out by the Event/PropagationMatrix class. This is because it takes much less time to calculate the decay matrices than the interaction matrices.

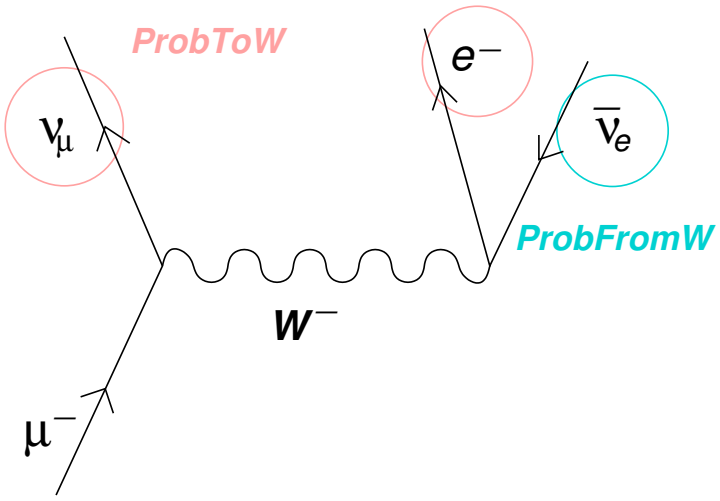

Figure 7.1: Definition of Weak decay probability e.g. μ to e decay. `getWeakDecayProbToW` and `getWeakDecayProbFromW` return the energies as above, respectively.

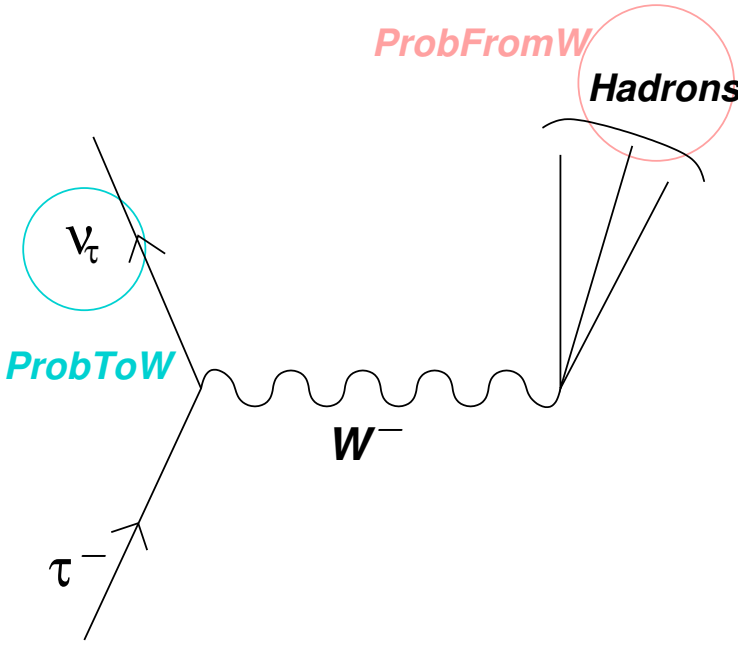

Figure 7.2: Definition of τ to hadron decay probability. `getTauHadronDecayProbToW` and `getTauHadronDecayProbFromW` return the energies as above, respectively.

The parameters provided in this class are:

`double[] nuMuMtx` Array for the energy tansfer probability from μ to ν_μ or electron
`double[] nuMtx` Array for the energy tansfer probability from μ to ν_e
`double[] lifetimeMtx` Array for the lifetime of μ considering the Lorentz duration
`int dimension` Dimension of arrays
`double delta` The bin width of the matrix defined in Particle class (See 5.1)
`Particle p` The Particle class which should be μ

Constructor : `public MuDecayYMatrix(Particle p)`
Generate arrays and requires that `p` is μ.

`public void setMuDecayMatrix(int iLogY)`
Calculate the decay matrix corresponding to given `iLogY` from μ to ν_{mu} or electron.

`public void setLifeTimeMatrix(int iLogE)`
Calculate the life time matrix considering the Lorentz duration.

`public double getMuToNuMuDecayMatrix(int iLogY)`
Get the element of the decay matrix of μ to ν_μ corresponding to given `iLogY`, i.e. `nuMuMtx[iLogY]`.

`public double getMuToNuEDecayMatrix(int iLogY)`
Get the element of the decay matrix of μ to ν_e corresponding to given `iLogY`, i.e. `nuMtx[iLogY]`.

`public double getMuToEDecayMatrix(int iLogY)`
Get the element of the decay matrix of μ to electron corresponding to given `iLogY`, i.e. `nuMuMtx[iLogY]`.
Note that this method returns same value of the `getMuToNuMuDecayMatrix(int iLogY)`.

`public double getLifeTimeMatrix(int iLogE)`
Get the element of the lifetime matrix.

`public boolean isValidIndex(int iLogY)`
Check if given `iLogY` is valid.

7.3 TauDecayYMatrix.java

This class makes the matrix of the energy transfer dN/dlogY for the τ decays. The matrix elements are calculated by the methods supplied by the Decay class. The bin width is defined in the Particle class. Actually each matrix element dN/dlogY, where logY = $\log E_{produced} - \log E_\tau$, is calculated by the integration of dN/dY from logY $-$ 0.5$\times$bin width to logY + 0.5$\times$bin width The transfer matrix to the ν_τ is acquired by the method `getTauToNuTauDecayMatrix()`, that to the other neutrinos such as ν_e by `getTauToNuDecayMatrix()` that to the other charged leptons such as electron by `getTauToChargedLeptonDecayMatrix()` while the decay into hadrons is obtained by `getTauToHadronsDecayMatrix()`.

The parameters provided in this class are:

`double[] nuTauMtx` Array for the energy tansfer probability from τ to ν_τ or electron
`double[] nuMtx` Array for the energy tansfer probability from τ to ν_μ or ν_e
`double[] leptonsMtx` Array for the energy tansfer probability from τ to μ or e
`double[] hadronsMtx` Array for the energy tansfer probability from τ to hadrons
`double[] lifetimeMtx` Array for the lifetime of τ considering the Lorentz duration
`int dimension` Dimension of arrays
`double delta` The bin width of the matrix defined in Particle class (See 5.1)
`Particle p` The Particle class which should be τ

Constructor : `public TauDecayYMatrix(Particle p)`
Generate arrays and requires that `p` is τ.

`public void setTauDecayMatrix(int iLogY)`
Calculate the decay matrix corresponding to given `iLogY` from τ to neutrinos, charged leptons and hadrons.

`public void setLifeTimeMatrix(int iLogE)`
Calculate the life time matrix considering the Lorentz duration.

`public double getTauToNuTauDecayMatrix(int iLogY)`
Get the element of the decay matrix of τ to ν_τ corresponding to given `iLogY`, i.e. `nuTauMtx[iLogY]`.

`public double getTauToNuDecayMatrix(int iLogY)`
Get the element of the decay matrix of τ to ν_μ or ν_e corresponding to given `iLogY`, i.e. `nuMtx[iLogY]`.

`public double getTauToChargedLeptonDecayMatrix(int iLogY)`
Get the element of the decay matrix of τ to charged lepton corresponding to given `iLogY`, i.e. `leptonsMtx[iLogY]`.

`public double getTauToHadronDecayMatrix(int iLogY)`
Get the element of the decay matrix of τ to hadrons corresponding to given `iLogY`, i.e. `hadronsMtx[iLogY]`.

`public double getLifeTimeMatrix(int iLogE)`
Get the element of the lifetime matrix.

`public boolean isValidIndex(int iLogY)`
Check if given `iLogY` is valid.

7.4 MuDecayMatrix.java

This class is the similar with the `MuDecayYMatrix.java` but the matrix element represents the energy differential rate dN/dLogE, instead of dN/dLogY.

7.5 TauDecayMatrix.java

This class is the similar with the `TauDecayYMatrix.java` but the matrix element represents the energy differential rate dN/dLogE, instead of dN/dLogY.

7.6 MuDecayBase.java

In order that RunManager and Event classes can handle all of the interactions/decays in the same foundation, this class is extended from its super class, MonteCarloBase[1]. The MuDecayBase is one of the "Base" classes for mu decay contained in decay package.

Similar with the InteractionsBase class, it calculates the Monte Carlo probability table using the decay matrices calculated by the MuDecayYMatix.java.

The parameters provided in this class are:

`private MuDecayYMatrix muDecayMtx` MuDecayYMatrix object
`private Particle p` Particle object
`private int dim` Dimension of the MuDecayYMatrix

Constructor : `public MuDecayBase(MuDecayYMatrix muDecayMtx)`
Make the lifetime table of cumulative cross section.

[1] RunManager, Event and MonteCarloBase are classes of event package.

`public void setCumulativeTable(MuDecayYMatrix muDecayMtx)`
In order to use Monte Carlo method, the cumulative table of differential cross section is needed. The dimension of the matrices which are made by MudecayYmatrix is `dim`. The elements of cumulative table had been already normalized to 1.

`public double getPathLength(int iLogE, RandomGenerator rand)`
`public double getPathLength(double logEnergy, RandomGenerator rand)`
Get the pathlength of the mu decay. The pathlength has a fluctuation around its meanfreepath. A random number[2] generated by given `RandomGenerator` is used for deciding pathlength. This method is called by Event.java, and the rest of the calculation is done in Event.java (See `getPhysicalPathLength()` in Section11.2).

`public double getNeutrinoPathLength(int iLogE, RandomGenerator rand)`
`public double getNeutrinoPathLength(double logEnergy, RandomGenerator rand)`
These are dummy methods, i.e. these methods do not have to be called. But these methods are need to exist beause this class is daughter class of MonteCarloBase class.

`public double getProducedEnergy(int iLogE, RandomGenerator rand)`
`public double getProducedEnergy(double logEnergy, RandomGenerator rand)`
Get produced log energy by generated random number sorting cumulativeTable which normalized to 1. In order to make the energy have continuous value, another random number is added.

`public int getPropFlavor()`
Get the flavor of the particle propagating.

`public int getPropDoublet()`
Get the doublet of the particle propagating.

`public int getProducedFlavor()`
Get the flavor of the produced particle.

`public String getInteractionName()`
Get the name of interaction.

`public int getTypeOfInteraction()`
Get the type of interaction (Interaction→0, Decay→1)

7.7 TauDecayBase.java

In order that RunManager and Event classes can handle all of the interactions/decays in the same foundation, this class is extended from its super class, MonteCarloBase class. The TauDecayBase is one of the "Base" classes for tau decay contained in decay package.

Similar with the InteractionsBase class, it calculates the probability table using the decay matrices calculated by the TauDecayYMatix.java.

The parameters provided in this class are:
`private TauDecayYMatrix tauDecayMtx` TauDecayYMatrix object
`private Particle p` Particle object
`private int dim` Dimension of the TauDecayYMatrix

Constructor : `public TauDecayBase(TauDecayYMatrix tauDecayMtx)`
Make the lifetime table of cumulative cross section.

[2]The `rand.GetRandomDouble()`, `r`, returns a random number from 0 to 1.

`public void setCumulativeTable(TauDecayYMatrix tauDecayMtx)`

In order to use Monte Carlo method, the cumulative table of differential cross section is needed. The dimension of the matrices which are made by TaudecayYmatrix is `dim`. Tau can decay into e/μ/hadron, but τ to e decay and τ to μ decay have the same decay rate. So two cumulative tables are produced in this method (for lepton and for hadron). Note that "hadron" indicates the *superposition of various hadrons*. So branching ratio (`BRatio`, defined in Decay.java) for τ to hadron decay is the sum of branching ratio for each hadron. The elements of cumulative tables had been already normalized to 1.

`public double getPathLength(int iLogE, RandomGenerator rand)`
`public double getPathLength(double logEnergy, RandomGenerator rand)`

Get the pathlength of the tau decay. The pathlength has a fluctuation around its meanfreepath. A random number[3] generated by given `RandomGenerator` is used for deciding pathlength. This method is called by Event.java, and the rest of the calculation is done in Event.java (See `getPhysicalPathLength()` in Section11.2).

`public double getNeutrinoPathLength(int iLogE, RandomGenerator rand)`
`public double getNeutrinoPathLength(double logEnergy, RandomGenerator rand)`

These are dummy methods, i.e. these methods do not have to be called. But these methods are need to exist beause this class is daughter class of MonteCarloBase class.

`public double getProducedEnergy(int iLogE, RandomGenerator rand)`
`public double getProducedEnergy(double logEnergy, RandomGenerator rand)`

Get produced log energy by generated random number sorting cumulativeTable which normalized to 1. The decay mode is decided by calling `setDecayMode()`. In order to make the energy have continuous value, another random number is added.

`public void setDecayMode(RandomGenerator rand)`

Choose the decay mode. The cumulative table of branching ratio (`cumBRatio[]`) is produced, and one of the decay mode is chozen by random number. Subject to the chozen decay mode, the corresponding cumulative tables produced in the `setCumulativeTable()` is substituted for `cumulativeTable`. Note that τ to hadon decay is treated which produced one hypothetical "hadron" particle, which is superposition of πs, ρs and so on.

`public int getDecayMode()`

Get the decay mode (0:τ to electron, 1:τ to μ, 2:τ to hadron).

`public int getPropFlavor()`

Get the flavor of the particle propagating.

`public int getPropDoublet()`

Get the doublet of the particle propagating.

`public int getProducedFlavor()`

Get the flavor of the produced particle. The produced flavor is corresponding to the decay mode.

`public String getInteractionName()`

Get the name of interaction.

`public int getTypeOfInteraction()`

Get the type of interaction (Interaction$\rightarrow$0, Decay$\rightarrow$1)

[3]The `rand.GetRandomDouble()`, `y`, returns a random number from 0 to 1.

Chapter 8

Package : iceCube.uhe.propagation

This package contains the classes for numerically solving the transport equations [1], which provides a faster and more accurate evaluation of the particle propagation in long distances. The MainRun has the main method to run the calculation.

The contained classes are:

- PropagationMatrix
- RunPropagationMatrix
- MainRun
- PropagationMatrixFactory

8.1 PropagationMatrix.java

Matrix of the energy and particle species transfer by the interactions/decays. The matrix elements are a priori calculated by the methods supplied by the InteractionMatrix in the interactions package (See 6.2) and by the Decay classes in the decay package (See 7.1). Table.8.1 shows the first order interactions/decays channels considered in this class. Switching on/off an individual interactions/decays channels are done by the bits pate meter interactionsSwitch and decaySwitch (See Table.8.2). The methods in this class will be called by RunPropagationMatrix class to trace the particle propagation.

Table 8.1: Channels of infinitesimal propagation
Rows are primary and columns are produced/survival particles.

	ν_e	ν_{mu}	ν_{tau}	e	μ	τ	hadron
ν_e	NC[a]/G[h]/S[i]	G	G	CC[b]/G	G	G	NC/CC/G
ν_μ		NC/S			CC		NC/CC
ν_τ			NC/S			CC	NC/CC
μ	D[c]	CC/D		PC[d]/B[e]/K[f]/D	PC/S	PC	D/PN[g]/CC
τ	D	D	CC/D	PC/B/K/D	PC/D	P/S	D/PN/CC

[a] Neutral Current interaction
[b] Charged Current interaction
[c] Decay
[d] Pair Creation
[e] Bremsstrahlung
[f] Knock-on Electron
[g] Photo-nuclear interaction
[h] Glashow Resonance
[i] Survive

Table 8.2: Interactions/Decays Switch:Switching on/off an individual interactions channels are done by the bits parameter `interactionsSwitch` and `decaySwitch`. 'PairC Heavy' indicates the pair creation producing $\mu^{\pm}$ or $\tau^{\pm}$, and 'GR' the Glashow resonance.

interactionsSwitch								
bit8	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
GR	CC(Lepton$\rightarrow \nu$)	PhotoNuclear	Bremsstrahlung	Knock-on	PairC Heavy	PairC	NC	CC

decaySwitch		
bit7-2	bit1	bit0
Reserv.	TauDecay	MuDecay

The parameters provided in this class are:

`int dimension` Dimension of matrix

`final int dimResonance = 180` Dimension of matrix at the resonance peak energy, logE = 6.8.

`double massNumber` Mass number of the propagation medium

`public double dX` Step size of the propagation [g/cm^2]

`public double dXDecay` Step size of the propagation [g/cm^2], the shortest decay length

`private double[][][] temp`

`private double[] intProbNeutrino,intProbMu,intProbTau,intProbNuE`
Total interaction probability of neutrino (CC and NC),μ,τ, and the Glashow resonance.

