The FluxReader Framework

Gareth Kafka

Department of Physics
Harvard University

August 18, 2014

Abstract

This technical note details the FluxReader package. The framework and its purpose are first introduced. Next, the steps necessary to set up and run the framework are discussed. Finally, the internal mechanics and code to FluxReader are described. This note is intended for developers and contains information unnecessary to general users. It is current as of revision 5.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

FluxReader is a framework designed to read flux files and create plots from their contents. It is specifically optimized to run over Dk2Nu files [1] and create large numbers of similar histograms simultaneously and quickly. The design philosophy for FluxReader is to remain as general and flexible as possible. More opaque code is hidden within behind the scenes classes and functions. The result is that the end user only needs to use simple and easy to use member functions to create powerful macros quickly. These macros can generate huge assortments of plots efficiently using a very small amount of user configuration.

Dk2Nu files consist of a TTree with each entry containing information pertaining to a neutrino ray [2]. Single histograms can be created quickly using the TTree::Draw method, even allowing for the application of cuts and weights. However, studies often involve making the many plots of the same variable with slightly different cuts, like distributions of parent p_T vs p_z for each neutrino parent species and daughter neutrino flavor. This alone can generate close to 30 histograms. The TTree::Draw method becomes very inefficient in these cases, and FluxReader is designed for these purposes.

The FluxReader framework is independent of any single experiment. This note describes how to access and build the framework, gives a detailed account of how the framework works, and provides some details on building and running macros. In Chapter 2, a valid kerberos

ticket is assumed. The details of how FluxReader works are discussed in Chapter 3. This chapter is intended for users that have a good knowledge of the C++ language and is written at a level necessary for developers.

This note is current as of FluxReader revision 5 [3], Dk2Nu revision 81 [4], and ROOT version 5.34 [5]. Individual files may not always be cited directly, but if not, they can all be found in one of these repositories/reference guides.

Chapter 2

Framework Setup

Since FluxReader is independent of any single experiment, it exists in a repository on its own. This chapter describes how to access the framework, what is done to set up and compile the framework, and how to run macros once the framework is built. A valid kerberos ticket is assumed for the remainder of this chapter (and for the remainder of this note, in general).

2.1 Access and Permissions

FluxReader is a subproject of NuUtils and exists in its own redmine project [6]. In order to use the framework, the code must be checked out into a user's local area. There are two commands which can do this, one for general users, and a different one for developers. Checking out the code is discussed in greater detail on the FluxReader Wiki [7].

All users must first decide where FluxReader will live in their local areas. It is recommended to put FluxReader in a user's top level directory. For experiments with tagged software releases, the framework can be put into a development release. If this is done, setting up both the experiment environment and the FluxReader environment may cause warnings to occur, as both may try to set the same environment variables or external products. This is covered in more detail in Section 2.2. It is NOT recommended to put FluxReader in a specific tag. This is likely to cause errors.

Once the user has decided where FluxReader will exist, ensures the directory exists, and moves to that directory, the code can be checked out. For general users, the command is:

```
> svn co http://cdcvs.fnal.gov/subversion/fluxreader/trunk/FluxReader
```

This will check out a copy of FluxReader into the current directory. By checking out the code with this command, a user can make local changes. However, these changes cannot be commit to the FluxReader repository.

For developers, the code must be checked out differently.

```
> svn checkout svn+ssh://p-fluxreader@cdcvs.fnal.gov/cvs/projects/\
> fluxreader/trunk/FluxReader
```

Note that this is shown here for readability; it can (should) be entered on one line (and if so, omit the trailing '\' from the first line). Checking out the code in this fashion allows the user (a developer) to make local changes and commit those changes back to the repository. Anyone can *attempt* to run this command; however, it will result in an error (Permission Denied) for users not listed as a developer on the redmine overview page [6].

2.2 Setup

In order to use FluxReader, a user must source the setup script, SetupFluxReader.sh by running the following command.

```
> source SetupFluxReader.sh
```

This script sets up several ups products, including ROOT, Dk2Nu, and genie_xsec. It also sets up cmake, which is used to build the framework, and the environment variable FLUXREADER_PRIV, which is used to load the FluxReader library when running ROOT macros. It is recommended to set up FluxReader in a fresh terminal session without setting up anything tied to a particular experiment.

Instead of blindly setting up any of the above products, the script checks if any of their associated environment variables are set. If they are, then the product is already set up, and the script will warn the user of a possible version conflict. Otherwise, the script sets up the correct version of the product. The version conflict might occur if a user has set up an environment for a particular experiment; this often sets up a particular version of ROOT, for instance. If a user attempts to run FluxReader after receiving the version conflict warnings and subsequently has an error running the framework, the first step in solving the problem would be to start a fresh terminal session and only set up FluxReader so it builds cleanly.

2.3 Compilation

Once FluxReader has been set up as described in Section 2.2, and assuming the user is still in the FluxReader directory, the code can be compiled using the following two commands.

```
> cmake .
> gmake all
```

When the code is first checked out, only the following files will exist in the top level directory (there are other files in the include and src directories).

```
> ls -alFGh
total 44K
drwxr-xr-x 7 gkafka 2.0K Aug 14 10:13 ./
drwxr-xr-x 3 gkafka 2.0K Aug 14 10:13 ../
-rw-r--r-- 1 gkafka 2.4K Aug 14 10:13 CMakeLists.txt
drwxr-xr-x 3 gkafka 2.0K Aug 14 10:13 Demo/
-rw-r--r-- 1 gkafka 1.3K Aug 14 10:13 FluxReaderTemplate.C
drwxr-xr-x 3 gkafka 2.0K Aug 14 10:13 include/
drwxr-xr-x 3 gkafka 2.0K Aug 14 10:13 lib/
-rw-r--r-- 1 gkafka 2.1K Aug 14 10:13 load_flxrd.C
-rwxr-xr-x 1 gkafka  922 Aug 14 10:13 SetupFluxReader.sh*
drwxr-xr-x 3 gkafka 2.0K Aug 14 10:13 src/
drwxr-xr-x 6 gkafka 2.0K Aug 14 10:13 .svn/
```

The cmake command automatically generates a Makefile needed by gmake, as well as several other output and log files. The user should not change any of these files. The gmake

command actually compiles the code. If the code is being built for the first time, the gmake command will result in the library libFluxReader.so appearing in the lib directory. At the top level, the following files will exist after compilation.

```
> ls -alFGh
total 88K
drwxr-xr-x 8 gkafka 2.0K Aug 14 10:16 ./
drwxr-xr-x 3 gkafka 2.0K Aug 14 10:13 ../
-rw-r--r-- 1 gkafka  13K Aug 14 10:16 CMakeCache.txt
drwxr-xr-x 6 gkafka 4.0K Aug 14 10:16 CMakeFiles/
-rw-r--r-- 1 gkafka 2.7K Aug 14 10:16 cmake_install.cmake
-rw-r--r-- 1 gkafka 2.4K Aug 14 10:13 CMakeLists.txt
drwxr-xr-x 3 gkafka 2.0K Aug 14 10:13 Demo/
-rw-r--r-- 1 gkafka 1.3K Aug 14 10:13 FluxReaderTemplate.C
drwxr-xr-x 3 gkafka 2.0K Aug 14 10:13 include/
drwxr-xr-x 3 gkafka 2.0K Aug 14 10:13 lib/
-rw-r--r-- 1 gkafka 2.1K Aug 14 10:13 load_flxrd.C
-rw-r--r-- 1 gkafka  20K Aug 14 10:16 Makefile
-rwxr-xr-x 1 gkafka  922 Aug 14 10:13 SetupFluxReader.sh*
drwxr-xr-x 3 gkafka 2.0K Aug 14 10:13 src/
drwxr-xr-x 6 gkafka 2.0K Aug 14 10:13 .svn/
```

As long as the file Makefile exists, the user can perform a clean build by running the following commands.

```
> gmake clean
> cmake .
> gmake all
```

However, if none of the file dependencies have changed, i.e., there are no new include or src files and all of the include preprocessor directives are the same, then the cmake step can be omitted. If the cmake line is necessary due to different dependencies, one may be required to delete CMakeCache.txt before cmake will run again.

2.4 Running Macros

2.4.1 ROOT Environment Setup

Much like SetupFluxReader.sh sets up the appropriate environment at the terminal, load_flxrd.C sets up the appropriate environment for running macros in ROOT. Essentially, this means specifying include file locations so ROOT can load the necessary libraries. One of the first things that the script does is process a different script, load_dk2nu.C, located in the $DK2NU/scripts directory. That particular script makes sure ROOT can find the appropriate Dk2Nu libraries. The next portion of load_flxrd.C tells ROOT where to find the various header files listed as includes, including ROOT files, Dk2Nu header files, and FluxReader header files. Finally the script loads the FluxReader library, libFluxReader.so.

