Notes on the `FSRoot` Package

Ryan Mitchell

September 27, 2021   (v3.0+)

Abstract

`FSRoot` is a set of utilities to help manipulate information about different Final States (FS) produced in particle physics experiments. The utilities are built around the CERN `ROOT` framework. This document provides an introduction to `FSRoot`.

Contents

1 Installation and Initial Setup

The FSRoot utilities are built around the ROOT framework, so a working version of ROOT is a prerequisite. Notes on ROOT versions:
– v6.18 and after: FSRoot should work.
– v5.34.09 up to v6.18: FSRoot should work, but without some functionality (e.g. RDataFrame).
– v5.34.08 and before: FSRoot may or may not work.

The FSRoot source code is on GitHub:
https://github.com/remitche66/FSRoot

1. To download the current working version, use git clone:

   ```
   > git clone https://github.com/remitche66/FSRoot.git FSRoot
   ```

   Optionally, to switch to a specific tag (v3.0 for example):

   ```
   > git checkout tags/v3.0
   ```

   Alternatively, download and unpack (tar -xzf) a released version from here:

   ```
   https://github.com/remitche66/FSRoot/releases
   ```

2. Set the location of FSRoot in your login shell script (e.g. .cshrc):

   ```
   setenv FSROOT [path]/FSRoot
   ```

3. To build a static library (which will appear as $FSROOT/lib/libFSRoot.a) use:

   ```
   > cd $FSROOT
   > make
   ```

4. To use FSRoot interactively within a ROOT session, add these lines to your .rootrc file (usually found in your home directory):

   ```
   Unix.*.Root.DynamicPath: .:$(FSROOT):$(ROOTSYS)/lib:
   Unix.*.Root.MacroPath:   .:$(FSROOT):
   Rint.Logon:     $(FSROOT)/rootlogon.FSROOT.C
   Rint.Logoff:    $(FSROOT)/rootlogoff.FSROOT.C
   ```

   The last two lines load and unload FSRoot when you open and close ROOT. Alternatively, you could load FSRoot manually from ROOT:

   ```
   root> .x $FSROOT/rootlogon.FSROOT.C
   ```

   When FSRoot is loaded and compiled, you should see a message saying "Loading the FS-Root Macros" along with output from the compilation. [Note that the rootlogon.FSROOT.C file also sets up default styles, which are not essential. If these conflict with styles you have defined elsewhere, you can tweak or remove these.]

5. It may sometimes be necessary to add the FSRoot directory to library path variables:

   ```
   setenv DYLD_LIBRARY_PATH $DYLD_LIBRARY_PATH\:$FSROOT
   setenv   LD_LIBRARY_PATH   $LD_LIBRARY_PATH\:$FSROOT
   ```

2 Basic Operations: the `FSBasic` directory

2.1 Basic Conventions: the `TTree` format

Some `FSRoot` operations on `ROOT` `TTree` variables assume a particular format for the `TTree`. `FSRoot` generally expects a `TTree` with branches holding numbers (usually `double`).

Four-vectors are assumed to have the form:

> `[AB]EnP[CD]`, `[AB]PxP[CD]`, `[AB]PyP[CD]`, `[AB]PzP[CD]`

where `[CD]` is a particle label (often `1`, `2`, `3`, `2a`, etc., but it could also be `CM` or `BEAM` or anything else) and `[AB]` labels the type of four-vector (for example, it could be `R` for raw or `K` for kinematically fit or `MC` for Monte Carlo, etc.). The `FSRoot` code has no requirements on `[AB]` – it could be anything up to two characters long (or nothing). For `[CD]`, the final state utilities described in Sec. 3 use the numbering convention described in Sec. 3.1. Otherwise, `[CD]` could be anything and any length (but not empty).

For convenience, variables associated with a given particle (like the χ^2 of a track) should use the same particle labels `[CD]` as above. For example, for a pion with four-momentum `EnP5`, `PxP5`, `PyP5`, `PzP5`, the corresponding track χ^2 could be `TkChi2P5`.

To use MC tagging utilities, `FSRoot` uses tree variables named `MCDecayCode1`, `MCDecayCode2`, and `MCExtras` (see Sec. 3.5).