`private double[][] nuEToNuE etc.` Matrices for infinitesimal propagation e.g. $\nu_e \rightarrow \nu_e$

`private double[][] FnuEToNuE etc.` Matrices for finite propagation e.g. $\nu_e \rightarrow \nu_e$

`private double[][] SnuEToNuE etc.` Matrices for storing calculated matrices e.g. $\nu_e \rightarrow \nu_e$

`private final static int CHARGED_FLAG=1 etc.` Allows the Charged Current

`private final static int ALL_FLAG = 511` Allows all the interaction and decay channels

`private InteractionsMatrix nuCCMtx etc.` InteractionMatrix generated by MakeNeutrinoChargeMtx

`private String nuCCMtxObjectFile etc.` Name of dumped object for Charged Interaction etc.

`protected double neutrinoFactor = 1.0` Neutrino CC and NC enhancement factor

`private MuDecayMatrix muDecayMtx` MuDecayMatrix object (not dumped)

`private TauDecayMatrix tauDecayMtx` TauDecayMatrix object (not dumped)

`Particle nuE etc.` Particle object e.g. ν_e

`ParticlePoint s` Point of the particle propagating

`int interactionsSwitch` Bit switch for interactions

`int decaySwitch` Bit switch for decays

Constructor : `public PropagationMatrix(Particle nuE, Particle nuMu, Particle nuTau, Particle e, Particle mu, Particle tau, Particle pi, ParticlePoint s, int interactionsSwitch, int decaySwitch double neutrinoFactor)`

Check given particles. Read all the InteractiosMatrix objects and generating the DecayMatrix objects. All the InteractiosMatrix objects and the DecayMatrix objects are initialized. The infinitesimal propagation distance dX [g/cm^2] is also determined, chosen the shortest mean free path.

Constructor : `public PropagationMatrix(Particle nuE, Particle nuMu, Particle nuTau, Particle e, Particle mu, Particle tau, Particle pi, ParticlePoint s, int interactionsSwitch, int decaySwitch)`

Same as above, but setting the `neutrinoFactor` with 1.

`public void init()`

Initialize the propagation matrices to diagonal matrices.

`public void initALL()`

Initialize all the propagation matrices including the store matrices to diagonal matrices.

`public void calculateTransferMatrix()`

Calculate the elementary(infinitesimal) interaction/decay transfer matrix. The mass density of the propagation medium is approximated to be constant and equal to the current particle location for saving CPU time. The infinitesimal propagation over dX is a subject to consider here.
The elements of `intProbNeutrino[]` etc. are calculated first. They indicate the interaction probability, i.e.

$$\texttt{intProbNeutrino[kLogE]} = dX N_A \ (\ \sigma_{CC}(kLogE) + \sigma_{NC}(kLogE) \ ),$$

where σ_{CC} and σ_{NC} are the total cross section of Charged and Neutral Current Interactions. The calculated matrix element are stored in `nuEToNuE[kLogE][jLogE]` etc.
The `nuEToNuE[kLogE][jLogE]` indicates transfer matrix from ν_e with `kLogE` to ν_e with `jLogE`, i.e.

$$\texttt{nuEToNuE[kLogE][jLogE]} = dX N_A \frac{d\sigma_{NC}}{dlogE}(kLogE \rightarrow jLogE) + \frac{d\sigma_{GR}^{lepton}}{dlogE}(kLogE \rightarrow jLogE).$$

The interaction which produces ν_e from ν_e is Neutral Current Interaction and the Glashow resonance of lectonic channel. The similar procedures are taken for the other leptons; like `tauToMu[][]`. The decay contribution are also calculated and added to the matrix element.

`public void propagateDX()`

Propagate the particles over a dX [g/cm^2]. The calculation is performed by multiplication of the elemental transfer matrix calculated by calculateTransferMatrix(). The resultant energy distributions are stored in `FnuEToNuE[iLogE][kLogE]` etc. They are calculated as

$$\begin{aligned}
&\texttt{dNFromNuE} = \\
&\quad \textstyle\sum_{kLogE} \{ \ \texttt{FnuEToNuE[iLogE][kLogE]} \times \texttt{nuEToNuE[kLogE][jLogE]} \\
&\qquad + \texttt{FnuEToMu[iLogE][kLogE]} \times \texttt{muToNuE[kLogE][jLogE]} \\
&\qquad + \texttt{FnuEToTau[iLogE][kLogE]} \times \texttt{tauToNuE[kLogE][jLogE]} \}
\end{aligned}$$

$$\texttt{dNFromNuE} \mathrel{+}= \texttt{FnuEToNuE[iLogE][jLogE]} (1 - \texttt{intProbNeutrino[jLogE]} - \texttt{intProbNuE[jLogE]})$$

$$\texttt{FnuEToNuE[iLogE][jLogE]} = \texttt{dNFromNuE},$$

where ($1 -$ `intProbNeutrino[jLogE]` $-$ `intProbNuE[jLogE]`) indicates the probability of survive electron neutrino. The particle energy distribution after propagation over n$\times$dX [g] can be, for example, obtained by calling this method n times.

`public void propagateDXpowered()`

This method doubles the finite propagation matrix, `FnuEtoNuE[iLogE][jLog]` etc. calculated by propagateDX(), i.e.

$$\begin{aligned}
&\texttt{FnuEToNuE[iLogE][jLogE]} = \\
&\quad \textstyle\sum_{kLogE} \{ \ \texttt{FnuEToNuE[iLogE][kLogE]} \times \texttt{FnuEToNuE[kLogE][jLogE]} \\
&\qquad + \texttt{FnuEToNuMu[iLogE][kLogE]} \times \texttt{FnuMuToNuE[kLogE][jLogE]} \\
&\qquad + \texttt{FnuEToNuTau[iLogE][kLogE]} \times \texttt{FnuTauToNuE[kLogE][jLogE]} \\
&\qquad + \texttt{FnuEToMu[iLogE][kLogE]} \times \texttt{FmuToNuE[kLogE][jLogE]} \\
&\qquad + \texttt{FnuEToTau[iLogE][kLogE]} \times \texttt{FtauToNuE[kLogE][jLogE]} \ \}.
\end{aligned}$$

In order to propagate the particles over 2^n DX [g/cm^2], this method should be called "n" times.

`public void propagateX()`

Propagate the particles over X [g/cm^2] where X is the propagation distance for the finite propagation

matrix. It multiplies the finite propagation matrix calculated by `propagateDX()` and `proagateDXpowerd()`, i.e.

$$\begin{aligned}&\texttt{FnuEToNuE[iLogE][jLogE]} = \\ &\quad \textstyle\sum_{kLogE} \{\ \texttt{FnuEToNuE[iLogE][kLogE]} \times \texttt{nuEToNuE[kLogE][jLogE]} \\ &\qquad + \texttt{FnuEToNuMu[iLogE][kLogE]} \times \texttt{nuMuToNuE[kLogE][jLogE]} \\ &\qquad + \texttt{FnuEToNuTau[iLogE][kLogE]} \times \texttt{nuTauToNuE[kLogE][jLogE]} \\ &\qquad + \texttt{FnuEToMu[iLogE][kLogE]} \times \texttt{muToNuE[kLogE][jLogE]} \\ &\qquad + \texttt{FnuEToTau[iLogE][kLogE]} \times \texttt{tauToNuE[kLogE][jLogE]}\ \}.\end{aligned}$$

where `nuEToNuE[kLogE][jLogE]` indicates finite propagate matrix copied by `copyTransferMatrix()`.

`public void copyTransferMatrix()`
Copy the propagate matrices to the transfer matrices, i.g.

$$\texttt{nuEToNuE[iLogE][jLogE]} = \texttt{FnuEToNuE[iLogE][jLogE]}.$$

`public void storePropagateMatrix()`
Store the propagation matrices calculated so far to the store matrix i.g. `SnuEToNuE` which save energy distribution of neutrinos and leptons propagating to the current location.

$$\begin{aligned}&\texttt{SnuEToNuE[iLogE][jLogE]} = \\ &\quad \textstyle\sum_{kLogE} \{\ \texttt{SnuEToNuE[iLogE][kLogE]} \times \texttt{FnuEToNuE[kLogE][jLogE]} \\ &\qquad + \texttt{SnuEToNuMu[iLogE][kLogE]} \times \texttt{FnuMuToNuE[kLogE][jLogE]} \\ &\qquad + \texttt{SnuEToNuTau[iLogE][kLogE]} \times \texttt{FnuTauToNuE[kLogE][jLogE]} \\ &\qquad + \texttt{SnuEToMu[iLogE][kLogE]} \times \texttt{FmuToNuE[kLogE][jLogE]} \\ &\qquad + \texttt{SnuEToTau[iLogE][kLogE]} \times \texttt{FtauToNuE[kLogE][jLogE]}\ \}.\end{aligned}$$

You can then initialize the propagation matrices by init() and start another propagation calculation in a different section of the trajectory. You can bring the results stored here back to the propagation matrix by calling the method `copyTransferMatrixFromStore()`.

`public void copyTransferMatrixFromStore()`
Copy the store matrices to the propagate matrices, e.g.

$$\texttt{FnuEToNuE[iLogE][jLogE]} = \texttt{SnuEToNuE[iLogE][jLogE]}.$$

`public void setDx(double dX)`
Set infinitesimal propagation length.

`public double getFnuEToNuE(int iLogE,int jLogE)` **etc.**
Get the element of propagation matrix corresponding given `iLogE` and `jLogE`, i.e. from ν_e which energy is `iLogE` to ν_e which energy is `jLogE`.

8.2 RunPropagationMatrix.java

This class runs the PropgationMatrix to calculate the energy distribution and the flux of particles after propagation in the Earth. In order to save CPU, the propagate matrix is multiplied 2^k times. In order to avoid sizable error, the trajectory length is divided to 5 sections. The tracing the particle geometry is done by the `PrticlePoint` object descried in Chapter 4.

The parameters provided in this class are:
`Particle nuE etc.` Particle object e.g. ν_e

`ParticlePoint s` Point of the propagation
`PropagationMatrix propMtx` PropagationMatrix object

Constructor : `public RunPropagationMatrix(double nadirAngle, int intSwitch,int decaySwitch, int mediumNumber) throws IOException`

Generate the `ParticlePoint`, `Particle`, and the `PropagationMatrix` objects. The medium is assumed to be the standard rock.
The parameters given to this constructor are:

double nadirAngle:	Nadir angle [deg] of trajectory of the incoming particles.
int inSwitch:	Interaction switch to turn on/off the individual interaction channel.
int decaySwitch:	Decay switch to turn on/off the individual decay channel.
int mediumNumber:	Medium number 0→ice, 1→ rock

Constructor : `public RunPropagationMatrix(double nadirAngle, int intSwitch int decaySwitch, int mediumNumber, double neutrinoFactor) throws IOException`

Same as above, but generating the `PropagationMatrix` objects with `neutrinoFactor` which enhances cross section of neutrino CC and NC interaction.

`public void traceParticles(double trajectoryLength)`

This method divides given trajectory length into 5 sections. Then repeats following calculation **1)** ∼**7)** 5 times.

1) Build the elementary infinitesimal propagation matrices by
   `propMtx.calculateTransferMatrix();`,
   where `propMtx` is `PropagationMatrix` object generated in the constructor.
2) Trace the propagation steps in order to obtain how many steps is needed to propagate the section (`n`).
3) Propagate the particles over 10×dX [g/cm^2], i.e. infinitesimal propagation is multiplied 10 times by calling
   `propMtx.propagateDX()` × 10 times.
4) Propagate the particles over 2^k times, i.e. finite propagation matrix is multiplied 2^k times by calling
   `propMtx.propagateDXpowered()` × k times,
   where k is determined by `n` calculated in **2)**.
5) During **4)**, store the matrices by calling
   `propMtx.copyTransferMatrix()`,
   when particles reaches 3 % of the total path in their journey.
6) Store the resultant matrices by calling
   `propMtx.storePropagateMatrix()`.
7) Initialize the propagation matrices by
   `propMtx.init()`.

Then the stored matrix is copied back to the propagation matrix by calling
`propMtx.copyTransferMatrixFromStore()`.
Finally, it propagates the particles by multiple the matrix stored in **5)** by calling
`propMtx.propagateX()`,
until the particles reaches the final point.

`public void traceParticles()`

Same as the `traceParticles(double trajectoryLength)`, but the end point is decided to be where the particles emerged from the underground.

`public void saveMatrix(DataOutputStream out) throws IOException`

Write out the calculated propagation matrices to given DataOutputStream, `out`.

8.3 MainRun.java

This class has the main method to start the simulation. A user has to input parameters; `NadirAngle[deg]`, `(trajectoryLength[cm])`, `intSwitch, decaySwitch, upDownFlag, output file-name`.

By running this class with a given nadir angle, you can obtain the energy distribution matrices after the propagation in the earth with the angle. When you omit the trajectoryLength in running the main method, the `ParticlePoint` object calculated the whole trajectory length for a given nadir angle from entering to the earth through emerging from the earth.

The parameters provided in main method are:

`String fileName` Name of the output file you want to make
`int intSwitch` Interactions switch defined in PropgationMatrix class
`int decaySwitch` Decay switch defined in PropgationMatrix class
`int upDownFlag` 0→Down-going, 1→Up-going
`int mediumNumber` 0→ice, 1→rock
`double nadirAngle` Nadir angle[deg]
`double trajectoryLength` Trajectory length[cm]
`double detectorDepth` Depth of detector[cm]
`RunPropagationMatrix run` RunPropagationMatrix object

`public static void main(String[] args) throws IOException`

In the case of considering Up-going propagation, the medium has to be rock, On the other hand, in the case of Down-going, the medium has to be ice. In the both case, particles are traced by the `traceParticles()` supplied by RunPropagationMatrix class. When the Down-going particles are considered, we consider it as Up-going propagation by redefining the relevant geometrical parameters with the zenith angle equals to original nadia angle (See Fig.8.1 and 8.2).

Example of run

at ~/javalib/classes/

```
% java -Xms128m -Xmx256m iceCube.uhe.propagation.MainRun
          NadirAngle[deg] (trajectoryLength[cm]) intSwitch decaySwitch
                                                 upDownFlag file-name
```

A user has is required to input some arguments such as `NadirAngle[deg]` except `trajectoryLength[cm]`. If users did not input the `trajectoryLength[cm]`, the default value (equals to axis length; See Fig.8.1, 8.2) is used for the calculation.

8.4 PropagationMatrixFactory.java

PropagationMatrixFactory provides interface methods to access propagation matrix data stored in the disk. The matrix data generated by RunPropagationMatrix (Section 8.2)is written in form of simple two dimensional array of double variables. An object using the matrix data can easily access an element of the matrix data array via PropagationMatrixFactory's getDF(Particle inParticle, Particle outParticle) method.

An example program to use this class is DrawPropagationMatrixByFactory.java in the propagation package.

The parameters provided in this class are:

`boolean includeGlashowResonance`

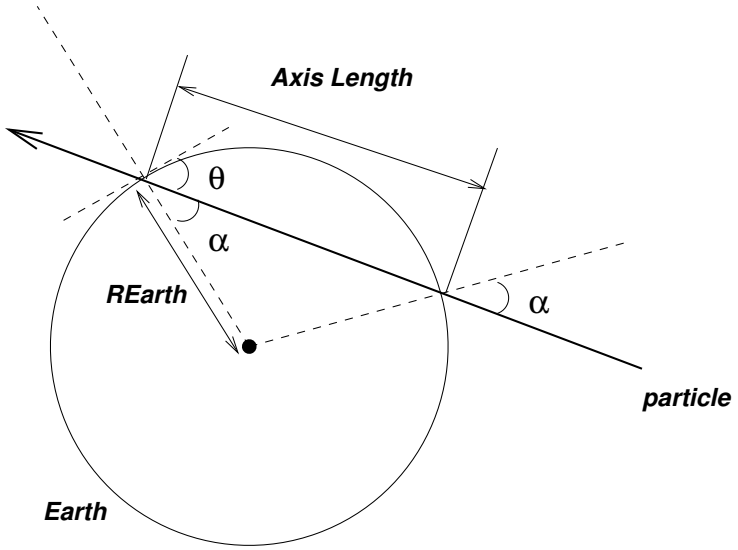

Figure 8.1: Up-going Propagation. α is Nadir Angle and θ is Zenith Angle.