The script load_flxrd.C is run as an interpreted script; i.e., without any '+' sign after the file name. Any scripts that will actually run FluxReader must be run as compiled scripts; i.e., with a single '+' sign after the file name. Both load_flxrd.C and the compiled script can be given to root as two arguments so long as load_flxrd.C comes first. The example below demonstrates how to run the FluxReader template script, FluxReaderTemplate.C.

```
> root load_flxrd.C FluxReaderTemplate.C+
```

2.4.2 Compiled ROOT Macros

Compiled ROOT macros are what the user executes in order to run FluxReader. There are several required pieces of code that are needed in these macros, and other recommended ones. To discuss these necessities, the code from the template script FluxReaderTemplate.C is reproduced below, with line numbers.

```
1 // This template script includes all of, and nothing more than,
2 // the basic necessities for running FluxReader
3
4 #ifdef __CINT__
5 void FluxReaderTemplate()
```

```
6 {
7   std::cout << "Sorry, you must run in compiled mode." << std::endl;
8 }
9 #else
10
11 // C/C++ Includes
12 #include <iostream>
13 #include <string>
14
15 // ROOT Includes
16 #include "TFile.h"
17
18 // Package Includes
19 #include "Detectors.h"
20 #include "FluxReader.h"
21 #include "Parameters.h"
22 #include "Utilities.h"
23 #include "Vars.h"
24
25 using namespace flxrd;
26
27 void FluxReaderTemplate()
28 {
29   // First, we need to set up a Parameters object
30   Parameters p(false);
31
32   // Add at least one detector
33   p.AddDetector(knova_fd);
34
35   // Next, we'll create a FluxReader object
36   string dk2nu_loc = "/nusoft/data/flux/dk2nu/nova/2010/
                        flugg_mn000z200i_20101117.gpcfgrid_lowth/";
37   dk2nu_loc += "*dk2nu.root";
38   FluxReader *fr = new FluxReader(dk2nu_loc, 2);
39
40   // The FluxReader needs to generate something!
41   fr->AddSpectra(p, "enu", "Energy (GeV)", Bins(100, 0., 10.), kEnergy);
42
43   // The last thing to do before running is to set up an output file
44   TFile* out = new TFile("/nova/ana/users/gkafka/FluxReader/
                           HelloWorld.root", "RECREATE");
45
46   // This function loops over the files, and fills all the histograms!
47   fr->ReadFlux(out);
48   out->Close(); // Close output file
49   delete fr; // Clean up
50 }
```

```
51
52 #endif
```

The code on lines 4-9 and 52 must enclose every compiled macro. They force the macro to run in a compiled mode, as the ROOT C interpreter cannot handle some of the FluxReader classes. The function that appears on line 5 must match the macro name, without the ".C" file extension.

The code on lines 11-23 are all of the include files needed by the macro. All of the files in FluxReader have this grouping of C/C++ files, ROOT files, package files, then other external files, as necessary. Other scripts may need other files, but the ones shown here are generally necessary for all macros. FluxReaderTemplate.C technically does not need the iostream header, but it is generally a good idea for debugging purposes.

Everything in FluxReader exists in the flxrd namespace, so it is a good idea to include line 25. While this is not necessary, omitting this line requires adding the namespace scope to every FluxReader object used in the macro.

Lines 27, 28, and 50 enclose the main macro function. Just like line 5, the function on line 27 must match the macro name without its extension.

A working macro requires a Parameters object with at least one Detector object, and a FluxReader object with at least one Spectra added. The FluxReader object takes a path to input Dk2Nu files as input to its constructor. This is all demonstrated on lines 29-41.

Once the necessary objects have been set up, the function `FluxReader::ReadFlux` can be called with a `TFile*` or `TDirectory*`. This input specifies where the FluxReader output will go, and is demonstrated on lines 43-47. Lines 48 and 49 are some manual clean up.

Chapter 3

How FluxReader Works

This chapter describes how FluxReader works internally. It is written at a level necessary for a developer, and thus contains much more information than is required for an end user. The various classes will be introduced from the bottom up, so if a particular class is dependent on another prerequisite class through inheritance or friendship, the prerequisite class will be detailed first. This means that the motivation for a particular design decision sometimes becomes opaque; in these cases an attempt to describe this motivation is made as best as possible.

Everything within the FluxReader framework is created in the namespace flxrd. Any classes or functions that omit a namespace scope are assumed to be using this flxrd namespace. Any exceptions to this will explicitly show the namespace scope, including any references to classes or functions in the std library. Lastly, all FluxReader header files exist in the include directory, and all FluxReader cxx files exist in the src directory, though these directory names are omitted when mentioned [3].

Some of the FluxReader classes and functions borrowed heavily from comparable ones in the NOνSoft CAFAna framework [8]. Notably, the FluxReader Var and Weight classes were derived from the CAFAna Var and Cut classes, the FluxReader Utilities function Wildcard function comes from the CAFAna Utilities Wildcard function, and the FluxReader class

SetBranches function is derived from the BranchList function in the BrachList class defined in the SpectrumLoaderBase class in CAFAna. The references are current as of revision 10606 of CAFAna.

3.1 Utilities

There are several functions defined within Utilities.h and Utilities.cxx that are used elsewhere within the FluxReader framework. The most important function to the user is the `Bins` function, which takes three arguments.

```
std::vector<double> Bins(int nbins, double min, double max);
```

The Spectra class requires a `std::vector<double>` input for binning. When the user wants to use `nbins` equally sized bins between `min` and `max`, this function quickly converts these three parameters into a `std::vector<double>`.

The `Wildcard` function takes as input a `std::string` that may or may not contain shell style wildcard characters, and returns a `std::vector<std::string>` that expands all of the wildcard characters as the shell would. (This means that if there are no wildcard characters in the input string, then the vector will have length 1.)

```
std::vector<std::string> Wildcard(std::string fileWildcard);
```

This function is intended to allow the user to specify a group of input files by using a single path with wildcard characters; however, the user is never required to use this function itself. Instead, the FluxReader constructor calls this function after the user provides a `std::string` input. On the other hand, the user can access `Wildcard` to check that the files specified are what the user expects.

The last Utility function is the `OverrideAddresses` function. This function is only used when reading non-Dk2Nu files, and is discussed in Section 3.10.

Table 3.1: The Detector class private variables, their type, and the public functions that return each variable. The return functions listed return the variable as the same type as is stored, each taking no inputs.

Detector Spec	Type	Variable Name	Return Function
Name	`std::string`	`fDetName`	`GetDetName()`
Nuclear Target	`std::string`	`fTarget`	`GetTarget()`
Coordinates	`std::vector<double>`	`fCoord`	`GetCoords()`
Size	`std::vector<double>`	`fSize`	`GetSizes()`
Uses	`int`	`fUses`	`GetUses()`

3.2 Detector

The Detector class defined in Detector.h and Detector.cxx stores information about a particular detector. This includes a detector name, the typical nuclear target, the detector coordinates in the detector coordinate system in cm, the detector size in cm, and the number of times to smear a neutrino ray through the detector volume, hereafter referred to as uses. All of these are stored in private variables. Both the detector name and nuclear target are each stored as a `std::string`, the coordinates and size are each stored in a `std::vector<double>` (each with length 3), and the uses is stored as an `int`. With the exception of the uses, each parameter cannot be changed once the Detector object is constructed. Table 3.1 lists each private variable and its associated public member function that can return the stored value. Both the detector coordinates and size can be accessed in multiple ways, and Table 3.2 list them all.

Existing detectors are predefined in Detectors.h, and Detectors.cxx makes sure these objects are built during compilation. However, custom detectors can be easily defined. The Detector constructor takes an argument for each stored variable.

```
Detector(const std::string& det_name, const std::string& target,
         const double& coordx, const double& coordy, const double& coordz,
         const double& sizex, const double& sizey, const double& sizez,
         const int& nuses);
```

As mentioned above, the uses is the only parameter that can be changed after a Detector is constructed.

Table 3.2: All of the different methods for returning the detector coordinates and size.

Detector Spec	Return Function	Return Type
Coordinates	`GetCoords()`	`std::vector<double>`
	`GetCoordX()`	`double`
	`GetCoordY()`	`double`
	`GetCoordZ()`	`double`
	`GetTCoords()`	`TVector3`
Size	`GetSizes()`	`std::vector<double>`
	`GetSizeX()`	`double`
	`GetSizeY()`	`double`
	`GetSizeZ()`	`double`
	`GetHalfSizeX()`	`double`
	`GetHalfSizeY()`	`double`
	`GetHalfSizeZ()`	`double`

```
void SetUses(int nuses);
```

Allowing this parameter to change allows the user to change the default value of 1 in each of the predefined Detectors. However, the Parameters class, which stores a list of Detectors, also has a function to change the uses of a Detector, and users should use this version.

3.3 ParticleParam

The ParticleParam class defined in ParticleParam.h and ParticleParam.cxx contains information about a particle. It stores a PDG value in a private integer, and a name or label in a private std::string. Both of these parameters can be accessed by the publicly accessible functions, GetPDG() and GetName(), respectively.

There are no base ParticleParam objects in the FluxReader framework. Instead, objects are created of two other classes, NuFlav and Parent, both which inherit from ParticleParam, as shown in Figure 3.1, and are defined and implemented in the same files as their common mother.

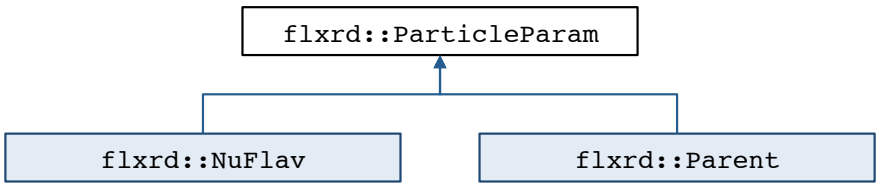

Figure 3.1: The inheritance structure of ParticleParam and its daughter classes, NuFlav and Parent.

3.3.1 NuFlav

The NuFlav class inherits from the ParticleParam class and stores information on neutrino flavors. Predefined NuFlav objects exist for the electron neutrino, anti-electron neutrino, muon neutrino, anti-muon neutrino, tau neutrino, and anti-tau neutrino.

The NuFlav class also introduces functions to manipulate vectors of NuFlavs. The `AllNuFlavs` function creates a `std::vector<NuFlav>` while `RemoveNuFlav` removes entries from a vector. Since a NuFlav stores both a particle name and PDG value, `RemoveNuFlav` can be called using a particle name to remove, a PDG value to remove, or a NuFlav object to remove. No matter which version is used, all NuFlavs that have a matching input parameter will be removed.

```
std::vector<NuFlav> AllNuFlavs(bool SignSensitive);

void RemoveNuFlav(std::vector<NuFlav> &nuflavs, int rmpdg);
void RemoveNuFlav(std::vector<NuFlav> &nuflavs, std::string rmname);
void RemoveNuFlav(std::vector<NuFlav> &nuflavs, const NuFlav& rmflav);
```

The Parameters class uses these functions to create its own list of NuFlavs, so the user does not have to use them directly. The boolean input to `AllNuFlavs` determines whether or not to ignore neutrino sign. By default this option is true, meaning that the function will create a separate entry for each flavor of neutrino and anti-neutrino. However, since the user never directly uses this function, and Parameters does not give this as an option to the user, in practice there will always be a splitting for neutrinos and anti-neutrinos.