2.2 Basic Histogram Utilities: the `FSHistogram` class

Basic histogram functions are provided by the `FSHistogram` class in the `FSBasic` directory. Like most other functions within `FSRoot`, the functions within the `FSHistogram` class are static member functions, so there is never a need to deal with instances of `FSHistogram`.

The basic functions are `FSHistogram::getTH1F` and `FSHistogram::getTH2F`. Here are examples that can be run from either the `ROOT` command line or from a macro (see `Examples/Intro/intro.C`). The first draws a 1d histogram and the second draws a 2d histogram. The third argument is the variable to plot; the fourth holds the number of bins and bounds.

```
> FSHistogram::getTH1F(fileName,treeName,"Chi2DOF","(60,0.0,6.0)","")->Draw();
> FSHistogram::getTH2F(fileName,treeName,"Chi2DOF:Event",
      "(100,0.0,100.0,100,0.0,6.0)","")->Draw("colz");
```

Cuts can be added in the fifth argument:

```
> FSHistogram::getTH1F(fileName,treeName,"Chi2DOF",
      "(60,0.0,6.0)","Chi2DOF<2.0")->Draw();
```

The variable and cut arguments can contain shortcuts. For example, `MASS(1,2)` is expanded into the total invariant mass of particles 1 and 2, where 1 and 2 are not necessarily numbers (they are the `[CD]` in Sec. 2.1). Characters in front of `MASS` (for example) are prepended to the variable names (the `[AB]` in Sec. 2.1). This is all done in the function `FSTree::expandVariable`, which can also be run independently for testing. A list of all macros can be seen using `FSTree::showDefinedMacros` (for example, `RECOILMASS`, `MOMENTUM`, `COSINE`, etc.). An example for `FSHistogram::getTH1F`:

```
> FSHistogram::getTH1F(fileName,treeName,"MASS(2,4,5)",
    "(100,3.0,3.2)","Chi2DOF<2.0&&abs(MASS(2,4)-1.0)<0.2")->Draw();
```

Examples for `FSTree::expandVariable`:

```
> cout << FSTree::expandVariable("MASS(1,2)+MASS(2,3)") << endl;
  // ==> "(sqrt(pow(((EnP1+EnP2)),2)-pow(((PxP1+PxP2)),2)
                -pow(((PyP1+PyP2)),2)-pow(((PzP1+PzP2)),2)))
        +(sqrt(pow(((EnP2+EnP3)),2)-pow(((PxP2+PxP3)),2)
                -pow(((PyP2+PyP3)),2)-pow(((PzP2+PzP3)),2)))"

> cout << FSTree::expandVariable("ABMASS(1,C)") << endl;
  // ==> "(sqrt(pow(((ABEnP1+ABEnPC)),2)-pow(((ABPxP1+ABPxPC)),2)
                -pow(((ABPyP1+ABPyPC)),2)-pow(((ABPzP1+ABPzPC)),2)))"

> cout << FSTree::expandVariable("RECOILMASS(CM;D,2)") << endl;
  // ==> "(sqrt(pow(((EnPCM)-(EnP2+EnPD)),2)-pow(((PxPCM)-(PxP2+PxPD)),2)
                -pow(((PyPCM)-(PyP2+PyPD)),2)-pow(((PzPCM)-(PzP2+PzPD)),2)))"
```

Histograms are automatically cached so they are made only once. To save histograms at the end of a session, use the function:

```
> FSHistogram::dumpHistogramCache();
```

To read in the cache at the beginning of a session:

```
> FSHistogram::readHistogramCache();
```

To clear a cache from memory during a session:

```
> FSHistogram::clearHistogramCache();
```

To see the contents of the cache:

```
> FSHistogram::showHistogramCache();
```

2.3 Basic Cut Utilities: the `FSCut` class

Additional shortcuts for making plots are available through the `FSCut` class in the `FSBasic` directory. The following example (not using `FSCut`) defines two cuts and then uses them to make a plot, as above:

```
> TString cutCHI2("Chi2DOF<2.0");
> TString cutMASS("abs(MASS(2,4)-1.0)<0.2");
> FSHistogram::getTH1F(fileName,treeName,"MASS(2,4,5)",
    "(100,3.0,3.2)",cutCHI2+"&&"+cutMASS)->Draw();
```