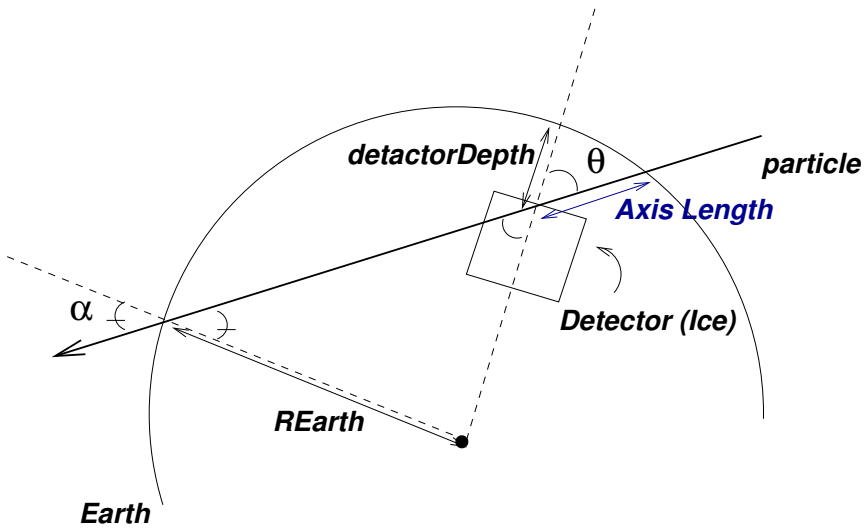

Figure 8.2: Down-going Propagation. The red arrow indicates that we can think Down-going propagation as Up-going propagation by re-calculation of "new" Nadir Angle. α is Nadir Angle and θ is Zenith Angle.

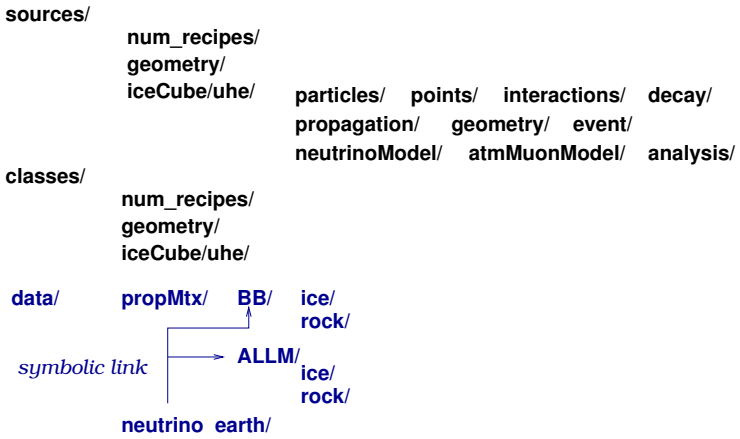

Figure 8.3: Directory structure of the JULIeT source and class files with the suggested structure where the generated matrix data files should be contained.

Flag:false→read a old matrix data without the Glashow resonance.
You can not read the old matrix by default, because the old matrix have smaller dimension due to the additional channels by the Glashow resonance. This flag can be handled by calling `whetherPropagationMatrixWithGlashowResonance(false)` in the `PropagatingNeutrinoFlux` or `PropagatingAtmMuonFlux` class (See section 12.2 or 13.5).

Constructor : `public PropagationMatrixFactory()`
Generates the double array to store the matrix data.

`public void readMatrix(DataInputStram in)`
Reads out the (pre-generated) matrix array from a given input stream.

`public double getDF(Particle inputParticle, Particle outputParticle)`
Return the matrix element, $dF/dLogE \times \Delta LogE$ of *outputParticle* originated in *inputParticle* before the propagation.

```
public double getDF(int inputFlavor, int inputDoublet, double logEinput,
                    int outputFlavor, int outputDoublet, double logEoutput)
```
Return the matrix element, $dF/dLogE \times \Delta LogE$ of particle of inputFlavor, inputDoublet (See Section 5.1) with $\log_{10}$(Energy [GeV])= logEinput originated in a particle of outputFlavor, outputDoublet with $\log_{10}$(Energy [GeV]) = logEoutput

A couple of the application classes in the `analysis package` uses this factory.

8.5 Suggested directory for generated propagation matrix data files

The matrix data generated by RunPropagationMatrix (Section 8.2) can be written in form of the data files in any directory. The default directory is, however assumed in some application/demo programs using the pre-generated matrix data. Figure 8.3 shows the directory structure of the JULIeT. The matrix data is suggested to be put in the directory called `data/propMtx` in the same level of the source and class directories.

Under the matrix directory, the subdirectory categorized by the photo-nuclear interaction model exists. Then the subdirectories `ice/` and `rock/` includes the matrix data files in the corresponding medium. The symbolic link, `neutrino_earth` links to your default model directory.

DrawAngularNeutrinoFlux.java/DrawNeutrinoFluxIntegral.java in the neutrino model package, for example, assumes this directory structure, to read the matrix data and draw the particle fluxes. Maintaining of the directory structure shown in Fig. 8.3 is essential in the analysis package when you weight I3Particle events with the flux/propagation matrix values.

You can download the generated matrix data files required by those applications from the JULIeT website http://www.ppl.phys.chiba-u.jp/JULIeT/ in the form of the gzipped tar ball. Unpacking The tar files indeed yields the directory structure suggested here.

Chapter 9

Package : geometry

Although the point package described in Chapter 4 handles the particle location along its track as shown in Fig. 4.1 and it is indeed responsible for particle tracking in the propagation package (Chapter 8), it is necessary to deal with 3-dimensional description of particle geometry in order to generate MC events since our detector is a 3D array and geometrical relation between any given DOM (detector) and the particle location must be described in 3-dimensional space. From this reasons, this package contains a couple of base class on vectors and coordinates. The iceCube.uhe.geometry package described later inherits some of them from this package.

The contained classes are:

- J3Vector
- J3UnitVector
- J3Line
- Coordinate
- EarthCenterCoordinate
- EarthLocalCoordinate

9.1 J3Vector.java

This is a base class to handle a vector **a** in 3D space.

The parameters provided in this class are:

`private double x` x component of vector
`private double y` y component of vector
`private double z` z component of vector
`private double length` absolute length of vector

Constructor : `public J3Vector(double x, double y, double z)`
Initialize the vector by given arguments and calculate the vector length immediately by calling the method setLength().

Constructor : `public J3Vector()`
Generate the vector $\mathbf{a} = (0.0, 0.0, 1.0)$ as default.

`public void setX(double x)`
Set x component and calculate the vector length immediately by calling the method setLength().

`public void setY(double y)`
Set y component and calculate the vector length immediately by calling the method setLength().

`public void setZ(double z)`
Set z component and calculate the vector length immediately by calling the method setLength().

`public void putVector(J3Vector a)`
Make copy-operation. It copes the given vector.

`public double getX()`
Get x component of the vector.

`public double getY()`
Get y component of the vector.

`public double getZ()`
Get z component of the vector.

`public void setLength()`
Calculate the length of vector **a**.

`public double getLength()`
Returns the length of vector **a**.

`public static double getDotProduct(J3Vector a, J3Vector b)`
Calculate the dot product **ab**.

`public static double getAgnleInRadian(J3Vector a, J3Vector b)`
Calculate the angle [rad] between **a** and **b**.

`public static double getAgnleInDegree(J3Vector a, J3Vector b)`
Calculate the angle [deg] between **a** and **b**.

`public static J3Vector add(J3Vector a, J3Vector b)`
Return the vector $\mathbf{c} = \mathbf{a} + \mathbf{b}$.

`public static J3Vector subtract(J3Vector a, J3Vector b)`
Return the vector $\mathbf{c} = \mathbf{a} - \mathbf{b}$.

`public void increment(J3Vector a, J3Vector b)`
Change **a** to $\mathbf{a} = \mathbf{a} + \mathbf{b}$.

`public void decrement(J3Vector a, J3Vector b)`
Change **a** to $\mathbf{a} = \mathbf{a} - \mathbf{b}$.

`public static J3Vector getCrossProduct(J3Vector a, J3Vector b)`
Calculate the cross product $\mathbf{a} \times \mathbf{b}$.

`public static J3Vector multipleFactor(double f J3Vector a)`
Return the vector $f \times \mathbf{a}$.

9.2 J3UnitVector.java

This is an inheritance class from J3Vector, describing an unit vector.

Constructor : `public J3UnitVector(double x, double y, double z)`
Generate the vector by given arguments with nomalization so that $|\mathbf{a}| = 1$.

9.3 J3Line.java

This is an inheritance class from J3Vector to represent a line vector $\mathbf{a} = \mathbf{r_0} + l\dot{\mathbf{n}}$ where $\mathbf{r_0}$ is loaction of a fixed point on this line, $\mathbf{n}$ is the unit vector of its direction, and l corresponds to axis length of this line.

The parameters provided in this class are:

`double axisLength` axis length of line, l
`J3UnitVector n` direction of line, $\mathbf{n}$
`J3Vector r0` a fixed point on this line $\mathbf{r_0}$

Constructor : `public J3Line(J3Vector fixedPoint, J3UnitVector n, double l)`
Generate line vector $\mathbf{a} = \mathbf{r_0} + l\dot{\mathbf{n}}$.

Constructor : `public J3Line(J3Vector fixedPoint, J3UnitVector n)`
Same as above, but $l = 0$ as default.

`public void configure()`
Calculate the line's x, y, and z component and set them to the member variables for given $\mathbf{r_0}, l$, and $\mathbf{n}$. The methods described below calls this method whenever necessary and you do not have to call it directly except for debugging purposes.

`public void setAxisLength(double l)`
Set axis length and calculate line vector component by configure().

`public void setR0(J3Vector r0)`
Set $\mathbf{r_0}$ and calculate line vector component by configure().

`public void setDirection(J3UnitVector n)`
Set $\mathbf{n}$ and calculate line vector component by configure().

`public double getAxisLength()`
Return l.

`public J3UnitVector getDirection()`
Return $\mathbf{n}$.

`public J3Vector getR0()`
Return $\mathbf{r_0}$.

9.4 Coordinate.java

This class describes a right-handed coordinate. Many methods to transform vector from one coordinate to another are also provided.

The parameters provided in this class are:

`J3Vector origin` Location of the origin of this coordinate
`J3Vector ex` x axis
`J3Vector ey` y axis
`J3Vector ez` z axis

Those parameters above are defined at the point where a particle emarges from underground. See Fig.9.1.

Constructor : `public Coordinate(J3Vector origin, ex, J3UnitVector ez)`
Generate the coordinate system with given origin, x and z axises. y axis is given from the right-handed rule as shown in Fig. 9.1.

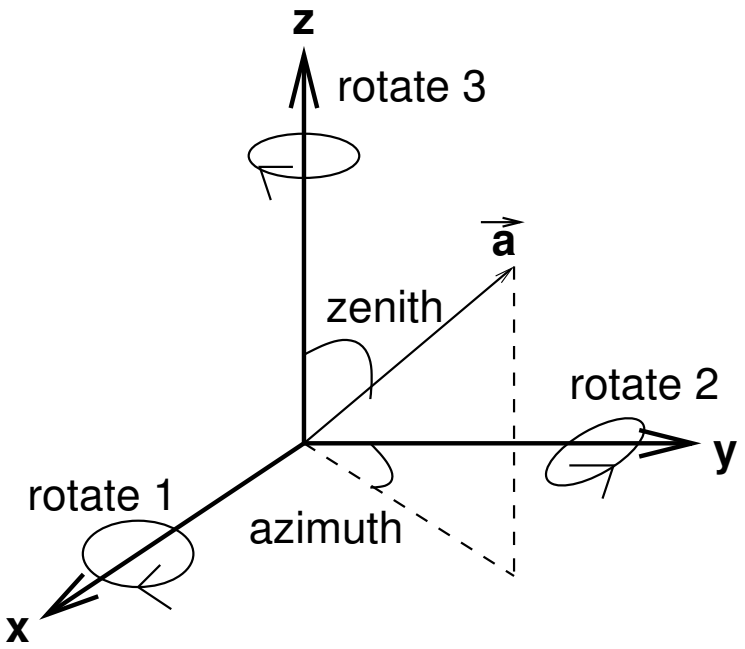

Figure 9.1: Coordinate system defined in this class. Zenith, azimuth, and rotation oeration are also illustrated.

Constructor : `public Coordinate(J3Vector origin)`
Generate the coordinate system with given origin, $ex = (1, 0, 0) ey = (0, 1, 0) ez = (0, 0, 1)$.

Constructor : `public Coordinate()`
Same as above, but its origin is localted at (0, 0, 0).

`public J3Vector getPointVectorFromPolarCoordinate(double r, double theta, double phi)`
Generate a vector represened by the polar system given in the argument. The zenith angle, theta, and the azimuth angle, alpha, are assumed to follow the definition shown in Fig. 9.1.

`public void rotate(int axis, double theta)`
Rotate this coordinate around axis by angle theta. Its definition is illustrated in Fig. 9.1.

`public void transformThisCoordinate(J3UnitVector nz, double alpha)`
Tansform this coordinate to that with the Z axis parallel to the nz vector. After this transformation, the new x-y plain is rotated by the given rotation angle alpha [rad] to redefine the directions of x and y axis. Alpha = 0 makes the (new) x axis stay on the original x-y plain.

`public void setOrigin(J3Vector origin)`
Set the new origin defined by J3Vector origin.

`public J3Vector getEx()`
Get the x axis vector.

`public J3Vector getEy()`
Get the y axis vector.

`public J3Vector getEz()`
Get the z axis vector.

`public J3Vector getOrigin()`
Get the location of the origin.

`transformVectorToThisCoordinate(J3Vector r, Coordinate external)`

Transform the vector r represented by the coordinate "external" to that described by this coordinate. The coordinate "external" must be described by the base coordinate with x axis $(1,0,0)$, y axis (0.1.0) and z axis $(0,0,1)$. In the JULIeT library, a realization of this "base" coordinate is the EarthCencterCoordinate illustrated in Section 9.5.

`transformUnitVectorToThisCoordinate(J3UnitVector n)`

Transform the unit vector r represented by the base coordinate to that described by this coordinate. A difference from transformVectorToThisCoordinate is that here n is condered as a direction and no parallel transformation is made.

9.5 EarthCenterCoordinate.java

This class describes the base coordinate extended from Coordinate.java, but assumed that its origin $(0,0,0)$ is located at center of the Earth and that y axis $(0,1,0)$ points to Prime Meridian towards the Greenwich, z axis $(0,0,1)$ to the North Pole. It provides several methods to handle latitute/longitude parameters.

The parameters provided in this class are:

`public final static double REarth` Radius of the earth

Constructor : `public EarthCenterCoordinate()`

Simply calls super().

`public double getLongitude(J3Vector r)`

Returns the longitude of the vector **r** in unit of degree. A positive value corresponds to West while a nagative to East.

`public double getLatitude(J3Vector r)`

Returns the latitude of the vector **r** in unit of degree. A positive value corresponds to North while a nagative to South.

`public J3Vector getPointVectorFromLongitudeLatitude(double height, double longitude, double latitude)`

Generate the vector given by longitude [deg], latitude [deg], and height [cm] from the earth center.

`public J3Vector getSurfacePoint(double longitude, double latitude)`

Generate the vector located at the earth surface given by longitude [deg] and latitude [deg].

9.6 EarthLocalCoordinate.java

This class describes a local coordinate with its origin inside or on the earth. The z axis always points to the local upward direction, *i.e.*, that from the earth center to its origin. Inheriting from Coordinate.java, this is a base class to realize many coordinates. The IceCube coordinate system would be described using this class, for example.

Constructor : `public EarthLocalCoordinate(J3Vector origin, double alpha)`

Generate the local coordinate system with given origin. The xy plain is roated by an angle of alpha in the end by calling Coordinate.transformThisCoordinate(nz,alpha).

Constructor : `public EarthLocalCoordinate(J3Vector origin)`

Generate the local coordinate system with given origin. No final ratation is made so that the y axis points to local South.

Constructor : `public EarthLocalCoordinate(double height, double longitude, double latitude, double alpha`

Generate the local coordinate with orgin located at given longitude, latitude, and height from the earth center.

`public J3Vector transformVectorToEarthCenter(J3Vector r_local)`

the vector $\mathbf{r_{local}}$ represented by this local coordinate to those by the EarthCenter coordinate with $\mathbf{ex} = (1, 0, 0)$, $\mathbf{ey} = (0, 1, 0)$ and $\mathbf{ez} = (0, 0, 1)$. Returns the transformed vector.

`public J3Vector transformUnitVectorToEarthCenter(J3Vector n_local)`

Same as above, but the vector is an unit vector.