Technically, the user can create additional NuFlav objects simply by specifying a particle name and PDG to the NuFlav constructor. However, as will be discussed in the Parameters

class, these objects cannot actually be used.

```
NuFlav(std::string name, int pdg);
```

3.3.2 Parent

The Parent class inherits from the ParticleParam class and stores information on neutrino parent species. Predefined Parent objects exist for the μ^+, μ^-, π^+, π^-, K^+, K^-, and K_L. There are also objects defined for the case that parent sign is ignored, i.e., the μ^+ and μ^- are treated as a single μ. These objects are defined for the μ, π, and K. Note that the K_L is always considered separately from the charged versions.

The Parent class introduces functions to manipulate vectors of Parents, nearly identical to those in the NuFlav class. The `AllParents` function creates a `std::vector<Parent>` while `RemoveParent removes entries from a vector. Just as with \verb`RemoveNuFlav—, there are three versions of `RemoveParent`.

```
std::vector<Parent> AllParents(bool SignSensitive);

void RemoveParent(std::vector<Parent> &parents, int rmpdg);
void RemoveParent(std::vector<Parent> &parents, std::string rmname);
void RemoveParent(std::vector<Parent> &parents, const Parent& rmpar);
```

The Parameters class uses these functions to create its own list of Parents, so the user does not have to use them directly, as was the case with the versions within NuFlav. The boolean input to `AllParents` behaves in the same way as before, and is also true by default for the Parent case. When this option is specified as true or left as default, the Parent objects that populate are all the sign sensitive objects mentioned above. When the input is specified as false, each sign insensitive Parent object replaces the two sign sensitive objects it represents. Unlike the case with NuFlav, the option to consider sign is preserved for the user in the Parameters class.

The user can create other Parent objects simply by specifying a name and PDG for the Parent constructor. The PDG value will need to be unique from any other Parent, however.

This will be discussed in more detail with the Parameters class.

```
Parent(std::string name, int pdg);
```

3.4 XSec

The XSec class defined in XSec.h and XSec.cxx is unique in the FluxReader framework in that it gets its input from files that are not flux files. The reason the product genie_xsec must be set up is because XSec gets its cross section information from a file found within the `$GENIEXSECPATH` directory, and this environment variable is set up when genie_xsec is set up. The file found using this environment variable has a particular structure. In its top level there are only subdirectories, and there is one directory for each combination of neutrino flavor and nuclear target. The naming convention for these directories is nu_<flavor>(_bar)_<Atomic Symbol><Z>. Thus, for electron neutrinos scattering off of hydrogen, the directory name is nu_e_H1, and for anti-muon neutrinos scattering off of iron, the directory name is nu_mu_bar_Fe56. Inside each directory, each cross section is contained as a TGraph* and is named by its process, for example, tot_cc.

When an XSec object is constructed, it takes no arguments.

```
XSec();
```

The constructor calls the publicly accessible `SetXSecFileName` function without specifying its optional argument.

```
void SetXSecFileName(std::string override);
```

This function sets up the private variable `fXSecFileName` with the absolute path to the cross section file. By leaving the optional argument blank, the default case using the `$GENIEXSECPATH` environment variable is used. It is possible for the user to override this option and specify a different file to use after an XSec object has been constructed, though

Table 3.3: Functions to generate cross sections and their input arguments and return type. Neutrino PDG is entered as an int, Target Nuclei and Interaction Process are each entered as an std::string, Cross Section Spline is entered as a TSpline3*, Number of Bins is entered as an int, the Min and Max Bin Values are each entered as a double, and the Pointer to Array of Bin Edges is entered as a double*.

Function	Input Arguments	Return Type
`GetXSec`	Neutrino PDG Target Nucleus and Interaction Process	`TSpline3*`
`GetXSecRatio`	2 Neutrino PDGs 2 Target Nuclei and Interaction Processes	`TSpline3*`
`GetGraph`	Neutrino PDG Target Nucleus and Interaction Process	`TGraph*`
`GetGraphRatio`	2 Neutrino PDGs 2 Target Nuclei and Interaction Processes	`TGraph*`
`GetHist`	Cross Section Spline Number of Bins, Min and Max Bin Values	`TH1*`
`GetHist`	Cross Section Spline Number of Bins, Pointer to Array of Bin Edges	`TH1*`

not recommended. However, this is not an option for generating the cross sections used to fill each Spectra object.

At this point, cross sections can be generated from the file using several different functions. These are listed in Table 3.3. `GetXSec` and `GetXSecRatio` are recommended for general purposes, and this is what the rest of the FluxReader framework uses. However, as the cross sections are originally stored as `TGrpah*` objects, `GetGraph` is actually the base case.

When `GetGraph` is provided a neutrino PDG (`int`), nuclear target (`std::string`), and interaction process (`std::string`), it calls the private helper function `SetXSecGenStr` with these same arguments.

```
void SetXSecGenStr(int pdg, std::string tar, std::string type);
```

This function uses the arguments to set the private variable `fXSecGenStr` with the relevant directory and cross section process in the required format as discussed above. The `TGraph*` is then pulled from the cross section file using the (publicly accessible) `GetXSecFileName`

and `GetXSecGenStr` functions.

```
std::string GetXSecFileName();
std::string GetXSecGenStr();
```

When `GetGraphRatio` is called, it first pulls the two relevant graphs using `GetGraph`. The input arguments to `GetGraphRatio` are two sets of the arguments that are supplied to `GetGraph`, the first set being the numerator graph and the second set being the denominator graph. It then checks that both graphs have the same number of points, and computes the ratio, $y_N(x)/y_D(x)$. There is no check that each point has the same x value.

A `TSpline3*` can be generated simply by providing its constructor with a `TGraph*`. `GetXSec(Ratio)` does this by first creating a graph using `GetGraph(Ratio)`. However, the `TSpline3*` constructor can take optional arguments that constrain the spline's endpoints. There are two optional arguments to both `GetXSec` and `GetXSecRatio` that can set the value of the lefthand, or low energy endpoint, and how this value is approached. See the ROOT documentation for more information.

Both versions of `GetHist` behave in the same way. For each bin in the resultant histogram, the input spline is averaged over the bin. Especially at higher energies, cross sections rise approximately linearly with energy. This means that the area under the spline can be approximated as a trapezoid. Instead of finding the area of one trapezoid covering a full bin, each bin is split into a minimum of ten sections, and the areas of the trapezoid from each section are summed. If a bin covers 1 GeV or less, then the bin is split into ten equal sections. If the the bin covers more than 1 GeV, then starting at the low edge, the bin is split into 0.1 GeV sections until the upper edge is within 0.1 GeV, and the last section covers the remainder of the bin. This is shown in Figure 3.2. The sum of the areas of each trapezoid within the bin is averaged over the full bin width, and the bin content is set to the result. As a note, the trapezoid area is computed by using the `XSecEval` function. This is preferred to the built in `TSpline::Eval(Double_t x)` function because `XSecEval` checks for negative values and automatically sets these results to 0 instead of returning a negative value, since

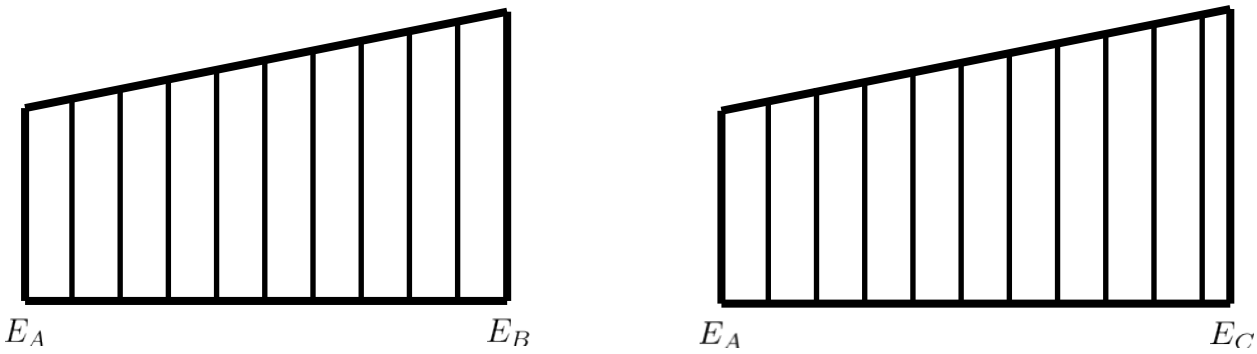

Figure 3.2: The trapezoidal integration scheme used to calculate each bin for a cross section histogram. Left: $E_A < E_B \leq E_A + 1$ GeV, so the bin is split into 10 equal sections. Right: $E_C > E_A + 1$ GeV, so the bin is split into as many 0.1 GeV sections as possible and the last, rightmost section covers the remainder of the bin.

cross sections must always be positive.

```
double XSecEval(TSpline3* s, double x);
```

When a user attempts to create a cross section, XSec automatically checks for valid inputs. The input cross section file only has information for a finite set of neutrino flavors, target nuclei, and interaction process, so if the user provides an input that will not match anything in the cross section file, XSec will not try to create a cross section, but instead provides the user with a list of acceptable inputs if the offending parameter was a neutrino flavor or target nucleus. If the interaction process was invalid, XSec tells the user to use the `ListIntTypes` function. This function outputs the relatively large list of acceptable interaction processes.

```
void ListIntTypes()
```

Three other private functions are used behind the scenes in XSec. The first is `GetGraphMath`.

```
TGraph* GetGraphMath(TGraph* g1, double c1, TGraph* g2, double c2);
```

This function outputs a graph computed from $c1 * g1 + c2 * g2$. One of the acceptable target nuclei is CH_2, which is technically not found in the input cross section file. `GetGraph(Ratio)` recognizes this and recursively pulls the graphs for C and H separately, then merges them with `GetGraphMath`.