Chapter 10

Package : iceCube.uhe.geometry

This package contains the classes that involves the geometry of the IceCube observatory such as its coordinate system. They are inherited from or using the classes and their methods in the general geometry package described in Chapter 9.

The contained classes are:

- IceCubeCoordinate
- Volume
- IceCubeVolume
- ParticleTracker

10.1 IceCubeCoordinate.java

This class provides the IceCube coordinate system where y axis points to the grid north and z axis is normal to the earth surface and verticaly upward. It is extended from EarthLocalCoordinate (Section: 9.6).

The parameters provided in this class are:

`public static double northing` North coordinate of the IceCube center

`public static double easting` East coordinate of the IceCube center

`public static double elevation` Elevation above sea level of the IceCube center [cm]. The sea level is assumed to be "rock" surface in the JULIeT simulation.

`double glacierDepth` Depth of the Polar glacier [cm]. Initialized by the default value 2.829903408e5 cm.

Constructor : `public IceCubeCoordinate()`

Generate the IceCube coordinate system.

`public void setGlacierDepth(double depth)`

Set the depth of the glacier depth.

`public double getGlacierDepth()`

Return the glacier depth.

10.2 Volume.java

This class defines a cubic volume. Used for generating the "detector" volume in the JulietEventGenerator. The center of the volume is the origin of the IceCubeCoordinate (See Section 10.1). All the geometry parameters defined in this class is represented by the IceCube coordinate system.

The parameters provided in this class are:

`J3Vector p1,p2,p3,p4,p5,p6` Normal vectors to define 6 planes of the cubic volume.

Constructor : `public Volume(double sizeOfOneSide)`

Generate the cubic volume with dimension of sizeOfOneSide.

```
public boolean isInsideVolume(J3Vector r)
public boolean isInsideVolume(J3Vector r, J3Vector shift)
```

Return whether the location vector **r** or (**r** - **shift**) represented by the IceCube coordinate system (Section 10.1) is inside this volume or not.

```
public boolean isJ3LineInsideVolume(J3Line line,
                                    double axisLengthFrom, double axisLengthTo)
public boolean isJ3LineInsideVolume(J3Line line, J3Vector shift,
                                    double axisLengthFrom, double axisLengthTo)
```

Return whether the line vector **line** or (**line** - **shift**) penetrates this volume in the range of axis length from axisLengthFrom to axisLengthTo. The J3Line vector **line** must be represented by the IceCube Coordinate System.

10.3 IceCubeVolume.java

It generates the volume with demension of 1.0×10^5 [cm].

Constructor : `public IceCubeVolume()`

Generate the IceCube volume.

10.4 ParticleTracker.java

Provides a few static methods useful for geometrical calculations in the IceCube coordinate system.

```
public static void setInitialPoint(J3Line axis, IceCubeCoordinate iceCube)
```

Set the J3Line axis to where it enters into the glacier surface from the upper atmosphere. axis must be represented by the base coordinate, EarthCenterCoordinate (See Section 9.5).

```
public static boolean isInsideEarth(J3Vector r_center,
                                    IceCubeCoordinate iceCube, EarthCenterCoordinate center,

                                    Volume outVol, int flag)
public static boolean isInsideEarth(J3Vector r_center, J3Vector shift,
                                    IceCubeCoordinate iceCube, EarthCenterCoordinate center,
                                    Volume outVol, int flag)
```

Check whether the particle location defined by the EarthCenterCoordinate (See Section 9.5), $\mathbf{r_{center}}$, or $\mathbf{r_{center}}$ shifted by the J3Vector **shift**, is inside the Earth. It considers the depth of the glacier. The **shift** must be defined by IceCubeCoordinate (See Section 10.1). When flag = 1, returns false if it is outside the volume outVol.

Chapter 11

Package : iceCube.uhe.event

This package contains the classes related to 'occurrence of event', In this package, there is the RunManager class which has main method.
The contained classes are:

- RunManager
- Event
- MonteCarloBase

11.1 MonteCarloBase.java

This class is an abstract class. This is a super class of InteractionsBase ,ElectronBase and GlashowResonanceBase which are classes of interactions package and MuDecayBase and TauDecayBase which are classes of decay package. In order to register interactions and decays in a list, all the interactions/decays should have a common super class. This abstract class defines the methods for both interactions and decay that determines the path length and produced energy with the Monte Carlo method. The actual implementation of the methods are made by each 'Base' class. This class also defines methods to get the information of interactions/decays so that the Event class can know that information.

The parameters provided in this class are:

`static final double initialLogEnergyProducedMinimum`	Minimum value of log energy of produced particles

`public static double getLogEnergyProducedMinimum()`
Get `initialLogEnergyProducedMinimum` which is the minimum value of log energy of produced particles. In the default, it equals to 2.0, so produced energy can be from 10^2[GeV] to 10^{12}[GeV].

`public abstract double getPathLength(int iLogE, RandomGenerator rand);`
`public abstract double getPathLength(double logEnergy, RandomGenerator rand)`
Get path length of interactions involved. The method, `getPhysicalPathLength()`, in the Event class calls this method in order to compare the lengths and to choose the shortest one.

`public abstract double getProductEnergy(int iLogE, RandomGenerator rand)`
`public abstract double getProductEnergy(double logEnergy, RandomGenerator rand)`
Get the log energy of the produced particle.

`public abstract double getNeutrinoPathLength(int iLogE, RandomGenerator rand)`
`public abstract double getNeutrinoPathLength(double logEnergy, RandomGenerator rand)`
Get the logEnergy of the produced particle.

`public abstract int getPropFlavor()`
Get the flavor of the particle propagating.

`public abstract int getPropDoublet()`
Get the doublet of the particle propagating.

`public abstract int getProducedFlavor()`
Get the flavor of the produced particle.

`public abstract String getInteractionName()`
Get the name of interaction.

`public abstract int getTypeOfInteraction()`
Get the flag indicates type of interaction. Interaction→0, Decay→1.

11.2 Event.java

This class defines the behavior of an event. This is the fundamental class for propagating particles by the Monte Carlo method. This class chooses the interaction returns the shortest path length, and makes an 'event' then propagates particles. So Event object has the information of an interaction/decay which has occurred, *e.g.* transfered energy, produced particle species, path length etc.

The parameters provided in this class are:

`static final double ln10`	Common logarithm
`static final double epsilon`	Round off error
`List MonteCarloList`	List to register the MonteCalroBase objects
`ListIterator MonteCarloIterator`	Iterator
`private double stepDx`	Step size[g/cm^2]
`ParticlePoint point`	ParticlePoint object
`double massNumber`	MassNumber in the medium
`Particle propParticle`	Particle which is propagating
`MonteCarloBase mcBaseInPlay`	MonteCarloBase object currently in play
`private double cascadeEmgEnergy`	Energy of electromagnetic cascade
`private double cascadeHadronEnergy`	Energy of hadronic cascade
`private int expandedDim`	Dimension of produced log energy

Constructor : `public Event(MonteCarloBase[] mcBases, Particle p, ParticlePoint s)`
Register the Particle and ParticlePoint objects. Given Particle object indicates the particle which is propagating. It sets the `massNumber` corresponding to given ParticlePoint object. It generates the list and its iterator to register `MonteCarloBases`.

`void setMassNumber()`
Calculate the mass number in the current medium. Sum up all the mass number of atoms. included in the medium.

`double getMassNumber()`
Get the mass number in the current medium.

`public void registerMonteCarloBase(MonteCarloBase[] mcBases)`
Register the MonteCarloBase objects which are elements of the array, `mcBases` to the `MonteCarloList`.

`public double getPhysicalPathLength(RandomGenerator rand)`
Calculate the path length determined by sampling the interaction points for all the interactions/decays channels registered to the `MonteCarloBase`. The shortest path length is picked up and returned. The determined interaction/decay which is picked up is put in the `mcBaseInPlay`.

Calculation of the path length:

a) In the case of Changed Lepton interaction
The number of particles when they propagate X[g/cm^2] is given by

$$\frac{dN}{dX} = -\sigma(E)N,$$
$$N(E,X) = N(E,X_0)\ \exp[-(X-X_0)N_A\ \sigma(E)],$$

where N_A is the Avogadro's Number. Therefore, the probability that interaction has occurred while a particle propagates X[g/cm^2] is given by

$$P(E,X) = 1 - exp[-XN_A\ \sigma(E)].$$

The P(E,X) is determined by random number(0$\sim$1). So the path length is

$$X = \frac{-1}{N_A\ \sigma(E)}\ \ln(1-r),$$

where r is random number.

b) In the case of Neutrino Interaction
Same as above, but with the neutrino cross section weight factor, w,

and the path length is calculated by

$$X = \frac{-1}{N_A\ w\sigma(E)}\ \ln(1-r),$$

where r is random number.
w is settable by setNeutrinoInteractionWeight(int weight)
of JulietEventGenerator class (See:Section 11.3)
that rewrites the static member variable `neutrinoFactor`
of InteractionsBase class (Section 6.16).

c) In the case of Decay
The number of particles when they propagate while t[sec] is given by

$$\frac{dN}{dt} = -\frac{1}{\tau(E)}N(t),$$
$$N(E,t) = N(t_0)\ \exp\Big[-\frac{t}{\tau(E)}\Big],$$

where τ is the lifetime of the particle. Therefore, the probability that Decay has occurred while a particle propagates t[sec] is given by

$$P(E,t) = 1 - \exp\Big[-\frac{t}{\tau(E)}\Big].$$

From $X = ct\rho$ (c is light speed and ρ is the density of the medium[g/cm^3]),

$$P(E,X) = 1 - \exp\Big[-\frac{X}{c\tau(E)\rho}\Big].$$

The P(E,X) is determined by random number(0$\sim$1). So the path length is

$$X = -c\tau(E)\rho\ \ln(1-r),$$

where r is random number.

`public String interactionsNameInPlay()`
Return the name of interaction/decay which has just occurred, i.e. the name of the `mcBaseInPlay`.

`public int getFlavorByIntereactionsInPlay()`
Get the produced particle's Flavor defined in Particle class (See Table.5.1).

`public double collideNow(RandomGenerator rand)`
The propagation particle now collides with nuclear/nucleon or decays, then changes its energy via `mcBaseInPlay` which is determined by `GetPhysicalPathLength()`. The energy of the produced particle (which may be equal to cascade energy depending on the interaction) is returned.
The boolean parameters, `compareNC` `compareCC` and `compareGR` is the flags for checking Neutrino interactions. First, if the `mcBaseInPlay` is Neutral current interaction, the propagating neutrino has to be changed into neutrino, i.e. the same kind of particle. Second, if the `mcBaseInPlay` is Charged Current interaction, the neutrino has to be changed into the charged lepton which has the same flavor of incident particle. Third, if the `mcBaseInPlay` is the Glashow resonance, the electron neutrino is changed into a charged lepton according to channel. After the leptonic resonance, propagation of the produced neutrino is ignorable, because of its long mean free path.
The `recoilEnergy` will be the new energy of the `propParticle` after the interaction/decay. If the `recoilEnergy` is less than the mass of the particle, `propParticle` will stop by RunManager, but it remains to have its own mass as its energy. This method also looks over the energy and kind(Electromagnetic or Hadronic) of produced cascade. In order to know the whole energy which particles deposit during their journey, `cascadeEmgEnergy` and `cascadeHadronEnergy` is summed up.

`public double getCascadeEmgEnergy()`
Get the produced energy transfered to electromagnetic cascades. The `cascadeEmgEnergy` is summed up in the `collideNow(RandomGenerator rand)`, therefore this is the whole energy of produced electromagnetic cascades while their journey.

`public double getCascadeHadronEnergy()`
Get the produced energy transfered to hadronic cascades. The `cascadeHaronEnergy` is summed up in the `collideNow(RandomGenerator rand)`, therefore this is the whole energy of produced hadronic cascades while their journey.

`public double getCascadeTotalEnergy()`
Get the total cascade energy, i.e. the sum of `cascadeEmgEnergy` and `cascadeHadonEnergy`.

`public void changeParticle(int newFlavor, int newDoublet)`
Change the particle into different particle by changing flavor and doublet. Note that this method don't change the energy. This method is called when the decays or neutrino interaction have occurred.

`public void setStepDx(double dx)`
`public double getStepDx()`
Set/get the step size for `traceParticle(double pathlength)`.

`public void traceParticle(double pathlength)`
Trace particle by given path length [g/cm^2], and changes the point of the particle.

11.3 JulietEventGenerator.java

This class is a class to generate MC events using Event class described in Section 11.2. The method runSingleEvent() generates an event when called. A basic procedure for generating MC events using this class is following.

1. The constructor **JulietEventGenerator()** reads the Interaction Matrices (See Section 6.2) which are necessary for event generation. It calls **configureJULIeT()** that reads the files of the Interaction Matrices and generate the relevant Interactions Base Matrices (See Section 6.16) and Decay Base Matrices (Sections 7.6 and 7.7).

2. call the method **definePropagatingParticle()** to set the propagating particle. Usually it has already been set by the constructor and you do not have to call this method unless you need to run particles different from the ones you set in the constructor.

3. call the method **definePropagationGeometry()** to set the geometry of the propagating particle. The geometrical parameters given here is in the form of J3Line defined in the IceCube Coordinate class (See Section 10.1).

4. call the method **configurePropagationGeometry()**. It calculates the geometry in the Earth Center Coordinate (Section 9.5) and the location where the particle enters into the earth. Depending the option flag, you can start the propagation at this point, or start it at 880 m away from the IceCube center for saving CPU time. This methods registers this starting location following the flag you input. The methods **wherePrimaryParticleStartsInEarthCenterCoordinate()** and **wherePrimaryParticleStartsInIceCubeCoordinate()** tells you this location in form of J3Vector.

5. call **runSingleEvent()** to generate an event. The results are stored in the Java Lists "particleList", "locationIce3List" concerning secondary produced particles during the propagation, that generates emg/hadronic cascades, "trackParticleList" and "trackLocationIce3List" for storing information about the primary tracks. All of them can be readout by calling the relevant method like **getParticleIterator()** provided by the JulietEventGenerator class.

An example to show how to generate the events using this class is **RunJuliet.java** in the event package. Its JNI version, written by C++ are **RunJuliet.cxx** and **RunJuliet.h**. They have been a reference to call the JulietEventGenerator under the IceCube offline software environment where C++ is the default language for a project. The "juliet-interface" project in the IceCube offline software used them as a prototype.

The parameters provided in this class are:

`ParticlePoint point` ParticlePoint object
`int materialNumber` Ice→0, Rock→1

`Particle propParticle` Particle object which indicates the particle propagating
`double primaryEnergy` Initial value of the propagating particle's energy
`static int primaryFlavor` Flavor of the primary particle
`static int primaryDoublet` Doublet of the primary particle

`J3Line particleAxis_J3Line_center`
Geometry of primary particle track in the earth center coordinate.
`J3Line particleAxis_J3Line_ice3`
Geometry of primary particle track in the IceCube coordinate.
`J3Vector startLocation_J3Vector_ice3`
Location where the propagation starts. Defined in the IceCube Coordinate
`J3Vector startLocation_J3Vector_center`
Location where the propagation starts. Defined in the earth center Coordinate
`J3Vector endLocation_J3Vector_ice3`
Location where the propagation ends. Defined in the IceCube Coordinate
`J3Vector shift_J3Vector_ice3`
Geometry Shift (if exists) of the particle trajectory. Defined in the IceCube Coordinate

`IceCubeCoordinate ice3Coordinate` The IceCube coordinate.
`EarthCenterCoordinate ice3Coordinate` The Earth Center coordinate.

`IceCubeVolume ice3vol` The IceCube volume.
`Volume ice3OuterVol` The outer volume. This sets the boundary outside which tracing the propagation particle is put to an end because the particle is too far away from the detector.

`double startLocation` Started location of incident particle
`int propagationFlag` Flag on the propagation start point.
`MonteCarloBase[] mcBases` Array of the MonteCarloBase objects
`InteractionsMatrix[] intMtx` Array of the InteractionsMatrix objects

`int numOfDecay` Parameter for counting registered decay channels
`boolean mudecay` Flag for Mu decay
`boolean taudecay` Flag for Tau decay
`int tauDecayFlag` Flag for Tau decay
`int electronBaseFlag` Flag for ElectronBase
`int mTauDecay` Where does the Tau decay registered in the `mcBases[]`.
`MuDecayBase muDecayBase` MudecayBase object
`TauDecayBase tauDecayBase` TaudecayBase object

`Event event` Event object
`int primaryiLogE` Index of bin of the log primary energy

`String interactionsMatrixDirectoryInIce` Directory path for the dumped InteractionsMatrix
`String interactionsMatrixDirectoryInRock` Directory path for the dumped InteractionsMatrix
`String interactionsMatrixDirectory` Directory path (Ice or Rock)

`RandomGenerator rand` Random generator defined in numRecipes

`List particleList`
Store the particle secondary produced during the propagation of primary particles forming cascades.The relevant particle object is stored.
`ListIterator particleIterator` Iterator of trackParticleList.
`List locationIce3List`
Store the location where the secondary particle is produced. J3Vector representing this location in the IceCube coordinate is stored.
`ListIterator lLocationIce3Iterator` Iterator of locationIce3List.

`List trackParticleList`
Store the particle forming the track including the primary particle. The relevant particle object is stored. When the primary particle changed, say, by its decay, then new particle is also stored in this list.
`ListIterator trackParticleIterator` Iterator of trackParticleList.
`List trackLocationIce3List`
Store the location the primary track starts in form of J3Vector. When the primary particle changed, say, by its decay, then location where the track of new particle starts is also stored in this list.
`ListIterator trackLocationIce3Iterator` Iterator of trackLocationIce3List.
`int dim` Dimension of matrix defined in Particle class.

Constructor : `public JulietEventGenerator() throws IOException`
The parameters, e.g. number of events, flavor/doublet/energy of incident particle, format of the output data(See Captor3) etc., are registered by users interactively. The kind of medium is input by users so that the method can generate the ParticlePoint object. The RandomGenerator object is generated and it will be used for Monte Carlo method. The objects including dumped InteractionsMatrix objects, Mu/TauDecayBase objects and an array of MonteCarloBase objects, `mcBases[]`, which elements are generated by users interactively and generated in this constructor. Note that if the Charged current interaction of ν_e is registered, the ElectronBase objects are also generated (See 6.17) and registered. The way of inputting the parameters initialized in this constructor is described

in chapter3.

Constructor : `public JulietEventGenerator(int flavorID, int doubletID,...) throws IOException`

Same as above, but setting the primary particle and selection of the interaction channels are made by given ID valuables rather by the interactive way.

`public void configureJULIeT()`
Initialize primary particle, generate interactionsMatrix and generate IceCube geometry with coordinate systems. The Monte Carlo base matrix (Section 6.16) is generated for each of the interactions Matrices. Primary particle information can be updated after the configuration, while others are fixed here. Called by constructors.

`public void setStartLocationAlongTheAxis(double start)`
`public double getStartLocationAlongTheAxis()`
Set/get the location of primary particle along the J3Line axis of the primary particle track where the propagation starts. "location" is given by the axis length of J3Line, which also sets the initial axis length of the ParticlePoint objects to follow the propagating particle along the track.

`public J3Vector wherePrimaryParticleStartsInEarthCenterCoordinate()`
`public J3Vector wherePrimaryParticleStartsInIceCubeCoordinate()`
Returns the location of the vector where the particle starts its propagation defined by the relevant coordinate system.

`public J3UnitVector getPrimaryParticleDirectionInEarthCenterCoordinate()`
`public J3UnitVector getPrimaryParticleDirectionInIceCubeCoordinate()`
Returns the unit vector of direction of the primary particle defined by the relevant coordinate system.

`public void setGeometryShift(J3Vector shift)`
`public J3Vector getGeometryShift()`
Set/get the geometrical shift in IceCube coordinate.

`public void definePropagatingParticle(int flavor, int doublet, double energy)`
Set the particle that propagates. It is set by the constructor and you do not have to call this method in most of the case unless you need to initiate a different particle.

`public void definePropagationGeometry`
`(double x_ice3, double y_ice3, double z_ice3, double nx, double ny, double nz)`
Set the primary particle track geometry. Defined by the IceCube coordinate system.

`public void configurePropagationGeometry()`
Configure the particle propagation geometry.

`public void runSingleEvent()`
This is the method to run a single event with a monochromatic primary energy given by the constructor. This method works as written below:

1) The `propParticle` is generated as primary particle.
2) The `point` is generated with a given `materialNumber`.
3) The `event` is generated with `mcBases[]` registered in constructor.
4) The `pathLength` is sampled by `event.getPhysicalPathLength()`.
   This time, `mcBaseInPlay` is determined in event object.
5) Run particle by `pathLength` by calling moveParticleAxis().
6) If the propagating particle is still in your DETECTOR, the particle tracking is continued.
   This is judged by the method of ParticleTracker class(Section 10.4).
7) The `event.collideNow()` is called to make the interaction/decay took place.
8) According to the reaction type that just occurred, the new propagating particle is stored in the lists trackParticleList and trackLocationIce3List. The particles that souled be stored are

neutrino produced by the NC interaction, and muon and tau by decay processes, the CC/NC interaction and the Glashow resonance.

9) When the electron or hadron is produced, it supposedly initiates the electromagnetic/hadronic cascade. Information on the secondary particle and the location is stored in the lists particleList and locationIce3List.

10) The log-energy of the `propParticle` after the interaction should be greater than the minimum defined in `InteractionsBase`. When it is smaller than the minimum, the particle has to stop because (almost) all the primary energy has been lost. If the NC interaction occurred, the energy instead of the log-energy is checked, since the minimum energy of the `propParticle` calculated in the `collideNow()` is the mass of the particle which is massless neutrino for the NC interaction. Additionally, when the propagating particle generates an electron or a hadron via leptonic/hadronic tau decay, propagation ends because electron/hadron produces cascade and deposit all its energy.

11) Repeat 1)∼9) until the particle stops or changes into electron/hadron or passes through your DETECTOR, i.e. `trajectoryLength`.

`public void moveParticleAxis(double depth)`

Move the location of the primary particle along its track by depth g/cm^2. The interaction/decay length is given by the Event objects in unit of g/cm^2 while the geometry is defined by the physical length [cm]. This method, therefore, make conversions by calling getMediumDensity() of the Particle Point object (Section 4.1).

`public ListIterator getParticleIterator(), getLocationIce3Iterator(),..`

All these methods provide the iterator of the relevant lists to read the objects such as Particle or J3Vector stored in runSingleEvent().

`public static void setNeutrinoInteractionWeight(int weight)`

Set the neutrino interaction weight. The cross sections involving neutrino collisions are enhanced by a factor of *weight*. This mechanism is introduced for generating neutrino MC events in effective way. This method must be called **BEFORE runSingleEvent()** for simulating neutrino propagation.

11.4 RunManager.java

This generates events using the event class for simple geometry, but calculates the matricis that contains energy distribution of the secondary cascades. Although now the JulietEventGenerator is the one you should use for generating the MC events, this class used to be a main class responsible for event generation, and may be still useful becuase of its capability of generating the energy matrix. See Chapter 15 for details.

The parameters provided in this class are:

`static final double ln10` Common logarithm
`static final double epsilon` Round off error
`static final int neutrinoFactor` Factor introduced to save CPU(See 6.16)

`double[][] emgCascadeMtx` Matrix for integral electromagnetic cascade energy
`double[][] hadronCascadeMtx` Matrix for integral hadronic cascade energy
`double[][] totalCascadeMtx` Matrix for integral cascade energy
`double[][] oneEmgCascadeMtx` Matrix for differential electromagnetic cascade energy
`double[][] oneHadronCascadeMtx` Matrix for differential hadronic cascade energy
`double[][] oneTotalCascadeMtx` Matrix for differential cascade energy

`int numberOfEmgCascade` Number of electromagnetic cascades
`int numberOfHadronCascade` Number of hadronic cascades
`int numberOfTotalCascade` Number of both of ele-mag and hadronic cascades.
`int[] numberOfCascade` Number of cascades by each interaction/decay
`double[] energyOfCascade` Array for storing cascade energy of each interaction/decay

`ParticlePoint point` ParticlePoint object
`int materialNumber` Ice→0, Rock→1

`Particle propParticle` Particle object which indicates the particle propagating
`double primaryEnergy` Initial value of the propagating particle's energy
`static int primaryFlavor` Flavor of the primary particle
`static int primaryDoublet` Doublet of the primary particle
`static double trajectoryLength` Total Distance of the particle path[cm]
`double startLocation` Started location of incident particle
`MonteCarloBase[] mcBases` Array of the MonteCarloBase objects
`InteractionsMatrix[] intMtx` Array of the InteractionsMatrix objects

`int numOfDecay` Palameter for counting registered decay channels
`boolean mudecay` Flag for Mu decay
`boolean taudecay` Flag for Tau decay
`int tauDecayFlag` Flag for Tau decay
`int electronBaseFlag` Flag for ElectronBase
`int mTauDecay` Where does the Tau decay registered in the `mcBases[]`.
`MuDecayBase muDecayBase` MudecayBase object
`TauDecayBase tauDecayBase` TaudecayBase object

`Event event` Event object
`int primaryiLogE` Index of bin of the log primary energy

`String interactionsMatrixDirectoryInIce` Directory path for the dumped InteractionsMatrix
`String interactionsMatrixDirectoryInRock` Directory path for the dumped InteractionsMatrix
`String interactionsMatrixDirectory` Directory path (Ice or Rock)

`RandomGenerator rand` Random generator defined in numRecipes

`static int numberOfEvent` Run parameter to decide number of events
`static boolean integral` Flag:true→whole cascade energy while particles'journey
`static boolean differential` Flag:true→differential cascade energy while particles'journey
`static boolean eachInteraction` Flag:true→produce data for each channel respectively
`static String fileName` Name of output file
`static int typeOfEvent` Type of event determin which run method is called.
`int dim` Dimension of matrix defined in Particle class.

Constructor : `public RunManager() throws IOException`

The parameters, e.g. number of events, flavor/doublet/energy of incident particle, format of the output data(See Capter3) etc., are registered by users interactively. The kind of medium is input by users so that the method can generate the ParticlePoint object. The RandomGenerator object is generated and it will be used for MonteCarlo method. The objects including dumped InteractionsMatrix objects, Mu/TauDecayBase objects and an array of MonteCarloBase objects, `mcBases[]`, which elements are generated by users interactively and generated in this constructor. Note that if the Charged current interaction of ν_e is registered, the ElectronBase objects are also generated (See 6.17) and registered. The way of inputting the parameters initialized in this constructor is discribed in chapter3.

```
public void setStartLocation(double start)
public double getStartLocation()
```

Set/get the location of primary particle.

```
public void runEventDump(DataOutputStream out) throws IOException
```

This is the method to run events for making full-data-file which is textual data. This is mainly for debugging. All the profile of the propagation of particles are dumped in the file. The incident of primary particle is repeated `numberOfEvent` times. Note that the data are written *after* an interaction occured. This method traces the particle till it stops or reaches the end point.

`public void runSingleEvent()`

This is the method to run a single event with a monocromatic primary energy given by the constructor. It is called by the `runMultipleEvent_Each()`, `runMultipleEvent()` and `runEventOnMatrix()`. Therefore it does not have output method.

This method works as wrriten below:

1) The `propParticle` is geneated as primary particle.
2) The `point` is geneated with a given `materialNumber`. This is a starting point. Therefore `setStartLocation()` is called in order to store starting location.
3) The `event` is geneated with `mcBases[]` registered in constructor.
4) The `pathLength` is sampled by `event.getPhysicalPathLength()`. This time, `mcBaseInPlay` is determined in event object.
5) Run particle by `pathLength`.
6) If the propagating particle is still in your DETECTOR, calculations is continued.
7) The `event.collideNow()` is called, so the interaction/decay has just occured.
8) When the electron or hadron is produced, it initiates the electromagnetic/hadronic cascade. The index of log-energy of one cascade(=produced log-energy), i.e. `jLogE` is calculated and stored by `setOneEmgCascadeMtx()` etc.
9) The log-energy of the `propParticle` after the interaction should be greater than the minimum defined in `InteractionsBase`. When it is smaller than the minimum, the particle has to stop because (almost)all the primary energy has been lost. Additionary, when the particle changed into electron/hadron via Tau to electron/hadron decay, particle has to also stop because electron/hadron produces cascade and deposit all its energy.
10) Repeat 1)$\sim$9) until the particle stopps or changes into electron/hadron or passes throuth your DETECTOR, i.e. `trajectoryLength`.

`public void runMultipleEvent_Each(DataOutputStream out) throws IOException`

This is the method to run multiple events with a monocromatic primary energy given by the constructor. This method can make data of each interaction/decay respectively.

This method works as wrriten below:

1) Determine the `length` which is the row-dimension of matrices.
2) Generate the matrices for storing data.
3) The `propParticle` is geneated as primary particle.
4) The `point` is geneated with given materialNumber. This is a starting point. Therefore `setStartLocation()` is called in order to store starting location.
5) The `event` is geneated with `mcBases[]` registered in constructor.
6) The `pathLength` is sampled by `event.getPhysicalPathLength()`. This time, `mcBaseInPlay` is determined in event object.
7) Trace particle with `pathLength`.
8) If the propagating particle is still in your DETECTOR, calculations is continued.
9) The `event.collideNow()` is called, so the interaction/decay has just occured.
10) Search that where is the `event.mcBaseInPlay` in the `mcBases[]`.
11) If the `event.mcBaseInPlay` is Tau decay, ele-mag/hadronic cascade can be produced. Therefore the `m` is changed into `length-2`(emg) or `length-1`(hadronic).
12) When the electron or hadron is produced it produces the ele-mag/hadronic cascade, the index of log-energy of one cascade(=produced log-energy), i.e. `jLogE` is calculated and stored by `setOneEmgCascadeMtx(), energyOfCascade[]` etc.
13) The log-energy of the `propParticle` after the interaction should to be greater than the minimum defined in `InteractionsBase`. When it is smaller than the minimum, the particle has to stop because (almost)all the primary energy has been lost. Additionary, when the particle changed into electron/hadron via Tau to electron/hadron decay, particle has

to also stop because electron/hadron produces cascade and deposit all its energy.
14) Repeat 1)∼3) `numberOfEvent` times.
15) Make the `totalCascadeMtx[][]` for every interaction/decay.
16) Normalize and write out matrices to given DataOutputStream, `out`.
17) Check the interactions/decays registered in the `mcBases[]`.

`public void setOneCascadeMtx(int n, int jLogE)`

The 1.0 is added to the element of the `oneTotalCascadeMtx[][]`. The `n` indicates which interaction/decay produced the cascade, `jLogE` indicetes the index of log cascade energy. It also counts the number of cascade in order to use normalization.

`public void runMultipleEvent(DataOutputStream out) throws IOException`

This is the method to run multiple events with a monocromatic primary energy given by the constructor. It makes data of electromagnetic and hadronic cascade energy respectively.
This method works as wrriten below:

1) Generate the matrices for storing data, but the matrices has only one row because of monocromatic primary energy.
2) Call `runSingleEvent()`.
3) Make matrices for whole energy which particles deposit while their journey. The index of log of cascade energy i.e. `jLogE` is calculated and stored to `emgCascadeMtx[][]` etc. calculated and stored to `emgCascadeMtx[][]` etc.
4) Repeat 1)∼3) `numberOfEvent` times.
5) Normalized the elements of the matrices for whole energy deposit. The normalization factor is `numberOfEvent`. Note that when the primary particle is neutrino, `neutrinoFactor` is multiplied to the normalization factor.
6) Normalized the elements of the matrices for differential energy deposit. The normalization factor is `numberOfEmgCascade` etc.
7) Write out matrices to given DataOutputStream, `out`.

`public void runEventOnMatrix(DataOutputStream out) throws IOException`

This is the method to run multiple events with various primary energies in allowed region[1]. The results are bynary data stored in the matrix form.
This method works as wrriten below:

1) Generate the matrices for storing data.
2) Set the primary energy.
3) Call `runSingleEvent()`.
4) Make matrices for whole energy which particles deposit while their journey. The index of log of cascade energy i.e. `jLogE` is calculated and stored to `emgCascadeMtx[][]` etc. calculated and stored to `emgCascadeMtx[][]` etc.
5) Repeat 1)∼3) `numberOfEvent` times.
6) Normalized the elements of the matrices for whole energy deposit. The normalization factor is `numberOfEvent`. Note that when the primary particle is neutrino, `neutrinoFactor` is multiplied to the normalization factor.
7) Normalized the elements of the matrices for differential energy deposit. The normalization factor is `numberOfEmgCascade` etc.
8) Repeat 2)∼6) `dim` times for each primary `iLogE`.
9) Write out matrices to given DataOutputStream, `out`.