The other two functions are `MakeXSecTitle` and `MakeXSecRatioTitle`.

```
std::string MakeXSecTitle(int pdg, std::string tar, std::string type);
std::string MakeXSecRatioTitle(
              int pdg1, std::string tar1, std::string type1
              int pdg2, std::string tar2, std::string type2);
```

These functions are automatically called whenever any cross section plot is created. Nothing glamorous happens inside these functions; they simply combine the input arguments to make human readable plot titles. The ratio version does check whether certain inputs are the same and simplifies the title accordingly.

3.5 Parameters

As mentioned previously, one of the design goals of the FluxReader framework is to create a large amount of similar plots simultaneously. Specifically it makes plots of the same distribution for different sets of neutrino flavors, parent species, applied cross sections, and detector locations. The Parameters class defined in Parameters.h and Parameters.cxx allows the user to configure exactly what sets of parameters are used to generate each set of distribution. It also has an indexing method that supports automatic looping through use of the Indices class, also defined and implemented in Parameters.h and Parameters.cxx.

3.5.1 Indices

The Indices class stores eight private integers. Four represent the total number of each parameter, neutrino flavors, parent species, cross sections to apply, and detectors; these are labeled `nFlav`, `nPar`, `nXSec`, and `nDet`, respectively. The other four integers represent a current index and are restricted to integers in the range $0 \leq$ `i<param>` $<$ `n<param>`, for example, $0 \leq$ `iFlav` $<$ `nFlav`. (One exception of this is discussed below.) Indices has no functions to change the 'n' variables, but Indices makes Parameters a friend class, so it can update those values as necessary.

Indices defines a convention for a single master index combining each individual index. In base 10, the number *dcba* is can be considered $(a\times1)+(b\times10)+(c\times10\times10)+(d\times10\times10\times10)$. By analogy, the Indices convention considered the neutrino flavor like the units digit, the parent species like the tens digit, the cross section as the hundreds digit, and the detector as the thousands digit. However, unlike base 10, the multiplier is no longer a simple power of ten. With `nFlav` neutrino flavors, the multiplier for the parent species is `nFlav`. The multiplier for the cross section becomes `nFlav`×`nPar`, and so on. The full master index formula is shown in Equation 3.1.

$$\begin{aligned} i_{Master} &= i_{Flav} + \\ & \quad n_{Flav} \times i_{Par} + \\ & \quad n_{Flav} \times n_{Par} \times i_{XSec} + \\ & \quad n_{Flav} \times n_{Par} \times n_{XSec} \times i_{Det} \end{aligned} \tag{3.1}$$

The function `Indices::GetCurrentMaster` calculates and returns this value.

```
int GetCurrentMaster() const;
```

Indices overloads three operators necessary for automatic looping, `*` (dereference), `!=`, and `++` (pre-increment). The dereference operator simple calls and returns `GetCurrentMaster`. The not equal to operator takes an Indices object as input and compares the output of its own `GetCurrentMaster` to the output of the input object's `GetCurrentMaster`. The pre-increment operator does nothing if `iDet` $\geq$ `nDet`. Otherwise, it starts by incrementing `iFlav`. It then checks whether `iFlav` equals `nFlav`, and if so, it resets `iFlav` to 0 and increments `iPar`. It performs the same check procedure using `iPar`, `nPar`, and `iXSec`, and finally performs one last check procedure with `iXSec`, `nXSec`, and `iDet`. Thus, when the Indices object has reached its maximum, `iDet` = `nDet`, and `iFlav` = `iPar` = `iXSec` = 0, `GetCurrentMaster` would return the product `nDet` × `nXSec` × `nPar` × `nFlav`, and further use of the pre-increment operator cannot increase past this special value.

3.5.2 Parameters

The Parameters class can generally be thought of to have two purposes, configuring the parameter sets for Spectra and performing operations on the current parameter set. To these ends, Parameters stores a `std::vector<NuFlav>` of neutrino flavors, a `std::vector<Parent>` of parent species, a `std::vector<std::string>` of cross sections to apply, a `std::vector<Detector>` of detectors, and an Indices object. A Spectra will create a distribution for each set of parameters within the four vectors. While most of the vectors can hold any valid object, the cross section vector will only contain entries that exactly match one of the outputs from the XSec class function `XSec::ListIntTypes`, or the string `"NoXSec"`.

When a user constructs a Parameters object, the constructor takes an input boolean to determine whether to consider or ignore the sign of the neutrino parent, and this parameter is stored in the private boolean `fSignSensitive`. The user can check what this variable is set to using the function `IsSignSensitive`.

```
Parameters(bool SignSensitive = true);

bool IsSignSensitive() const;
```

When reading Dk2Nu files, the sign sensitivity will determine whether to consider the parent PDG code exactly, or the absolute value of the PDG code. The sign sensitivity also determines what default parameters are set up. The publicly accessible `SetDefaults` function is called automatically in the Parameters constructor. This function sets up the NuFlav vector with the `NuFlav::AllNuFlavs` function, sets up the Parent vector with the `Parent::AllParents` function, and sets up the cross section vector with no cross section, charged current, and neutral current. `SetDefaults` takes a boolean input for sign sensitivity which is passed to its call to `Parent::AllParents`.

```
void SetDefaults(bool SignSensitive);
```

The call to `SetDefaults` within the Parameters constructor uses the same boolean input as the boolean input to the constructor itself. However, the constructor later removes the tau

Table 3.4: Functions to remove each parameter. Each function is type void. The function will remove all objects that match the input argument.

Function	Input Type	Meaning of Input
`RemoveNuFlav`	`int`	Neutrino PDG
`RemoveNuFlav`	`std::string`	Flavor Label
`RemoveNuFlav`	`NuFlav`	NuFlav Object
`RemoveParent`	`int`	Parent PDG
`RemoveParent`	`std::string`	Species Label
`RemoveParent`	`Parent`	Parent Object
`RemoveXSec`	`std::string`	Cross Section Interaction Process
`RemoveDetector`	`std::string`	Detector Label

and anti-tau neutrinos from the neutrino flavor vector.

The user can add or remove parameters by using the appropriate public member functions. A NuFlav or Parent can be removed using `RemoveNuFlav` or `RemoveParent`, respectively. Either of these functions can be called by inputting an `int` PDG, `std::string` particle name, or an object of the appropriate class. `RemoveNuFlav` simply calls the corresponding version of the `NuFlav::RemoveNuFlav` function, and `RemoveParent` calls the corresponding version of the `Parent::RemoveNuParent` function. `RemoveXSec` and `RemoveDetector` each have a single version that takes as input a `std::string` corresponding to the name, or title, of the object to remove. These functions are summarized in Table 3.4.

The functions to add a parameter are more specific, and generally require the an object of the appropriate class as input. The exception is that there is no `AddNuFlav` function. For neutrino flavors, the user must call `ResetNuFlavs` to repopulate the flavor vector with all of the flavors, then remove undesired flavors. The tau and anti-tau neutrinos can be removed together using the `RemoveNuTaus` function, which makes two calls to `RemoveNuFlav`. The available functions to add parameters are summarized in Table 3.5.

As mentioned in the Detector section, the user can change the uses for a detector within Parameters. This is done using the `SetDetUses` function with a detector label and number of uses.

```
void SetDetUses(std::string detname, int nuses);
```

Table 3.5: Functions to add each parameter. Each function is type void. AddParent will not add an object if one with the same PDG already exists, AddXSec will not add an object if the name does not match an output from XSec::ListIntTypes and is not "NoXSec", and AddDetector will not add an object if one with the same label already exists.

Function	Input Type	Purpose of Function
`ResetNuFlavs`	None	Reset `NuFlav` vector to include all flavors
`AddParent`	`Parent`	Add given Parent object
`AddXSec`	`std::string`	Add given cross section interaction process
`AddDetector`	`Detector`	Add given Detector object

Table 3.6: Functions to get the current indices and the total number of each parameter. Each function returns type int and takes no arguments.

Function	Return	
`GetCurrentNuFlav`	Current Flavor	`fIndices.iFlav`
`GetCurrentParent`	Current Parent	`fIndices.iPar`
`GetCurrentXSec`	Current Cross Section	`fIndices.iXSec`
`GetCurrentDet`	Current Detector	`fIndices.iDet`
`GetCurrentMaster`	Current Master	`fIndices.GetCurrentMaster()`
`NFlav`	Number of Flavors	`fNuFlav.size()`
`NPar`	Number of Parents	`fParent.size()`
`NXSec`	Number of Cross Sections	`fXSec.size()`
`NDet`	Number of Detectors	`fDet.size()`

The remaining member functions of Parameters generally have to do with indexing. The user cannot access the private Indices object, but the public GetCurrent functions can return the current indices. The number of each parameter can also be accessed via public member functions. These functions are summarized in Table 3.6.

A limited amount of the information stored in each parameter vector can be accessed using the current indices. The following public function declarations from Parameters.h show what information is accessible.

```
/// Pull a stored detector to call its class functions
Detector GetDetector(int i_det) const;

/// Pull a stored NuFlav to call its class functions
NuFlav GetNuFlav(  int i_flav) const;

/// Shortcut to access necessary fields
std::string GetDetName  (int i_det)  const;
```

Table 3.7: Functions that can alter the current indices stored in fIndices. Each object is type bool indicating whether the function was successful in setting the index.

Function	Input Parameter	Goal of Function
`SetCurrentNuFlav`	Neutrino PDG	Set flavor index to stored NuFlav with matching PDG
`SetCurrentParent`	Parent PDG	Set parent index to stored Parent with matching PDG
`SetCurrentXSec`	Cross Section Index	Set cross section index to input
`SetCurrentDet`	Detector Index	Set detector index to input
`SetIndices`	Master Index	Set all indices to match input

```
int         GetNuFlavPDG(int i_flav) const;
int         GetParentPDG(int i_par)  const;
std::string GetXSecName (int i_xsec) const;
```

The two remaining public functions, both named `MaxMaster`, allow the user to access particular values of the master index.

```
int MaxMaster() const;
int MaxMaster(int i_det) const;
```

When no input is specified, the function returns the true maximum master possible, the product of the four totals of each parameter. When an input is specified, it is taken to be the index of a specific detector, and the value returned is the product of the detector index and the three remaining parameter totals, i.e., $i_{Det} \times n_{XSec} \times n_{Par} \times n_{Flav}$. This second version returns the maximum index for the specified detector (or rather, a value of 1 higher than this).