`public void setOneEmgCascadeMtx(int iLogE, int jLogE)`
`public void setOneHadronCascadeMtx(int iLogE, int jLogE)`
`public void setOneTotalCascadeMtx(int iLogE, int jLogE)`

The 1.0 is added to the element of the `oneEmgCascadeMtx[][]` etc. The `iLogE` indicates the index of log primary energy, `jLogE` indicetes the index of log cascade energy. It also counts the number of cascade in order to use normalization.

[1] In the default, 10^5[GeV]∼10^{12}[GeV].

`public static void main(String[] args) throws IOException`

This is the main method. RunManager object and DataOutputStream are generated. The main method calls one of the run-method depending on some parameters a user inputs.

Chapter 12

Package : iceCube.uhe.neutrinoModel

This package contains the classes for calculation of the neutrino flux.
The contained classes are:

- NeutrinoFlux
- DrawNeutrinoFlux
- DrawNeutrinoFluxIntegral
- DrawAngularNeutrinoFlux
- PropagatingNeutrinoFlux
- DrawPropagatingNeutrinoFlux

These package have program with main method to make prots by "grafig":Draw~.java. It can be a reference to learn how the neutrinoFlux and ProgatainfNeutrinoFlux objects work.

12.1 NeutrinoFlux.java

This class contains the methods for calculating UHE neutrino fluxes based on the following models (See Table12.1). The UHE neutrino fluxes calculated in this class indicate the fluxes reach the surface of the earth. Note that *no neutrino oscilation* was considered in these models. You can account the effects later in your own program.

The tables to contain the flux data is stored in j̃avalib/classes/icecube/uhe/neutrinoModel which are read out by this class.

The parameters provided in this class are:

`private String dataFile` Name of data file
`private int numberOfFlavor` Number of neutrino flavors in the model
`private double[][] logEArray` Array of log-energy
`private double[][] EFluxArray` Array of flux
`private static final double ln10` Common logarithm
`private int dataNumber` Number of data

Constructor : `public NeutrinoFlux(int model) throws IOException`

Reads out from the table corresponding to given model-parameter, `model`, and the data are stored in `logEArray[][]` and `EFluxArray[][]`.

model-parameter		m	Z_{max}	γ	Emax [ZeV]
1	The GZK Neutrinos	0	2		
2	The GZK Neutrinos	2	2		
3	The GZK Neutrinos	2	4		
4	The GZK Neutrinos	4	4		
5	The GZK Neutrinos	4	5	1.5	
6	The GZK Neutrinos	7	5	1.5	
7	The Z-burst				
8	The Top Down (SUSY)				
9	The Top Down (QCD)				
10	The GZK Neutrinos (by Sigl)	3	2	1	100
11	The GZK Neutrinos (by Sigl)	5	2	2	10

Table 12.1: UHE Neutrino Models. The model-parameter is index of the models. Model-1∼6 are GZK Neutrinos [5], Model-7 is Z-burst model [6], Model-8,9 are Top Down models [7], Model-10,11 are GZK Neutrinos [8]. The 'm' indicates evolution parameter, the 'Z_{max}' indicates "turn-on time" and the 'γ' . They are parameters which individualize the models. Note that there is no ν_τ in GZK model.

`public double getEFlux(double logEnergy, int particleID)`

Calculate the differential Energy Flux [GeV cm^{-2} sec^{-1} sr^{-1}] by interpolating the table read out in the constructor.

`public double getDFDLogE(double logEnergy, int particleID)`

Calculate the log differential Flux dF/dLogE [cm^{-2} sec^{-1} sr^{-1}] by interpolating the table read out in the constructor. $\ln 10/energy$ is multiplied because

$$\begin{aligned}\frac{dF}{dlogE} &= \frac{dF}{dE}\frac{dE}{dlogE} \\ &= E\ln 10\frac{dF}{dE} \\ &= \frac{\ln 10}{E}\left(E^2\frac{dF}{dE}\right),\end{aligned}$$

where the $\left(E^2\frac{dF}{dE}\right)$ is the differential energy flux given by the tables.

The DrawNeutrinoFlux.java is the program for drawing plots using the xfig-grafig tools, you have to write your own interface program for graphics if you want to use your favorite graphics application softwares, however. The DrawNeutrinoFluxIntegral.java is also the program for drawing plots which data is integrated over incident angle, therefore you can see energy distribution. The DrawAngularNeutrinoFlux.java is also the program for drawing plots, howeber, which data is integrated over energy, therefore you can see anglar distribution.

12.2 PropagatingNeutrinoFlux.java

This class contains the methods for calculating differential fluxes of neutrinos and charged leptons which reach the surface of the IceCube detector after propagate in the earth. The primary flux of UHE cosmic neutrinos is calculated in NeutrinoFlux class. The transfer(propagation) matrix of the propagation in the earth, is read out from the file generated a priori by PropagationMatrix.java in the propagation package. Full mixing of ν oscillation is assumed and made proper adjustment in the primary flux of UHE cosmic neutrinos.

The parameters provided in this class are:

`static int dimension` Dimension of matrices defined in Particle class

`boolean includeGlashowResonance = true`

Flag:false→Calculate the differential fluxes with old propagation matrix not including the Glashow resonance.
You can not read the old matrix by default because the old matrix have smaller dimension due to the additional channels by the Glashow resonance.

`double[][] FnuEToNuE etc.` Matrices for propagation matrix
`private List neutFluxList` List of NeutrinoFlux object
`private List modelNumberList` List of model parameter
`private static final double ln10` Common logarithm

Constructor : `public PropagatingNeutrinoFlux(int model) throws IOException`
Generate the `FnuEToNuE[][]` etc. Lists for the NeutrinoFlux object, `neutFluxList`, and model parameter, `modelNumberList`, are also generated. The given model parameter, `model` and NeutrinoFlux object corresponding to the model are added to the lists.

`private void generateMatrix()`
Generate the `FnuEToNuE[][]` etc.

`protected void generateNeutrinoFluxObject(int model) throws IOException`
Add a given model parameter and the NeutrinoFlux object for the model to lists, `modelNumberList` and `neutFluxList`.

`public void readMatrix(DataInputStream in) throws IOException`
Read the calculated propagation matrix. The transfer(propagation) matrix of the propagation in the earth, for example, `FnuMuToTau[][]` $= \frac{dN_(\nu_\mu \to \tau)}{dLogE_\tau(E_{\nu_\mu, E_\tau})}$ is read out from the file generated a priori by PropagationMatrix.java.

`protected NeutrinoFlux getNeutrinoFluxFromList(int model)`
Get NeutrinoFlux object corresponding to a given model parameter.

```
public double getDFNuEDLogE(double logE)
public double getDFNuEDLogE(int jLogE)
public double getDFNuEDLogE(int model, double logE)
public double getDFNuEDLogE(int model, int jLogE)
```
Calculate dF/dLogE [cm^{-2} sec^{-1} sr^{-1}] corresponding to a given model parameter, `model`, for producing ν_e. The propagation matrix is multiplied to the differential flux for ν_e to ν_e, ν_μ to ν_e and ν_τ to ν_e respectively.
For example, the $\frac{dF}{dlogE}$ for ν_e to ν_μ is given by

```
count += 0.5*neutFlux.getDFDLogE(logE,2)*FnuMuToNuE[iLogE][jLogE];
count += 0.5*neutFlux.getDFDLogE(logE,3)*FnuMuToNuE[iLogE][jLogE];
```

where `0.5*neutFlux.getDFDLogE(logE,2)` in the upper line means incident cosmic ν_{mu} calculated in NeutrinoFlux.java has reduced by a half because of the *neutrino oscillation*. By the same reason, `0.5*neutFlux.getDFDLogE(logE,3)` in the lower line means a half amount of incident cosmic ν_{tau} has became ν_{mu}.

```
public double getDFNuMuDLogE(double logE)
public double getDFNuMuDLogE(int jLogE)
public double getDFNuMuDLogE(int model, double logE)
public double getDFNuMuDLogE(int model, int jLogE)
```
Calculate dF/dLogE [cm^{-2} sec^{-1} sr^{-1}] for ν_μ by the similar method as above.

```
public double getDFNuTauDLogE(double logE)
public double getDFNuTauDLogE(int jLogE)
```

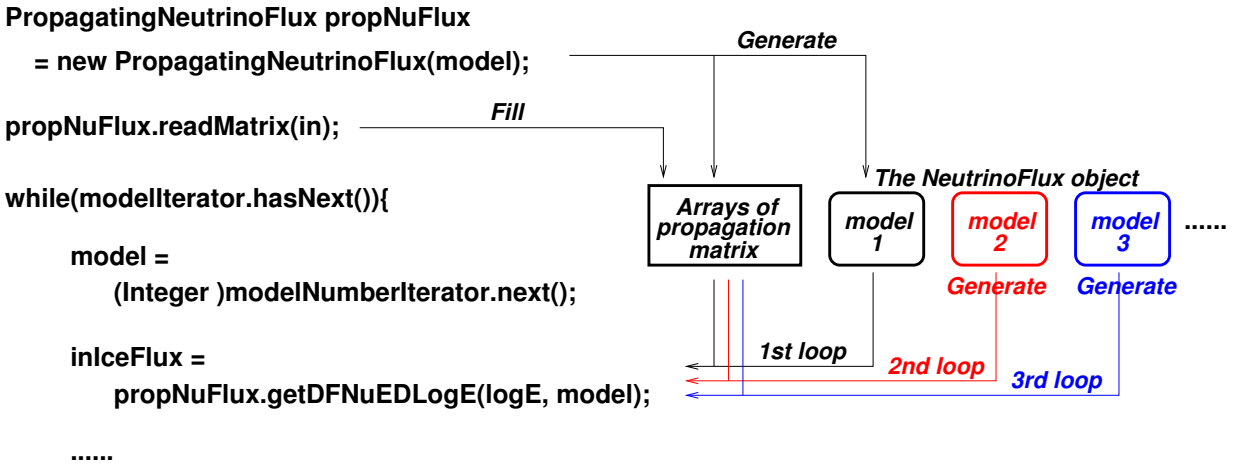

Figure 12.1: Analysis flow to calculate fluxes of various neutrino models at once. This is an example for electron-neutrino with energy of `logE` at the IceCube depth.

`public double getDFNuTauDLogE(int model, double logE)`
`public double getDFNuTauDLogE(int model, int jLogE)`
Calculate dF/dLogE [cm^{-2} sec^{-1} sr^{-1}] for ν_τ by the similar method as above.

`public double getDFMuDLogE(double logE)`
`public double getDFMuDLogE(int jLogE)`
`public double getDFMuDLogE(int model, double logE)`
`public double getDFMuDLogE(int model, int jLogE)`
Calculate dF/dLogE [cm^{-2} sec^{-1} sr^{-1}] for μ by the similar method as above.

`public double getDFTauDLogE(double logE)`
`public double getDFTauDLogE(int jLogE)`
`public double getDFTauDLogE(int model, double logE)`
`public double getDFTauDLogE(int model, int jLogE)`
Calculate dF/dLogE [cm^{-2} sec^{-1} sr^{-1}] for τ by the similar method as above.

`public void whetherPropagationMatrixWithGlashowResonance(boolean flag)`
Change the flag, `includeGlashowResonance`, to calculate differential flux with or without the Glashow resonance.

The DrawPropagatingNeutrinoFlux.java is the program for drawing plots using the xfig-grafig tools, you have to write your own interface program for graphics if you want to use your favorite graphics application softwares, however.

Figure 12.1 shows analysis flow to calculate fluxes of various neutrino models once it reads out the relevant matrix data file. The constructor of the `PropagatingNeutrinoFlux` generates arrays of propagation matrix and lists of model parameter and the `NeutrinoFlux` object for a given model. The `readMatrix` reads the relevant matrix data file and fill the matrix arrays. This process takes much time. The `getDFNuEDLogE` etc. calculates a neutrino flux at the IceCube depth for a given model and energy. If the `modelNumberList` does not have the model parameter according to a given model, the `NeutrinoFlux` object for the model is generated and added to the `neutFluxList`.

Chapter 13

Package : iceCube.uhe.muonModel

This package contains the classes for calculation of the atmospheric muon flux.
The contained classes are:

- ParticleFlux
- CosmicRayFlux
- AtmMuonBundleFlux
- AtmMuonFluxCorsika
- CascadeFluctuationFactory
- PropagatingAtmMuonFlux
- DrawPropagatingAtmMuonFlux.java
- DrawPropagatingAtmMuonBundleFlux.java
- DrawAngularAtmMuonFlux.java

The "Draw*.java" are the application to plot the calculated flux by "grafig" plotting package, but also supplies good examples to show the useage of the key objects like PropagatingAtmMuonFlux.

13.1 ParticleFlux.java

This is the abstract class concerning particle fluxes. The abstract methods,

```
abstract double getDFDLogE(double logEnergy, double cosTheta)

abstract double getEFlux(double logEnergy, double cosTheta)
```

are implemented in the inheritated subclasses, CosmicRayFlux, AtmMuonBundleFlux, and AtmMuonFluxCorsika.

13.2 CosmicRayFlux.java

CosmicRayFlux is responsible for calculating primary cosmic ray flux at the earth surface. The flux is represented by the power law formula $\sim E^{-\gamma}$ and the power law index and the intensity is taken from the compilation by Nagano and Watson (2000) using the AGASA, Haverah Park, Yakutsuk, and Haverah Park experiments.

The GZK cutoff feature is also implemented by calling the method setCutOffFeature(boolean cutoffExists).

Constructor : `public CosmicRayFlux()`
Calculate the base numbers to give the differential flux.

`public double getEFlux(double logEnergy)`
Calculate the differential Energy Flux [GeV cm^{-2} sec^{-1} sr^{-1}] based on the power law formula.

`public double getDFDLogE(double logEnergy)`
Calculate the log differential Flux dF/dLogE [cm^{-2} sec^{-1} sr^{-1}] based on the power law formula.

`public void setCutOffFeature(boolean cutoffExists)`
When cutoffExists sets true, cosmic ray flux is not extended beyond the energy defined by `private double logEHighestBound`.

13.3 AtmMuonBundleFlux.java

AtmMuonBundleFlux is responsible for calculating atmospheric muon flux at the earth surface, the main background in search for ultra-high energy cosmic neutrino signals.

The muon flux in the high energy range is mainly determined by muon bundle intensity originated in hadronic interactions in EAS cascade. This class relies on the empirical formula based on the Elbert model (See Cosmic Rays and Particle Physics by T.Gaisser) to calculate the flux. The free parameters appeared in the formula has the default values determined by the IceCube 2006 analysis in the present version.

13.3.1 The Elbert model and its representation in the AtmMuonBundle class

The Elbert model gives number of muons in a single EAS cascade by

$$\begin{aligned} N_\mu &= \frac{E_T}{E_0}\frac{A^2}{\cos\theta}\xi^{-\alpha}(1-\xi)^\beta \\ \xi &= \frac{AE_\mu}{E_0} \end{aligned} \tag{13.1}$$

where A is the mass number of primary cosmic rays with energy of E_0, θ is zenith angle of a muon bundle. The other parameters are $E_T = 14.5$ GeV, α=1.757, β=5.25. One can differentiate it to obtain the differential spectrum of muons in a bundle dN_μ/dE_μ.

Total energy summing over all muons contained in the bundle is effectively equivalent to the bundle energy as follows:

$$E_\mu^B \equiv \int_{E_{th}}^{E_0/A} \frac{dN_\mu}{dE_\mu} E_\mu dE_\mu. \tag{13.2}$$

Here E_{th} is the threshold energy of muons contributing to the bundle. The approximation using the fact that the integration above is mainly determined in the regime of $\xi \ll 1$ leads to the relatively simple formula to represent the bundle energy as

$$E_\mu^B = E_T \frac{A}{\cos\theta}\frac{\alpha}{\alpha-1}\left(\frac{AE_{th}}{E_0}\right)^{-\alpha+1}. \tag{13.3}$$

It implies that the muon bundle energy is projected to energy of primary energy cosmic ray. Because the cosmic ray flux is given by CosmicRayFlux class (Sec. 13.2), the AtmMuonBundle class calculates the rate of muon "bundle" as

$$\frac{dJ_\mu^B}{dE_\mu^B} = \frac{dE_0}{dE_\mu^B}\frac{dJ_{CR}}{dE_0}(E_0(E_\mu^B, \cos\theta, A, E_{th})). \tag{13.4}$$

The two parameters in Eqs. 13.4 and 13.3 are settable by calling the relevant methods in the AtmMuonBundle class.

13.3.2 Functions of AtmMuonBundleFlux class

The two classes CosmicRayFlux and CascadeFluctuationFactory are two major internal members. The CosmicRayFlux gives the intensity of primary cosmic rays in calculation of the muon rate by Eq. 13.4. CascadeFluctuationFactory (Sec. 13.4) takes care of the fluctuation in the hadronic interaction. The Elbert relation described by Eq. 13.3 implies the one-on-one relation between the muon bundle and the cosmic ray energy. In reality this is not true because the energy of muons in the bundles varies from one collision to another in the early hadronic interactions in the EAS cascade process. The problem is that we have only very poor knowledges of the hadronic interactions in the ultra-high energy region. To solve this problem in practical way, the distribution of the *ratio* of the dimensionless energy of the muon bundle

$$R = \frac{\left(\frac{E_\mu^B}{E_0}\right)}{\left(\frac{\bar{E_\mu^B}}{E_0}\right)}, \tag{13.5}$$

the deviation of the dimensionless energy from mean. The CascadeFluctuationFactory class gives the R-distribution based on the Corsika-QGSJET I simulation for protons. The AtmMuonBundleFlux class convolute the R value with the Elbert formula and perform the relevant calculations to obtain the muon bundle flux.

There are two set-methods involving CosmicRayFlux and CascadeFluctuationFactory.

`public void setCutOffFeature(boolean cutoffExists)`

Assumes the spectrum with the GZK cutoff in calculating the muon bundle flux when cutoffExists $= true$.

`public void setFluxCalculationMode(boolean includeFluctEffects, boolean fluctEventByEvent`

If includeFluctEffects= *true* (default), the R-distribution is convoluted in calculation of the atmospheric muon flux. if $false$, it simply gives you the bare Elbert model flux *i.e.,* case of $R = 1$ only. When fluctEventByEvent = $false$ (default), The R effect is taken into account with convoluting its distribution. *true*, R is sampled in even-by-event basis with the Monte Carlo method as taking the R-distribution as a *probability* distribution.

The other methods are similar with the class NeutrinoFlux, except the few methods to set α and E_{th} in Eq. 13.3. Below is the summary of the main important methods. Refer the API document or the source code for all the methods implemented in this class.

Constructor : `public AtmMuonBundleFlux()`

Use the default values of α and E_{th} determined by the IceCube 2006 data.

Constructor : `public AtmMuonBundleFlux(double alpha, double m, double E)`

Use α, A and E_{th} given in the argument for calculating the flux by Eq. 13.3.

`public double getEFlux(double logEnergy, double cosTheta)`

Calculate the differential Energy Flux [GeV cm^{-2} sec^{-1} sr^{-1}] as a function of muon bundle energy at the earth surface and cosine of zenith angle.

`public double getDFDLogE(double logEnergy, double cosTheta)`
Calculate the log differential Flux dF/dLogE [cm^{-2} sec^{-1} sr^{-1}] as a function of muon bundle energy at the earth surface and cosine of zenith angle.

`public double getDFDLogE(double logEnergy, double cosTheta, double beta, double depth)`
The flux is calculated as a function of muon bundle energy at the depth **after the propagation in the earth**. This task is done by PropagatingAtmMuonBundleFlux class more accurately using the matrix data, but this method provides flux in handy way
with the CEL approximation (See Sec. 6.19). beta is the inelasticity parameter and depth is the slant depth [$cm^2/g/GeV$].

`public double getDFDLogE(double logEnergy, double cosTheta, double depth)`
Same as the function above, but CELbeta class (Sec. 6.19) is used for inelasticity parameter.

`public double getEffectiveEnergyOfCRs(double E, double cosTheta)`
Returns E_0, the cosmic ray energy that would produce muon bundle energy E at the earth surface with cosine of the zenith angle. The calculation is done by inverting the Eq. 13.3.

Examples to show how to use this class are found at DrawAtmMuonBundleFlux.java.

13.4 CascadeFluctuationFactory.java

This class is responsible for the fluctuation factor of the bundle energy R given by Eq. 13.5. The distribution of R is obtained by running Corsika-QGCJET simulation and fitted with the log-normal distribution as a function of cosmic ray energy. Then the R-distribution is given by the method getProbability().

`public double getProbability(double logR, double logPrimaryEnergy, boolean asInIce)`
Returns the probability of the cosmic ray with $\log_{10}$(Energy [GeV]) = logPrimaryEnergy giving R concerning the muon bundle energy. If asInIce= *true*, the bundle energy at the IceCube depth is involved. If false, the muon energy at the earth surface is considered.

13.5 PropagatingAtmMuonFlux.java

This class contains the methods for calculating differential fluxes of atmospheric muon bundles which reach the surface of the IceCube detector after propagate in the earth. This is very similar with the class PropagatingNeutrinoFlux (Sec. 12.2), but AtmMuonBundle object instead of NeutrinoFlux is responsible for the primary flux at the earth surface and the propagation matrix data is then multiplied to give the flux after the propagation. The methods are, thus, similar with PropagatingNeutrinoFlux except some methods directly related to AtmMuonBundle class.

The parameters provided in this class are:

`static int dimension` Dimension of matrices defined in Particle class

`boolean includeGlashowResonance = true`
Flag:false→Calculate the differential fluxes with old propagation matrix not including the Glashow resonance.
You can not read the old matrix by default because the old matrix have smaller dimension due to the additional channels by the Glashow resonance.

`double[][] FnuEToNuE etc.` Matrices for propagation matrix

`AtmMuonBunfleFlux atmMuFlux` AtmMuonBundleFlux object

`double muonThresholdEnergy` Threshold energy of muons contributing the bundle.

Constructor : `public PropagatingAtmMuonFlux(double alpha,double muE,boolean cutoffExists)`

Generate the `FnuEToNuE[][]` etc. The AtmMuonBundleFlux object is also generated corresponding to the parameters α, E_{th} (See Sec 13.3). The GZK feature is assumed when cutoffExists = *true*.

`public void readMatrix(DataInputStream in) throws IOException`

Read the calculated propagation matrix. The transfer(propagation) matrix of the propagation in the earth, for example, `FnuMuToTau[][]` $= \frac{dN_(\nu_\mu \rightarrow \tau)}{dLogE_\tau(E_{\nu_\mu, E_\tau})}$ is read out from the file generated a priori by PropagationMatrix.java.

`public double getDFMuDLogE(double logE, double cosZenith)`

Calculate dF/dLogE [cm^{-2} sec^{-1} sr^{-1}] for producing muon bundles with $\log_{10}$(Energy [GeV])= logE after the propagation in the earth. The propagation matrix is multiplied to the differential flux for μ to μ

`public void setCutOffFeature(boolean cutoffExists)`

Assumes the spectrum with the GZK cutoff in calculating the muon bundle flux when cutoffExists $= true$.

`public void setFluxCalculationMode(boolean includeFluctEffects, boolean fluctEventByEvent`

If includeFluctEffects= $true$ (default), the R-distribution is convoluted in calculation of the atmospheric muon flux. if $false$, it simply gives you the bare Elbert model flux *i.e.,* case of $R = 1$ only. When fluctEventByEvent = $false$ (default), The R effect is taken into account with convoluting its distribution. $true$, R is sampled in even-by-event basis with the Monte Carlo method as taking the R-distribution as a *probability* distribution.

`public void whetherPropagationMatrixWithGlashowResonance(boolean flag)`

Change the flag, `includeGlashowResonance`, to calculate differential flux with or without the Glashow resonance.

This class together with PropagatingNeutrinoFlux class are extensively used by the weighting module in the IceTray-based IceCube analysis. The juliet-interface project provides the C++ interface to invoke these classes and calculate the flux at the IceCube detector depth.

Chapter 14

Package : iceCube.uhe.analysis

This package, newly added from JULIeT version3, provides analysis tools for the IceCube data. An simulation/real event is stored in I3Particle dataclass (Sec: 5.2) and the analysis tools class are to process the I3Particle events. This package allows you to analyze the JULIeT events/IceCube real events in a handy way without relying on complicated C++ -based framework.

The main functions in the analysis package are:

1. Making Histograms and plot them. The relevant classes are
   - I3ParticleAnalysisFactory
   - Criteria
   - EnergyHistogramMaker
2. Calculate the flux sensitivity and the event rate by putting the proper weights into an I3Particle event. The major classes responsible for this function are
   - I3ParticleWeightFiller
   - I3ParticlePropMatrixFiller
   - I3PartcleFlux

The Histogram and plot functions are provided from the JAIDA - Java implementation of the Abstract Interfaces for the Data Analysis, a part of the FreeHEP library. You can download the JAIDA class file from the FreeHEP website, http://java.freehep.org, but they are included in a ready-to-use manner in the JULIeT class jar file available at http://www.ppl.phys.chiba-u.jp/JULIeT/

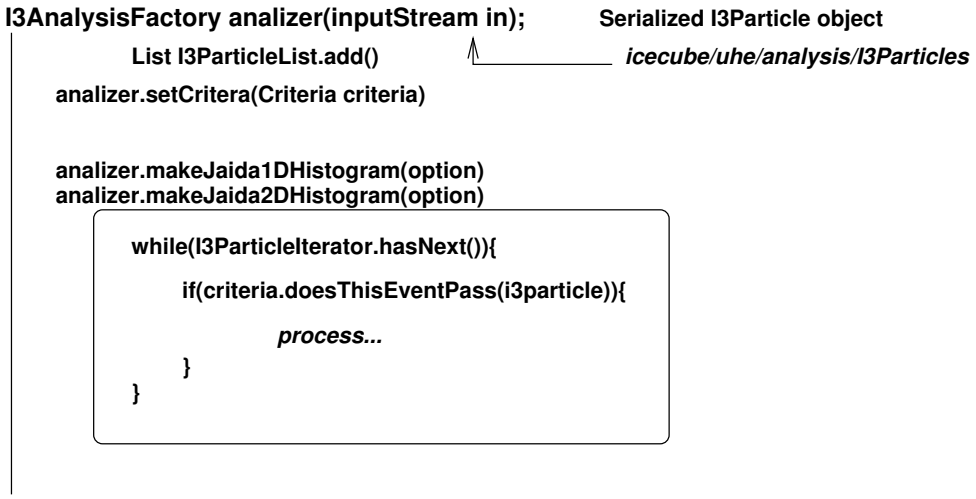

Figure 14.1: Analysis flow using I3Particle. I3ParticleAnalysisFactory gives a representative example to process I3Particle.

14.1 How to compile the java files using JAIDA

The JAIDA 3.3.3-02 or higher is the default version required by the analysis package. All the JAIDA jar files should be located under the directory java_lib/classes/ (java_lib is the name of your JULIeT directory in this example). This distribution can be realized by either

1. unpacking jaida-*version number*-bin.tar.gz and jaida-*version number*-i386-Linux-g++.tar.gz at java_lib/classes/. You can download them from the FreeHEP website.
2. download the JULIeT class jar ball from the JULIeT website and unpack them at java_lib directory.

Then follow the procedure below:

1. set the $JAIDA_HOME to your JAIDA directory - something like java_lib/classes/jaida-3.3.0-2.
2. set the $CLASSPATH to your class directory - like java_lib/classes.
3. source $JAIDA_HOME/bin/aida-setup.csh (for cshell users) aida-setup.sh (for bash users). It sets your environment valuables.
4. Then compile. For example,
   (a) cd java_lib/sources/iceCube/uhe/analysis
   (b) javac -sourcepath ~/java_lib/sources -d ~/java_lib/classes I3ParticleAnalysisFactory.java

   You find the I3ParticleAnalysisFactory.class in java_lib/classes/iceCube/uhe/analysis

14.2 Analysis with JAIDA

Many source codes in the analysis package includes sizable comments. Reading the source or its API document gives you a good gateway to learn the analysis process introduced here. We, therefore, briefly describe the analysis flow and algorithm here rather than explaining an individual class.

A good example to start is RunJaidaAnalysis.java, which runs the I3ParticleAnalysisFactory and plot the histograms. Figure 14.1 shows the flow. The factory reads the I3Particle object files from the disk. Many implementations in the other codes assumes classes/iceCube/uhe/analysis directory to locate the files. Then

I3ParticleAnalysisFactory builds List to contain each of I3Particle. In a method to process I3Particle objects (makeJaida1/2DHistogram() in this example) loops over I3Particle list and see if a given I3Particle event satisfies the criteria defined by Criteria class, which is also a member in the analysis package. If yes, then it fills JaidaHistogram.

This flow is seen in any method to process I3Particle in the other class as well, such as I3ParticleFlux. It is suggested to read the source of the class listed in the beginning of this chapter.

14.3 Analysis with weighting method

In most cases, generated I3Particle events by JULIeT follows the power law energy distribution, but the neutrino flux in nature would not. It is, thus, necessary to put the appropriate weights into an I3Particle in order to calculate the effective area, the event rate and so on. This task are carried out by I3ParticleMCPrimaryWeightFiller, I3ParticleWeightFiller and I3ParticlePropMatrixFiller classes in the analysis package. I3Particle class has a double field or map to store these weights. The corresponding set-methods in I3Particle called from these classes are

I3Particle.setMCPrimarySpectrumWeight(double flux)

I3Particle.setAtmosphercMuonFlux(double flux, String fluxName)

I3Particle.setGZKNeutrinoFlux(double flux, String fluxName)

I3Particle.putLogEnergyMatrix(int iLogE, double matrixElement)

Let us explain about the philosophy of the weighting method. Expected event rate N_{ev} from a primary neutrino flux J_ν at the earth surface can be obtained by

$$\frac{dN_{ev}}{d\log E_l d\Omega} = A^{eff}(\log E_l, \Omega) \int\limits_{\log E_l} d\log E_\nu \frac{dN_{\nu\to l}}{d\log E_l}(\log E_\nu, \log E_l, \Omega)\frac{dJ_\nu}{d\log E_\nu d\Omega}(\log E_\nu, \Omega), \tag{14.1}$$

where E_l is energy of the secondary lepton such as μ, E_ν is primary energy of neutrinos, $N_{\nu\to l}$ is number of secondary leptons produced inside the earth and reaching to the IceCube volume, and A^{eff} is the effective area of the IceCube observatory. The integral in the equation above accounts for the propagation effect in the earth and obtained by resolving the relevant transport equation. This is equivalent to the secondary lepton flux at the IceCube depth and pre-calculated by JULIeT class like PropagatingNeutrinoFlux.java (Sec. 12.2) or PropagatingAtmMuonFlux.java (Sec. 13.5).

The effective area A^{eff} in Eq. 14.1 can be estimated by either the semi-analytical way or the full-brown MC. The semi-analytical method gives A^{eff} as

$$A^{eff}(\log E_l, \Omega) = \int\limits_{\log E^{th}_{dep}\simeq 10\mathrm{PeV}} d\log E_{dep}\frac{dN_{dep}}{d\log E_{dep}}(\log E_l, \log E_{dep}, \Omega)A_0 \tag{14.2}$$

where E_{dep} is energy deposit of the secondary lepton propagating over 1km inside the IceCube volume. In EHE the energy deposit takes place mostly in form of a bunch of cascades. A_0 is typically 1 km^2 for the IceCube.

By the full MC, the effective area will be given by

$$\begin{aligned} A^{eff}(\log E_l, \Omega) &= A_0 \frac{N^{detected}}{\frac{dN_l^{MC}}{d\log E_l}(\log E_l, \Omega)\Delta\log E_l} \\ &= A_0 \sum^{detected} \frac{1}{\frac{dN_l^{MC}}{d\log E_l}(\log E_l, \Omega)\Delta\log E_l}, \end{aligned} \tag{14.3}$$

where $N^{detected}$ is number of MC events passing your criteria, $dN_l^{MC}/d\log E_l$ is the MC primary particle spectrum of the secondary leptons (mostly μ and τ for EHE). A_0 is the area of throwing primary particles in MC. $\Delta\log E_l$ is a bin width. Putting Equations [14.1] and [14.3] together will lead to the proper event weight as

$$w = A_0 \frac{1}{\frac{dN_l^{MC}}{d\log E_l}(\log E_l, \Omega)\Delta\log E_l} \int_{\log E_l} d\log E_\nu \frac{dN_{\nu\to l}}{d\log E_l}(\log E_\nu, \log E_l, \Omega)\frac{dJ_\nu}{d\log E_\nu d\Omega}. \tag{14.4}$$

Note that J_ν could be given by the GZK neutrino model, or any other EHE neutrino models as well as by the atmospheric muon/neutrino background prediction. So the weight given above could be not unique but various depending on what models you consider. $dN_l^{MC}/d\log E_l$ is filled for an individual I3Particle by I3ParticleMCPrimaryWeightFiller.java. The integral part in Eq. 14.4 is put by I3ParticleWeightFiller.java using PropagatingNeutrinoFlux and PropagatingAtmMuonFlux objects. The pre-calculated propagation matrix data must be installed in the way described in Sec. 8.5. You can download them from the JULIeT website.