Parameters has several private functions that aide indexing. First, the void `UpdateIndices` function is called whenever a parameter is added or removed, which updates the parameter totals within fIndices.

```
void UpdateIndices();
```

Various Set functions can alter the current indices within fIndices. These are summarized in Table 3.7.

Since each master index refers to a unique set of four parameters, and every histogram

that is ultimately created from each parameter set needs a name, the `NameTag` function creates a unique label by conglomerating the four parameter labels separated by '_' characters.

```
std::string NameTag(int master);
```

Parameters has the ability to automatically loop over all of its parameters. It implements the `begin` and `end` functions to accomplish this.

```
Indices begin();
Indices end() const;
```

The `begin` function sets a Parameters object's own fIndices current indices all to 0 and returns this object. In particular, this keeps the total number of each parameter intact. The `end` function creates a new Indices object and sets the total number of each parameter to match that in fIndices, then sets the new object's 'current' indices to be the maximum possible, i.e., `iDet` = `nDet`, `iFlav` = `iPar` = `iXSec` = 0. Given a Parameters object that has already been configured, the code below shows how to automatically loop over the parameters.

```
// Assuming params is a Parameters object,
// and preconfigured in earlier code...
for(const auto& index: params) {
  // Do something...
  std::cout << index << ", " << params.GetCurrentMaster() << std::endl;
}
```

As a final note, Parameters makes every Spectra class, FluxReader, and Combiner a friend class. When a Spectra is constructed, it makes use of the private Parameters copy constructor to freeze the current set of parameters.

```
Parameters(const Parameters& params);
```

Combiner uses the private `ClearAll` function to make a dummy Parameters object that it can build from the ground up. Calling `ClearAll` resets all of a Parameter object's vectors and sets all of its internal Indices variables to 0.

```
void ClearAll();
```

3.6 Var and Weight

Filling a histogram with weighted events requires two important values, namely the value to fill and the weight of the entry. The Var and Weight classes fulfill these roles. Both classes are very similar in structure; they read specific variables from a single Dk2Nu neutrino ray and return a single value based on an internally stored function. Despite their similarity, they are not derived from a common mother; they are fully independent of each other.

3.6.1 Var

The Var class defined in Var.h and Var.cxx stores a function that returns a single value based on a single Dk2Nu entry, and a list of variables it needs from the entry to evaluate its function.

The function would need a Dk2Nu object at the very least to be evaluated. A `typedef` is utilized to standardize the function type.

```
typedef double (VarFunc_t)(const bsim::Dk2Nu* nu, const int& i_nuray);
```

The Dk2Nu object's nuray branch is a vector, so the `i_nuray` input specifies which index to use. With this definition, the Var stores a private function.

```
std::function<VarFunc_t> fFunc;
```

The other private variable stored by Var is a set of variables that the function needs to read.

```
std::set<std::string> fBranches;
```

The code does not check whether variables are accessed inside the function that are not in the set of branches, but if this occurs, a segmentation fault is likely to occur at run time.

In order to evaluate the Var's function, the function call operator is overloaded to take the same inputs as the function. The code below from Var.h shows how this is implemented.

```
double operator()(const bsim::Dk2Nu* nu, const int& i_nuray)
{
  return fFunc(nu, i_nuray);
}
```

Var has two constructors, a standard and copy constructor. The copy constructor is used by the Spectra class to store its own internal copy of the Var. The standard constructor simply takes a set of branch names and a function. Common Var objects are defined in Vars.h, and Vars.cxx ensures that they are built during compilation. The common energy variable is printed below to demonstrate construction of a Var.

```
const Var kEnergy({"nuray", "nuray.E"},
                  [](const bsim::NuRay* nu, const int& i_nuray)
                  { return nu->nuray[i_nuray].E; });
```

The first part of the declaration, `const Var kEnergy` defines the type and variable name, much the same as `const int x`. The first input in the constructor, `{"nuray", "nuray.E"}` is the list of branch names needed by the function. The final two lines are a lambda function. The first line of the lambda defines what the function will capture (nothing) and what the function's input will be (a Dk2Nu object and integer). The second line in the curly braces is the actual function logic. This logic can span multiple lines of code by separating them by the normal semicolon.

Since the integer input to the function specifies an index of the Dk2Nu nuray vector, any variable that does not access this branch can omit `i_nuray` within the lambda function's input argument though `const int&` must always appear. This is demonstrated by the p_z variable below.

```
const Var kpz({"decay", "decay.pdpz"},
              [](const bsim::NuRay* nu, const int&)
              { return nu->decay.pdpz; });
```

3.6.2 Weight

The Weight class, defined in Weight.h and Weight.cxx, is very similar to the Var class. The main difference is that there are more inputs to the function for Weight than the function for Var. The `typedef` utilized by Weight shows the enlarged set of inputs.

```
typedef double (WeiFunc_t)(const double& w, const bsim::Dk2Nu* nu,
                           const int& i_nuray, const TObject* extW);
```

Like Var, Weight stores a list of branches it needs to read from the Dk2Nu object, a function it uses to evaluate, and overloads the function call operator. The list of branch names is exactly the same between the two classes, the function and overloaded function call operator reflect the inputs necessary for Weight.

```
std::function<WeiFunc_t> fFunc;

double operator()(const double& w, const bsim::Dk2Nu* nu,
                  const int& i_nuray, const TObject* extW)
{
  return fFunc(w, nu, i_nuray, extW);
}
```

Weight also has two constructors, both a standard and copy constructor. The copy constructor serves the same purposes as the Var copy constructor. The standard constructor is set up nearly identically to that in Var, it simply requires the appropriate enlarged set of inputs in the lambda function's input set.

There are three Weight objects defined within Weight.h and Weight.cxx. These are a default weight, no weight, and constant weight. The default weight, shown below, shows that the `double` input `w` is exactly the default weight. Notice that, since this weight needs only that input, the label for each of the other inputs is omitted.

```
const Weight kDefaultW({}, [](const double& w, const bsim::Dk2Nu*,
                              const int&, const TObject*)
                         { return w; });
```

The weight that applies no weight, kNoWeight, always returns 1, and the constant weight must be called with a double value, `c`, and the weight will always return that weight.

The other input unique to the Weight function is the `TObject` pointer, `extW`. This input allows for the application of external weights. When a Weight object is constructed that will use external weights, any object that inherits from a `TObject` can be used. However, within the function logic (in curly braces), the external weights must be cast as the class is actually is in order to use that class' functions. The FluxReader demo script Demo2_VarWeight.C shows how this is done using a `TSpline3`. The Weight defined there is shown below.

```
const Weight kAppXSec({"nuray", "nuray.E"},
                      [](const double& w, const bsim::Dk2Nu* nu,
                         const int& i_nuray, const TObject* extW)
                      { TSpline3* xsecspline = (TSpline3*)extW;
                        double energy = nu->nuray[i_nuray].E;
                        double xsec = xsecspline->Eval(energy);
                        if(xsec < 0.) {
                          xsec = 0.;
                        }
                        return w*xsec; });
```

The first line of the function logic casts the external weight as the correct class; this allows for usage of `TSpline3::Eval`. As a note, this Weight was created for demonstration purposes only, and should not be otherwise used to apply cross sections.

The file Weights.h should be populated with commonly applied weights that are not defaults defined in Weight.h and Weight.cxx. The file Weights.cxx ensures that these objects are built during compilation.

3.7 Spectra

The Spectra class defined in Spectra.h and Spectra.cxx is designed to contain all of the distributions that will be filled by FluxReader. Spectra itself is an abstract class that only sets up some common machinery for its various daughters. A separate daughter class is implemented for one, two, and three dimensional Spectra. A fourth daughter is implemented

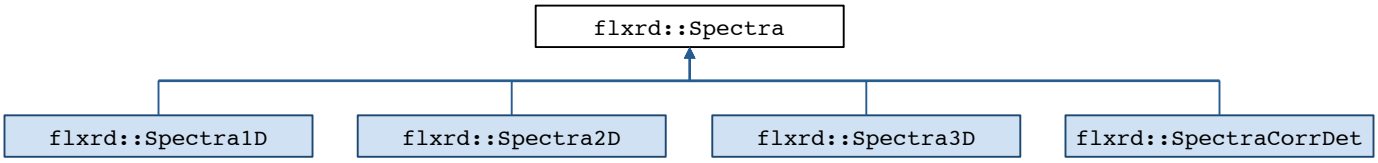

Figure 3.3: The inheritance structure of Spectra and its daughter classes.

for a Spectra type that plots a variable calculated at one detector location on its x axis and the same variable calculated at a different detector location on its y axis. The inheritance structure is shown in Figure 3.3. Every constructor for Spectra and its daughters is protected or private, so the user cannot interface directly with this class. However, Spectra makes FluxReader a friend class so it can create Spectra objects. (Each daughter class also explicitly makes FluxReader a friend class.)

Since Spectra is an abstract class, it defines common variables and virtual functions, though some functions are fully implemented. Each Spectra object stores a Parameters object that determines how many and what histograms are created. A title is defined that will label each histogram and be used an output directory name by FluxReader. Every Spectra will have at least a Var for its x axis and a Weight to weight each event. If external weights are needed, this object is stored as a `TObject*`.

Spectra stores a set of branch names needed by its weight and all its variables, and this set is populated when Spectra is constructed. This list can be accessed using the `BranchesToAdd` function.

```
std::set<std::string> BranchesToAdd() const { return fBranches; }
```

The `Detectors` function looks in the stored Parameters object and returns a set of Detector objects the Spectra will use.

```
std::set<Detector> Detectors() const;
```

Cross sections needed for filling are stored as `TSpline3` pointers in a `std::map`. The spline is the value in the map, and the key is a `std::string` unique to each spline. This key is generated using the `XSecName` function, which puts together the labels for a neutrino

flavor, interaction process, and detector name into one string.

```
std::string XSecName();
```

The splines are created and the map is populated by the `SetupXSec` function, which is called in the Spectra constructor.

```
void SetupXSec();
```

This function loops through the stored Parameters object, makes the cross section key for each parameter set, and if it is new, generates a new spline and adds it to the map.

The remaining fully implemented function simply returns the Spectra title.

```
std::string GetTitle() const { return fTitle; }
```

The remaining functions set up by the abstract class are all virtual. These functions, implemented by Spectra's daughters, are for accessing a specific histogram, filling the histograms, and writing the histograms to a directory. The function declaration are shown below.

```
virtual TH1* GetHist(int i_hist) = 0;
virtual void Fill(brim::Dk2Nu* nu,
                  std::map<std::string, int> nurayIndices) = 0;
virtual void WriteHists(TDirectory* dir) = 0;
```

The map input for `Fill` is set up in FluxReader and matches a particular index of the Dk2Nu nuray vector to a detector location.

As mentioned above, the constructor sets up the necessary cross section splines and creates a list of Dk2Nu variables needed by its variables and weight. If no external weights are needed, it sets its internal external weights object to the null pointer. Aside from the branches needed by the variables and weights, Spectra adds some default branches. The decay and nuray branches are automatically added. To differentiate neutrino flavor and parent species, the variables containing these values, stored in the decay branch, are added. Since cross sections are evaluated by energy, the nuray energy variable is automatically

added. Finally, the default weight mentioned in the Weight class requires the nuray branch propagation weight, the decay branch importance weight (and a cross section), so these variables are added.

3.7.1 Spectra1D, Spectra2D, Spectra3D

Spectra1D defined in Spectra1D.h and Spectra1D.cxx, Spectra2D defined in Spectra2D.h and Spectra2D.cxx, and Spectra3D defined in Spectra3D.h and Spectra3D.cxx are all very similar in design. Spectra1D is first discussed in detail, then the differences between it and Spectra2D and Spectra3D are shown.

Spectra1D is the one dimensional implementation of the Spectra class. It adds the private variable `std::vector<TH1D*> fHists`, a vector of one dimensional histograms. `GetHist` is implemented to return a specific histogram from that vector.

`Fill` is implemented to fill multiple histograms. It gets the neutrino flavor and parent species from the input Dk2Nu object and sets its internal Parameters object to these values using the functions `Parameters::SetCurrentNuFlav` and `Parameters::SetCurrentParent`. If those functions fail, `Fill` returns without filling any histograms. Assuming they both succeed, it loops over detectors, cross sections, and detector uses (in that order with the detectors loop as the outermost loop). The internal Parameters object is set to the current detector and cross section within the appropriate loop, then the corresponding master index is retrieved with `Parameters::GetCurrentMaster`. The histogram stored at that master index is then filled with the Dk2Nu entry, multiple times smeared through the detector if necessary. The value and weight to fill are computed using the overloaded function call operators discussed in Var and Weight.

`WriteHists` creates subdirectories in the `TDirectory*` given as input, then writes the stored histograms in these subdirectories. It creates a `std::string` to store the current detector name. Next, a loop over all parameter sets as described in Parameters is started. It sets the internal Parameters indices to the current index, then checks if the Detector label

is different from the previous iteration. If so, the current directory is moved to the top level directory given as input to `WriteHists` and the current detector name is updated. The top directory is searched for a subdirectory with the current detector, and it is created if it does not exist. Either way, the current directory is moved to the detector subdirectory. The histogram corresponding to the current master is then written in the current directory. Once the loop terminates, the current directory is moved to that before `WriteHists` was called, and the function returns.

Spectra1D also implements a new private function, `CreateHists`.

```
void CreateHists(std::string labelx, std::vector<double> binsx);
```

This function constructs all of the histograms to be filled and pushes them into the histogram vector. They are generated in a loop over the internal Parameters object. Each histogram is created to match the current parameter set, ensuring that `Fill` always fills the correct histogram. The title of each histogram is set to the Spectra title followed by a '_' character and the output from `Parameters::NameTag`. Every histogram is made with the same axis label and binning; these are from the two inputs to `CreateHists`.

The constructor for Spectra1D is private, but accessible to FluxReader through friendship. This constructor simply calls the base class constructor, then calls `CreateHists` to set up the histogram vector. While the base class constructor requires a Parameters object, title, x variable, weight, and optional external weight object, the Spectra1D constructor also requires an x axis label and binning scheme. These extra parameters get passed to `CreateHists`; they are not stored after construction. The different base constructor and Spectra1D constructor are shown below.

```
Spectra  (Parameters params, std::string title,
          const Var& varx, const Weight& weight,
          TObject* extWeights = nullptr);

Spectra1D(Parameters params, std::string title,
          std::string labelx, std::vector<double> binsx, const Var& varx,
          const Weight& weight, TObject* extWeights = nullptr);
```

Spectra2D is the two dimensional implementation, and Spectra3D is the three dimensional implementation of Spectra. They only have small differences from their one dimensional sibling. First, the private histogram vector changes from storing `TH1D*` to `TH2D*` or `TH3D*`. Spectra2D stores a second Var object for its y axis, while Spectra3D stores two extra objects for its y and z axes.

`GetHist` and `WriteHists` are implemented identically. (They cannot be in the base class since the base class does not have the vector of histograms.) `Fill` is the same for all three siblings except for the line that actually fills the histogram–since the number of dimensions differ, the number of arguments to the fill function differs. The higher dimensional version only change the line by evaluating the extra stored Var objects.

Spectra2D and Spectra3D also create a private `CreateHists` function. They require an additional axis label and binning (or two) for the additional axes, so this function could not be a virtual function. The implementation only changes by adding axis labels to each additional axis and using the histogram constructor for the appropriate dimension.

The constructors for Spectra2D and Spectra3D take additional axis labels and binnings, but also require the additional Var objects as well. The base constructor and all three dimensional daughter constructors are shown below.

```
Spectra  (Parameters params, std::string title,
          const Var& varx, const Weight& weight,
          TObject* extWeights = nullptr);

Spectra1D(Parameters params, std::string title,
          std::string labelx, std::vector<double> binsx, const Var& varx,
          const Weight& weight, TObject* extWeights = nullptr);

Spectra2D(Parameters params, std::string title,
          std::string labelx, std::vector<double> binsx, const Var& varx,
          std::string labely, std::vector<double> binsy, const Var& vary,
          const Weight& weight, TObject* extWeights = nullptr);

Spectra3D(Parameters params, std::string title,
          std::string labelx, std::vector<double> binsx, const Var& varx,
          std::string labely, std::vector<double> binsy, const Var& vary,
          std::string labelz, std::vector<double> binsz, const Var& varz,
```

```
const Weight& weight, TObject* extWeights = nullptr);
```

Each constructor calls the base class constructor and its own implementation of `CreateHists`. However, the two and three dimensional daughters also add the required branches for evaluating their y (and z) variables to their internal list of branch names.

3.7.2 SpectraCorrDet

SpectraCorrDet, defined in SpectraCorrDet.h and SpectraCorrDet.cxx, differs significantly from the other Spectra. The goal of the SpectraCorrDet is to plot a single variable with its value at one detector vs its value at a second detector. While being filled into an appropriate two dimensional histogram, it is weighted by the weight at the y axis detector. At the same time a one dimensional distribution of events at the x axis detector is stored using the weight at the x axis detector. Once filling is complete, this one dimensional spectrum is used to normalize each column of the final two dimensional distribution.

The normalization leads to another design decision. The basic dimensional spectra can be combined by simply adding bin by bin, but this is not possible for the SpectraCorrDet. The Combiner class performs that operation for the dimensional spectra, but this is done automatically for the SpectraCorrDet. That way, the one dimensional normalization distribution do not have to be written to file. This operation does not change the original plots, but simply creates new ones that combine plots with either similar neutrino flavors or parent species.

When a SpectraCorrDet is constructed, it creates and initializes two private booleans to false; these indicate whether the histograms have been combined or normalized. This class must store two histogram vectors, one for the desired two dimensional distributions, and a second for the one dimensional normalizations. The Parameters detector indices corresponding to the x and y detectors are also stored in private `int` variables.

`GetHist` and `WriteHists` are implemented nearly identically as with the dimensional spectra, but before any operations are performed, the functions first check whether the

histograms have been normalized yet, and if not, calls the `Normalize` function.

```
void Normalize();
```

`Normalize` first checks whether the plots have been combined yet, and calls `CombineAll` if not. It then sets the combined indicator boolean to true. Next it loops over all histograms in the two dimensional histogram vector, pulls the corresponding normalization histogram, and normalizes each column in the two dimensional histogram by the bin in the normalization histogram. Once the loop terminates, the normalization indicator boolean is set to true.

`Fill` is implemented with some similarities to its dimensional siblings. The correct neutrino flavor and parent species must be set, and a loop over cross sections must still be performed. However, there is no loop over detectors. The two dimensional spectra must now be filled by looping over both the x and y detector uses, while the one dimensional spectra is filled only by looping over x detector uses. As mentioned previously, the weight at the x detector is used for the normalization histograms, and the weight at the y detector is first used for the full two dimensional histograms.

SpectraCorrDet also implements a `CreateHists` function. It requires the label of both axis detectors, and a single axis label and binning. While looping over parameter sets, both the one and two dimensional plots are created, ensuring their histogram vector indices match. This loop does not go over the full parameter set; instead, it sets the internal Parameters object to the y axis detector, and loops over all flavors, parents, and cross sections. When actually constructing the histograms, the two dimensional histograms are given axis labels in the form "Detector Name Label," i.e., the detector label precedes the axis label for both the x and y axes. These histograms are titled in several steps. The first puts the Spectra title and output from `Parameters::NameTag` together into one string, separated by a '_' character. Since the name tag has the y axis detector at the end, this section is replaced by the x axis and y axis detectors together, also separated by a '_' character. The title of the two dimensional histogram is set to this string. The normalization plots are not given a title or axis label since they do not get written to file.