In this context the IceCube "sensitivity" for EHE neutrinos can be obtained from a quasi-differential event rate in neutrino model independent way. This approach has been widely used in many other experiments. The neutrino flux upper-bound with energy of E_0 from non-existence of signals is evaluated by putting

$$\frac{dJ_\nu}{d\log E_\nu d\Omega}(\log E_\nu, \Omega) = \left[\frac{dJ_\nu}{d\log E_\nu d\Omega}(\log E_0, \Omega)\right] \delta(\log E_\nu - \log E_0) \tag{14.5}$$

into Eq. 14.1. We get

$$\left[\frac{dJ_\nu}{d\log E_\nu d\Omega}(\log E_0, \Omega)\right] = \frac{N^{ev}(\geq \log E_l^{th})^{95\%}}{\int d\Omega \int_{\log E_l^{th}} d\log E_l A^{eff}(\log E_l, \Omega)\frac{dN_{\nu\to l}}{d\log E_l}(\log E_0, \log E_l, \Omega)} \tag{14.6}$$

where E_l^{th} is threshold energy of the secondary leptons, $N^{ev}(\geq \log E_l^{th})^{95\%}$ is number of upper bound events with 95 % C.L. The Poisson distribution gives 2.3 events for example. I3ParticleFlux class calculates this flux by reading out the serialized I3Particle object in the disk. This task is run by RunI3ParticleFlux.java. You find the way I3ParticleFlux process I3Particle is similar with the one shown in Fig. 14.1 - it utilizes Criteria objects that applies to each of I3Particle stored in the object list.

The I3ParticleFlux requires I3Particle objects filled with the propagation matrix $dN_{\nu\to l}/d\log E_l(\log E_0, \log E_l, \Omega)$ appeared in Eq. 14.6. This filling is carried out by I3ParticlePropMatrixFiller class in the analysis package and must be done *before* running I3ParticleFlux.

14.4 A note to the IceTray users : How to build I3Particle

The IceTray (The IceCube offline software framework) users simulates JULIeT events via *juliet-interface* project and the resultant IceCube MC data is stored in i3 format. You have to build I3Particle object from i3 files to run the analysis programs. Once this transfer is done, your analysis environment is completely free from the IceTray.

The easiest way to make this transformation is to run I3EHEEventDigest.cxx in the *Ophelia* project. It prints out the I3 event summary to the standard output. An example python script is DigestEHEEvents.py. Put "true" for "digestJulietParticle". Then connect its output to RunI3ParticleBuilder.java which runs I3ParticleBuilder class to build I3Particle from the I3EHEEventDigest output.

Chapter 15

How To Run The Simulation by RunManager.java

In the previous version of JULIeT, **RunManager.java** in event package is a tool to run JULIeT instead of **JulietEventGenerator.java** and its example of the main method **RunJuliet.java**, which is a current standard for running particles with MonteCarlo method. This **RunManager.java** was the origin for **JulietEventGenerator.java** and did not involve any geometry class, but it has its own features which can be of use for debugging and fundamenthal study of UHE particle propagation. Especially it has a capability of saving particle energy distribution in many forms like matrix whick makes it easy to see secondary particle energies. In fact most of the section IV in the PRD paper (S. Yoshida et. al., *Phys. Rev.* **D 69** 103004 (2004)) was written using this capability. This chapter is to describe about **RunManager.java** which was Chapter 3 in the previous manual.

RunManager.java in the event package runs the particles and calculates their energy profile with the Monte-Carlo method. The method using the propagation package, which has advantages in calulation of particles propagating in long distances, are described in Chapter 8.

RunManager.java runs partcles in an interactive manner. Some examples using the interactive mode are provided in the following.

Below is an interactions list considered in this simulation. Users can add/remove the individual interaction/decay to/from this list.

- Charged current interaction
- Neutral current interaction
- Bremsstrahlung
- Knock-on Electron
- Pair Creation to e/mu/tau pair
- Photo-nuclear Interaction
- Glashow resonance
- Decay

Allowed regions: Energy region of primary particles from 10^6[GeV] to 10^{12}[GeV]
Energy region of produced particles from 10^2[GeV] to 10^{12}[GeV]

15.1 Photo-Nuclear Interaction

The JULIeT in current version uses two types of cross section of Photo-Nuclear interaction. One of them is calculated by ALLM parameterization [2, 3]. Another is calculated by Bezrukov-Bugaev(BB) parameterization with hard part [4]. When you run the JULIeT, you should

- in /java_lib/classes/iceCube/uhe/classes(and souces)/, copy(overwrite) one of the class(and source) files,
  ~ALLM/PhotoNuclear.class and ~BB/PhotoNuclear.class
- in /java_lib/classes/iceCube/uhe/interactions/ice(and rock)/, copy(overwrite) one of the pre-calculated InteractionsMatrix, ~ALLM/muPhotoNuclearMtx(and tauPhotoNuclearMtx) and ~BB/muPhotoNuclearMtx(and tauPhotoNuclearMtx)
- in /java_lib/data/, remake the symbolic link by "`ln -s propMtx/ALLM(or BB)/ neutrino_earth`".

15.2 Types of the Output Formula

There are three types of output formula available in form of the methods supplied by the RunManager.java. There is a function to draw plots using the xfig-grafig tools, you have to write your own interface program for graphics if you want to use your favorite graphics application softwares, however.

15.2.1 Dump Full data

This is one of the output methods to run events for making full-text-based data. This is mainly for debugging. All the profile of the propagation of particles are dumped in the file. Users can input monocromatic primary energy. This mathod traces the particle till it stops or reaches the end point. The generated data is written in the ASCII format. The outputs include following data.

- Name of interaction
- Primary particle's energy
- Transfered energy
- Distance the primary particle gained

Note that all data are written *after* an interaction occured. The JULIeT traces the primary particle till it stops or passes throuth your DETECTOR. If the primary particle is τ, and if tau decay is registered, mu-interactions must be registered too.

Example of output

```
Colliding via PairCreation from Muon to Electron producing Electron
Particle Energy 9.999978228687162E8 [GeV]
Transfered Energy 2177.131283799154 [GeV]
Current distance the particle gained 449.02808319270446 [cm]
```

Example of run

at ~/java_lib/classes/

```
% java -Xms128m -Xmx256m iceCube.uhe.event.RunManager
```

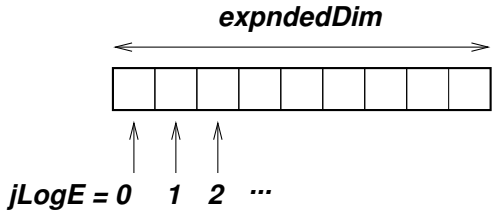

Figure 15.1: Schematic view of produced data. The 'expandedDim' is the dimension of array defined in InteractionsBase class(See 6.16). The 'jLogE' is the index of produced energy bin.

```
Number of Event -> 100            Input number of primary particles you run
Type of Event(0:single energy(full data) 1:single energy 2:multi energy(Matrix))->0
Name of File -> test.dat
Particle Flavor -> 2             e:0 mu:1 tau:2
Particle Doublet -> 1            nutrino:0 charged lepton:1
Particle Energy [GeV] -> 1.0e10
Medium in Propagation ice(0) rock(1) -> 0
Random Generator has been generated
Charged Current Interactions yes(1)/no(0)?->0
Neutral Current Interactions yes(1)/no(0)?->0
Muon Bremsstrahlung yes(1)/no(0)?->1
Tau Bremsstrahlung yes(1)/no(0)?->1
Muon Knock-on Electrons yes(1)/no(0)?->1
Tau Knock-on Electrons yes(1)/no(0)?->1
Muon e+e- Pair Creation yes(1)/no(0)?->1
Tau e+e- Pair Creation yes(1)/no(0)?->1
Muon mu+mu- Pair Creation yes(1)/no(0)?->1
Tau mu+mu- Pair Creation yes(1)/no(0)?->1
Muon tau+tau- Pair Creation yes(1)/no(0)?->1
Tau tau+tau- Pair Creation yes(1)/no(0)?->1
Muon Photo-nuclear interactions yes(1)/no(0)?->1
Tau Photo-nuclear interactions yes(1)/no(0)?->1
Glashow Resonance yes(1)/no(0)?->1
Mu decay yes(1)/no(0)?->1
Tau Decay yes(1)/no(0)?->1
...
```

15.2.2 Dump Single Energy data (Array)

This method of output produces bynary (array-formed) data (See Fig.15.1). In this method, you can choose 2 types of outputs in terms of the energy recording. One is to make a distribution of total energy deposit during the primary particle's journey. The another makes distribution of differential energy deposit, i.e. energy deposit per one collision. These outputs include following data.

- Distribution of energy deposit to electromagnetic cascade (log-scale)
- Distribution of energy deposit to hadronic cascade (log-scale)
- Distribution of total energy deposit (log-scale)

Note that when electron was produced, JULIeT considers the electron's energy as the primary energy of an electromagnetic cascade (and if hadron was produced, as the primary energy of a hadronic cascade).

For both differential and total(integral) energy deposit, you can group the energy deposit distribution as a function of interaction channels. Therefore, output is something like

- Distribution of energy deposit by Bremsstrahlung (log-scale)
- Distribution of energy deposit by Knock-on Electron (log-scale)

...

Example of run

at –/java_lib/classes/

```
% java -Xms128m -Xmx256m iceCube.uhe.event.RunManager

Number of Event -> 100            Input how many primary particles you run
Type of Event(0:single energy(full data) 1:single energy 2:multi energy(Matrix))->1
Each Interaction Data yes(1)/no(0)?  -> 0       If you input 1, register all interaction/decay
Type of Matrix(0:EnergyDeposit/1km 1:EnergyDeposit/cascade) -> 0   1 means total energy
deposit is recorded
Name of File -> test.dat
Particle Flavor -> 2        e:0 mu:1 tau:2
Particle Doublet -> 1       nutrino:0 lepton:1
Particle Energy [GeV] -> 1.0e10
Medium in Propagation ice(0) rock(1) -> 0
Random Generator has been generated
Charged Current Interactions yes(1)/no(0)?->0
Neutral Current Interactions yes(1)/no(0)?->0
Muon Bremsstrahlung yes(1)/no(0)?->1
Tau Bremsstrahlung yes(1)/no(0)?->1
...
Mu decay yes(1)/no(0)?->1
Tau Decay yes(1)/no(0)?->1
...
```

how to make plots by grafig

at –/java_lib/classes/

make a picture of each interaction data – No

```
%java -classpath ~/java_lib/classes/ iceCube.uhe.event.DrawEventMatrix_single test.dat 10
```

10 is logPrimaryEnergy i.e. 10^{10}[GeV]

make a picture of each interaction data – Yes

```
%java -classpath ~/java_lib/classes/ iceCube.uhe.event.DrawEventArrayForEachInteraction test.dat
n
```

primary flavor is e or mu:n=0, tau:n=1

All those source codes are found in the relevant source directory and you can compile them by javac with the ususal flag of "`-claspath ~/java_lib/classes/ -d ~/java_lib/classes`" (See Section 2.1).

Even if you are not a user of xfig-grafig package, you are suggested to take a look at these Draw*.java codes in order to write your own application for making plots.

15.2.3 Dump Multiple Energy data (Matrix)

This method basically runs the method described above but for various primary energies so that you can obtain the energy distribution as a function of primary energy of incident particles. The generated

data is also stored in bynary (matrix-based) format. The outputs include folloing data (for each primary energy).

- Distribution of energy deposit to electromagnetic cascade (log-scale)
- Distribution of energy deposit to hadronic cascade (log-scale)
- Distribution of total energy deposit (log-scale)

Note that if electron was produced, JULIeT considers the electron's energy as the primary energy of an electromagnetic cascade (and if hadron was produced, as the primary energy of a hadronic cascade).

In this method, users can choose 2 types of outputs, distribution of total (integral) energy deposit or differential energy deposit, i.e. energy deposit per one cascade.

It scans primary energies of incident particles automaticaly in the range from 10^6[GeV] to 10^{12}[GeV] with steps of 0.01 decade of logE [GeV]. The results are sorted in form of the matrix and saved in the binary file (See Fig.15.2).

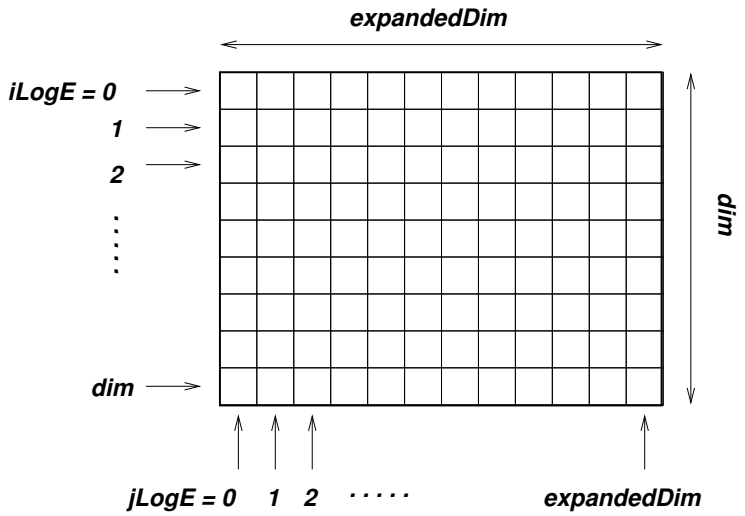

Figure 15.2: Schematic view of produced data. This is the dim-expandedDim matrix, i.e. Data-Matrix[dim][expandedDim]. The 'dim' indecates number of bins of incident energy, and the 'expandedDim' indecates one of produced energy. The 'dim' and 'expandedDim' are defined in InteractionsBase(See 6.16). The 'iLogE' and 'jLogE' are the index of incident/produced energy bin. The relation between 'iLogE' and energy is defined by Particle.java in the particle package (See Chapter 5).

Example of run:

at /java_lib/classes/

```
% java -Xms128m -Xmx256m iceCube.uhe.event.RunManager

Number of Event -> 100        Input how many primary particles you run
Type of Event(0:single energy(full data) 1:single energy 2:multi energy(Matrix))->2
Type of Matrix(0:EnrgyDeposit/1km 1:EnergyDeposit/cascade) -> 0  Input type of energy
deposit
Name of File -> test.dat
```

```
Particle Flavor -> 2          e:0 mu:1 tau:2
Particle Doublet -> 1       nutrino:0 lepton:1
Medium in Propagation ice(0) rock(1) -> 0
Random Generator has been generated
Charged Current Interactions yes(1)/no(0)?->0
Neutral Current Interactions yes(1)/no(0)?->0
Muon Bremsstrahlung yes(1)/no(0)?->1
Tau Bremsstrahlung yes(1)/no(0)?->1
...
Mu decay yes(1)/no(0)?->1
Tau Decay yes(1)/no(0)?->1
...
```

how to make a plot by grafig

at /java_lib/classes/

```
% java -claspath ~/java_lib/classes/ iceCube.uhe.event.DrawEventMatrix test.dat 10
```

10 is primary log energy i.e. 10^{10}[GeV]

Even if you are not a user of xfig-grafig package, you are suggested to take a look at these Draw*.java codes in order to write your own application for making plots.

Bibliography

[1] S. Yoshida, R. Ishibashi and H. Miyamoto, astro-ph/0312078, accepted for publication in Phys. Rev. D **69**, (2004).

[2] S. I. Dutta, M. H. Reno, I. Sarcevic and D. Seckel, Phys. Rev. D **63**, 094020, (2001).

[3] H. Abramowicz and A. Levy, hep-ph/9712415, (1997).

[4] E. Bugaev, T. Montaruli, Y. Shlepin and I. Sokalski, hep-ph/0312295, (2003).

[5] S. Yoshida *et al.*, The Astrophysical Journal, **479**, 547-559, 1997.

[6] S. Yoshida, G. Sigl and S. Lee, Phys.Rev.Lett., **81**, 5505, 1998.

[7] G. Sigl *et al.*, Phys.Rev.Lett., **D 59**, 043504, 1998.

[8] O. E. Kalahsev, G. Sigl *et al.*, Phys.Rev.Lett., **D 66**, 063004, 2002.