The SpectraCorrDet constructor requires a slightly different set of inputs than its dimensional siblings. The base constructor, one dimensional constructor, and SpectraCorrDet constructor are shown below.

```
Spectra  (Parameters params, std::string title,
          const Var& varx, const Weight& weight,
          TObject* extWeights = nullptr);

Spectra1D(Parameters params, std::string title,
          std::string labelx, std::vector<double> binsx, const Var& varx,
          const Weight& weight, TObject* extWeights = nullptr);

SpectraCorrDet(Parameters params, std::string title,
               std::string detX, std::string detY,
               std::string labelx, std::vector<double> binsx,
               const Var& varx,
               const Weight& weight, TObject* extWeights = nullptr);
```

The constructor first calls the base constructor, then checks that the Parameters object has at least two detectors. If not, the code breaks. The internally stored indices corresponding to the two detectors are set to -1. Next, the constructor loops over the Detectors in the Parameters object and changes the internally stored detector indices to the correct value when each detector is found. After the loop, the code breaks if either index remains at -1. Lastly, the constructor calls `CreateHists`.

Three other private functions are implemented, shown below.

```
void CombineAll();
void CombineNuFlavs(std::vector<TH2D*>& newHists,
                    std::vector<TH1D*>& newNorms);
void CombineParents(std::vector<TH2D*>& newHists,
                    std::vector<TH1D*>& newNorms);
```

These functions add histograms together to create larger combined plots. They are very similar to those found in Combiner, so they will be in Subsection 3.7.2.

3.8 FluxReader

The aptly named FluxReader actually reads the Dk2Nu files and provides each Spectra with the information necessary to fill histograms.

A user must construct a FluxReader object with a file path name. The constructor also takes two optional integer arguments. The path name can contain wildcard characters, and it is immediately expanded with the Utilities Wildcard function into a private vector of filenames. The first optional integer input specifies a maximum number of files to use, and the second is a number of files to skip. The files to skip are removed from the front of the filename vector. If what remains is larger than the first optional argument, files are removed from the end of the filename vector. When the first integer is left at or set to 0, this special value tells the constructor to not remove any files off the end of the vector. Lastly, the constructor stores the name of the standard Dk2Nu tree, `"dk2nuTree"`.

FluxReader has four `AddSpectra` functions, each corresponding to a specific daughter of Spectra, and the inputs to each function call exactly match the inputs to their respective constructors. Inside each of these functions, a Spectra objects is constructed and pushed into a private vector, `std::vector<Spectra*>`. By setting up this vector as pointers to the base class, it can contain any of the daughters. Ultimately, FluxReader only needs to access `Spectra::Fill` and `Spectra::WriteHists`, and these are both declared in the base class. `AddSpectra` also pulls the list of required branch names from each Spectra using `Spectra::BranchesToAdd` and puts them into one master set.

Once the user has finished setting up spectra with `AddSpectra`, the user can then call `ReadFlux` to do the real work. An output directory must be provided as input to the function. `ReadFlux` essentially calls the FluxReader private functions in a specific order and then loops over Dk2Nu entries. Each of these functions are introduced as necessary to `ReadFlux`.

```
void ReadFlux(TDirectory* out);
```

`ReadFlux` begins by calling `AddDefaultBranches`.

```
void AddDefaultBranches();
```

Dk2Nu has the ability to project neutrino rays to different locations on the fly, changing their propagation weights and energies accordingly. `AddDefaultBranches` adds the variables required to do this to the master set. The next function call is `InitialMessage`, a function that lets the user know that looping over flux files is imminent and outputs the titles of each Spectra that will be created.

```
void InitialMessage();
```

Next up is `SetNuRayIndices`.

```
void SetNuRayIndices();
```

This function calls `Spectra::Detectors` for each of its stored Spectra, and puts all of the results into its own private set Detectors. It then loops over this master set, and sets up a private map. The map keys are detector names, and the map values are integers. The integers will correspond to nuray vector indices. A dummy integer is set to 0, and this is the value for the first detector key. The dummy is incremented by the number of uses for the first detector, and the result is the value for the next detector. This continues until the last detector. Once the dummy has been incremented by the number of uses for the last detector, the final result is used as the value for a special key, `"znull"`. This will correspond to the first out of bounds index of the nuray vector.

`ReadFlux` next creates a `TChain*` from all of the files in the filename vector, and calls `SetBranches` with this chain.

```
void SetBranches(TTree* tree);
```

This function first sets every branch status to its input tree to 0, or off. It then sets its private vector of actual `TBranch*` objects (as opposed to their name as a `std::string`) to have enough size to contain as many elements as in the set of branch names. `SetBranches` then loops over the branch names. Each branch in the set is turned back on in the input

tree, and the tree's branch is added to the vector of `TBranch*` objects. If this operation fails, the code will break. FluxReader stores a private Dk2Nu object, and the `TTree*` branch addresses are set to this object. Every time `TTree::GetEntry` is called, the FluxReader Dk2Nu object is repopulated with the values of the new entry. Next, `SetBranches` outputs the names of all the branches that are turned on. Finally, the nuray vector in the FluxReader Dk2Nu object is sized to the value stored by `"znull"` in the nuray indices map.

This is all of the preliminary steps for reading the flux files. `ReadFlux` then begins looping over Dk2Nu entries from the input files. A message is output every 250000 entries and when moving to the next tree (file). A running sum of the Dk2Nu potnum variable is computed for each entry. Next each entry is reprojected to each detector location. The nuray index for each detector is found from the nuray index map, and the reprojected values are stored at the index in the FluxReader Dk2Nu object's nuray branch. Since the index in the map was incremented by the number of uses at each detector, there are enough consecutive entries in the nuray vector to store a copy of each projection. When the number of uses is greater than 1, FluxReader smears each projection through the detector volume with the `Smear` function.

```
TVector3 Smear(const Detector& det, double rr = -1);
```

`Smear` gets half of each dimension using the Detector GetHalfSize functions, and for each dimension picks a random point according to Equation 3.2.

$$-d_i/2 \leq x_i \geq d_i/2 \tag{3.2}$$

`Smear` currently supports rectangular and circular faces. When the parameter rr is specified, the x and y random points are recalculated until $x * x + y * y \leq r * r$.

Whether or not the projection location is smeared, the projection location is transformed into beam coordinates with `ToBeamCoords`.

```
void ToBeamCoords(const Detector& det, TVector& xyz);
```

This function pulls the detector location using the Detector GetCoord functions, then transforms the input coordinates according to Equation 3.3, where $\theta = 3.323155°$ is the beam angle.

$$\begin{array}{ccc} x & \rightarrow & d_x + x \\ \begin{pmatrix} y \\ z \end{pmatrix} & \rightarrow & \begin{pmatrix} d_y \\ d_z \end{pmatrix} + \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} \end{array} \tag{3.3}$$

Dk2Nu entries are reprojected to the location output by `ToBeamCoords`, and the results of these projections are what get stored in the FluxReader Dk2Nu object.

At this point, the internal Dk2Nu object and nuray index map are given to each Spectra for filling, and then the next loop iteration begins with the next entry.

Once the loop over entries is complete, `ReadFlux` outputs the total number of entries and POT and begins the write out process. A POT histogram is created with a single bin set to the total POT. This histogram, titled TotalPOT, is written out to the directory provided as input to `ReadFlux`. Next, a loop over each Spectra object is performed. In the output directory, a subdirectory is created that matches the title of current Spectra, and the current directory is changed to that new directory. `Spectra::WriteHists` is called using the current directory, which is the Spectra subdirectory. Once this loop terminates, all of the histograms have been written out, the current directory is returned to that before `ReadFlux` was called, and the function returns.

3.9 Combiner

The Combiner class, defined in Combiner.h and Combiner.cxx, is designed to combine certain subsets of histograms produced by FluxReader. It will create new histograms by adding together these subsets, but only for the basic one, two and three dimensional Spectra. The Combiner constructor takes the path to a FluxReader output file as its only argument.

```
Combiner(std::string out);
```

The constructor will open the file for updating, so any new plots are appended inside the relevant directories. Once the file is open, there is a loop over all objects in the top level file directory. The name of any object that is a `TDirectory` is stored in a private set of Spectra names. The name stored is the subdirectory name and title originally given to the Spectra.

The constructor next loops over all the Spectra it found in order to set up the Parameters that were used to originally create them, but it first sets up a vector of label names in case any of them are SpectraCorrDet objects. Inside the loop, local sets of labels are set up for each of the four parameter types. The current directory is set to the current Spectra, and another loop is performed over objects in this subdirectory. The label of any object that is a `TDirectory` is added to the set of detectors. Once the inner loop finishes, the outer loop proceeds based on the number of detectors found. If there were none, the current object was a SpectraCorrDet and the name of this object is added to the vector set up outside the outer loop. Otherwise, the current directory is moved to the first detector from the set. Inside this subdirectory, a loop is performed over all objects, which should all be histograms. The histogram name contains all the information about what parameter set was used to create it. A local copy of this name is created and deconstructed bit by bit to get the name of the neutrino flavor, parent species and cross section used in its generation. Each of these labels is stored in their appropriate set. Once this second inner loop terminates, a new Paramters object is constructed and set up with `SetupParameters`, discussed in more detail below. This object is added to a map that has the Spectra label as a key and the Parameters object as its value. Note that this is done regardless of what type of Spectra was in the current outer loop iteration. Once the outer loop over all Spectra is finished, the Combiner constructor loops over all SpectraCorrDet objects it found and erases each from the set of Spectra it has stored. This leaves those particular entries in the stored map, but essentially erases where to find the their keys. Lastly, the constructor calls `InitialMessage`, which outputs all of the (remaining) Spectra names, and the number of all four parameters in their corresponding

Parameters objects.

```
void InitialMessage();
```

The Combiner constructor makes use of the private `SetupParameters` function.

```
void SetupParameters(Parameters* params,
                     std::set<std::string>& dets,
                     std::set<std::string>& nuflavs,
                     std::set<std::string>& parents,
                     std::set<std::string>& xsecs);
```

This function removes all default parameters from the input Parameters object using the `Parameters::ClearAll` function, then uses `Parameters::ResetNuFlavs` to add back in all neutrino flavors. It then loops over all of these neutrino flavors and searches for each in the input set of neutrino flavors. If a particular flavor is not found in the input, then it is removed from the Parameters object. For the other three types of parameters, a loop is performed over the input set adding each into the Parameters object using dummy objects. Combiner only needs the parameter labels and does not care about things like position or PDG. Thus, the Parent dummy is created using a PDG of 0 and each dummy created increments the PDG by one, while using the correct label. For cross sections, no dummy needs to be set up; the label is directly added to the Parameters object. For detectors, the dummy object uses an empty target nucleus, positions and sizes of 0, and 0 uses.

Combiner has three public functions the user can use, each shown below.

```
void CombinerNuFlavs();
void CombinerParents();
void CombineAll();
```

All three behave similarly, so `CombineNuFlavs` will be shown in detail, then the differences between this function and the other two will be discussed.

`CombineNuFlavs` looks for sets of histograms that share a constant parent species, applied cross section, and detector location, and adds together all neutrino flavors. The resultant histogram has a name in the same form as any other, and uses `"allnu"` as its flavor name.

If any histogram has this string as part of its name, then Combiner must have already been performed, so `CombineNuFlavs` first checks for this condition with `CombineAlreadyCalled`.

```
bool CombineAlreadyCalled(std::string search);
```

This private function changes the current directory to the first detector in the first spectra found in the Combiner input file. It then loops over histograms in this subdirectory, looking for the input string in the histogram name. If the string exists in the name, then Combiner was already called and the function returns true. Otherwise, the current directory is changed back to what it was before `CombineAlreadyCalled` was called, and the function returns false.

`CombineNuFlavs` next begins a loop over all stored Spectra. The number of each parameter is pulled from the corresponding Parameters object, and three nested loops are started over the detectors, cross sections and parents, with parents being the inner loop. Along the way, the current directory is set to the current detector in the current spectra. Inside the inner loop, a master index is crafted manually using the convention defined in Indices with the current parent, cross section, and detector. The 'current' neutrino flavor is set to 0. The name of a histogram is recreated using the current spectra label and `Parameters::NameTag` function with the created index. This histogram will exactly match one in the input file to Combiner, and it is cloned into a new histogram. A final nested loop is performed over neutrino flavors. In this loop, the crafted master index is incremented by one at each iteration, matching the Indices convention. The new master index is used to update the recreated histogram name, and each of these names is used to get a histogram from the input file. These histograms are added to the one that was cloned from the input file.

Once the loop over neutrino flavors is finished, the final histogram's name is updated. The naming convention from the Parameters function `Parameters::NameTag` is exploited to quickly find and replace the neutrino flavor name with the replacement string used in the call to`CombineAlreadyCalled`. Finally, the histogram is appended to the current subdirectory. Outside of all the nested loops, the current directory is reset to that before `CombineNuFlavs` was called, and the function returns.

`CombineParents` is nearly identical to `CombineNuFlavs`. This function looks for sets of histograms with common neutrino flavors, applied cross sections, and detectors, and adds together all parent species. The biggest difference between the code in the two functions is the order of nested loops; in `CombineParents`, the final two loops are switched so that a loop over parent species is the inner most loop. The manually crafted master index must also be calculated differently. The original index sets the first parent index to 0 instead of the neutrino flavor index. Then, inside the inner loop over parents, the master index is incremented by the total number neutrino flavors to correspond to the Indices convention of incrementing the neutrino parent. Lastly, the replacement string sets the parent label to `"allpar"`; this is also used in the function's call to `CombineAlreadyCalled`.

`CombineAll` combines all histograms with a common applied cross section and detector, resulting in the most generic plot of neutrinos. It is similar in structure to `CombineNuFlavs` and `CombineParents` but has some important differences. Its call to `CombineAlreadyCalled` searches for plots with both neutrino flavors and parent species already combined, using the search string `"allnu_allpar"`. Then, it calls both `CombineNuFlavs` and `CombineParents`. This means that a user does not need to call either of those functions if `CombineAll` will be called later. Histograms must still be added together to create the final results, but the preceding combiner function calls mean that some of this work is already done. The set of nested loops in `CombineAll` completely omits a loop over parents. When the master index is manually set, the neutrino flavor and parent indices are set at 0. A histogram name is recreated as before, but it is immediately modified by replacing the parent label with `"allpar"`. This ensures that the histogram pulled and cloned from the input file is one that has already combined the parent species. A loop over neutrino flavors is then performed, and the parent label is always replaced with the combined parents string before pulling a histogram from the input file and adding it to the first, cloned histogram. The resultant histogram has its neutrino flavor replaced with `"allnu"`, and this final plot is appended to the appropriate subdirectory.

3.9.1 SpectraCorrDet Combining

The functions that combine histograms in SpectraCorrDet have the same names as those in Combiner, and are nearly identical copies. However, since the functions are part of the SpectraCorrDet class, there is no need to create a new Parameters object. Instead, the histograms can be pulled directly from the objects internal histogram vectors. The master index is crafted the same way as in Combiner, but this type of spectra does not loop over detectors, so the stored histogram vectors have indices that are offset from a true master. The histogram vector index is created by removing this offset. The Parameters function `MaxMaster` is called with the index of the detector before the y axis detector, and that value is subtracted from the master index. Both the original master and histogram index are necessary. The original master is used to recreate histogram names, while the histogram index is needed to access and add or clone the correct plots. The histogram addition and name replacement is performed the same way in SpectraCorrDet combine function as Combiner combine functions, but since SpectraCorrDet ultimately needs to normalize its histograms, each SpectraCorrDet combine function adds together its two dimensional plots and one dimensional normalization plots simultaneously.

The other major difference between the SpectraCorrDet and Combiner combine functions is the input arguments to the flavor and parent combining functions in the SpectraCorrDet versions. `SpectraCorrDet::CombineAll` still calls its class' versions of `CombineNuFlavs` and `CombineParents`. However, the results from both of those functions are stored in the two input vectors. `CombineAll` uses separate vectors for the combined flavor and combined parent histograms. When `SpectraCorrDet::CombineAll` proceeds in combining neutrino flavors from the already combined parent plots, it can simply add together all the histograms in the combined parent vector, instead of recreating a histogram name and modifying its parent label with `"allpar"`. Once `SpectraCorrDet::CombineAll` is finished creating new histograms, it appends the results to the stored vectors of two and one dimensional histograms starting with the combined neutrino flavor vectors and combined parent species vectors. It

lastly adds the histograms made directly within the `SpectraCorrDet::CombineAll` function body. This order is chosen to match that from `Combiner::CombineAll`.

3.10 Reading Non-Dk2Nu Files

The FluxReader framework provides machinery to read non-Dk2Nu files. This functionality **has not been tested**, so this section should be considered more as a design theory rather than actual mechanics. Furthermore, the Dk2Nu package has the ability to convert files to Dk2Nu format [9], so there should never be any reason to use these techniques.

FluxReader stores each neutrino ray entry in a Dk2Nu object, so any file must ultimately be mapped to this object. The Utilities function `OverrideAddresses` creates this map.

```
std::map<std::string, void*> OverrideAddresses(bsim::Dk2Nu* nu);
```

This function outputs a map with a standard branch label as its key, and the address of an actual Dk2Nu object as its value. The user must know the name of the non-standard branches and what they correspond to in Dk2Nu. The FluxReader function `OverrideDefaultVarName` will keep track of the differences.

```
void OverrideDefaultVarName(std::string oldname, std::string newname);
```

This function adds to a private map stored by FluxReader mapping Dk2Nu branch names to non-standard branch names. The user must call this function with the Dk2Nu branch name followed by the non-standard branch name for every non-standard branch needed. Since the map to Dk2Nu branch names exists, all Var and Weight objects should be constructed with Dk2Nu branch names.

The user must also call the FluxReader function `OverrideTreeName` if the tree storing neutrino entries has a non-standard name.

```
void OverrideTreeName(std::string treepath);
```

This function simply overwrites the path to the neutrino ray tree within the files to be read.

When `FluxReader::SetBranches` is called, it actually checks for override conditions. The default discussed in Section 3.8 only occurs if the tree name is the default path and the size of the map of standard to non-standard branch names is 0. When it is not, the code executed before outputting active branches to the user changes, though all branches are still turned off by default. First, a map using the Utilities `OverrideAddresses` function is created using the FluxReader Dk2Nu object as input. A loop is next performed over all necessary branches. The branch names should all be standard Dk2Nu names. The branch name is searched for in the map of standard to non-standard branch names. If it is found, the non-standard name is stored in a new local variable; otherwise, the standard branch is copied into the local variable. Using the name stored in the local variable, the branch is status is turned on in the flux file tree, and the branch address is set to the FluxReader Dk2Nu object branch.

This should ensure the correct variables are read into the correct location. However, files should be converted to Dk2Nu format [9] instead of using this method.

References

[1] R. Hatcher. Dk2Nu Package. https://cdcvs.fnal.gov/redmine/projects/dk2nu.

[2] R. Hatcher. Dk2Nu Tree Structure Source Code. https://cdcvs.fnal.gov/redmine/projects/dk2nu/repository/entry/trunk/dk2nu/tree/dk2nu.h.

[3] G. Kafka. FluxReader Repository. https://cdcvs.fnal.gov/redmine/projects/fluxreader/repository/show/trunk/FluxReader.

[4] R. Hatcher. Dk2Nu Repository. https://cdcvs.fnal.gov/redmine/projects/dk2nu/repository.

[5] R. Brun et. al. ROOT Reference Guide. http://root.cern.ch/root/html534/ClassIndex.html.

[6] G. Kafka. FluxReader Framework. https://cdcvs.fnal.gov/redmine/projects/fluxreader.

[7] G. Kafka. *FluxReader Framework Wiki*. https://cdcvs.fnal.gov/redmine/projects/fluxreader/wiki.

[8] C. Backhouse. NOνASoft CAFAna Framework. https://cdcvs.fnal.gov/redmine/projects/novaart/repository/show/trunk/CAFAna.

[9] R. Hatcher. *Dk2Nu Package Wiki*, "Converting old files." https://cdcvs.fnal.gov/redmine/projects/dk2nu/wiki#Converting-old-files.