As a shortcut to do the exact same thing, give the cuts names and then implement them using the keyword `CUT()`:

```
> FSCut::defineCut("chi2",cutCHI2);
> FSCut::defineCut("mass",cutMASS);
> FSHistogram::getTH1F(fileName,treeName,"MASS(2,4,5)",
    "(100,3.0,3.2)","CUT(chi2,mass)")->Draw();
```

The `FSCut` class can also be used to define sidebands, which can then be implemented using the keyword `CUTSB()`:

```
> TString cutCHI2SB("Chi2DOF>3.0&&Chi2DOF<7.0");
>   // the relative size of signal to sideband is 0.5
> FSCut::defineCut("chi2",cutCHI2,cutCHI2SB,0.5);
> FSHistogram::getTH1F(fileName,treeName,"MASS(2,4,5)",
    "(100,3.0,3.2)","CUT(mass)&&CUTSB(chi2)")->Draw();
```

If multiple sidebands are used simultaneously, then all combinations of sideband regions are considered and the resulting histogram is a sum of sideband regions with weights determined by the weights for individual regions:

```
> TString cutMASSSB("abs(MASS(2,4)-1.8)<0.6");
>   // the relative size of signal to sideband is 1.0/3.0
> FSCut::defineCut("mass",cutMASS,cutMASSSB,1.0/3.0);
> FSHistogram::getTH1F(fileName,treeName,"MASS(2,4,5)",
    "(100,3.0,3.2)","CUTSB(chi2,mass)")->Draw();
```

2.4 Basic Tree Utilities: the `FSTree` class

The `FSTree` class is also located in the `FSBasic` directory and provides basic utilities to operate on trees. Besides the static `FSTree::expandVariable` member function mentioned in Sec. 2.2, the most useful function is for skimming trees. For example:

```
> FSTree::skimTree(inputFileName, inputTreeName, outputFileName,
     "Chi2DOF<5.0");
```

This will take the tree named `inputTreeName` from the file `inputFileName`, loop over all events and select only those that pass the cut on `Chi2DOF`, then output the selected events to the file named `outputFileName` in a tree with the same name as the input tree. The shortcuts mentioned in Sec 2.2 can also be used here, for example:

```
> FSTree::skimTree(inputFileName, inputTreeName, outputFileName,
     "abs(MASS(1,2)-0.5)<0.1");
```

3 Final State Operations: the `FSMode` directory

3.1 Mode Numbering and Conventions

A "final state" (also called "mode") is made from a combination of: $\Lambda(\to p\pi^-)$, $\bar{\Lambda}(\to \bar{p}\pi^+)$, e^+, e^-, μ^+, μ^-, p, $\bar{p}$, $\eta(\to \gamma\gamma)$, γ, K^+, K^-, $K^0_S(\to \pi^+\pi^-)$, π^+, π^-, $\pi^0(\to \gamma\gamma)$.

As strings, these final state particles are: $\Lambda(\to p\pi^-) \equiv$ `Lambda`, $\bar{\Lambda}(\to \bar{p}\pi^+) \equiv$ `ALambda`, $e^+ \equiv$ `e+`, $e^- \equiv$ `e-`, $\mu^+ \equiv$ `mu+`, $\mu^- \equiv$ `mu-`, $p \equiv$ `p+`, $\bar{p} \equiv$ `p-`, $\eta(\to \gamma\gamma) \equiv$ `eta`, $\gamma \equiv$ `gamma`, $K^+ \equiv$ `K+`, $K^- \equiv$ `K-`, $K^0_S(\to \pi^+\pi^-) \equiv$ `Ks`, $\pi^+ \equiv$ `pi+`, $\pi^- \equiv$ `pi-`, $\pi^0(\to \gamma\gamma) \equiv$ `pi0`.

In a `TTree`, the final state particles should be listed in the order they are given above. The numbering for the [CD] (Sec. 2.1) starts at `1`. For final state particles that decay ($\Lambda(\to p\pi^-) \equiv$ `Lambda`, $\bar{\Lambda}(\to \bar{p}\pi^+) \equiv$ `ALambda`, $\eta(\to \gamma\gamma) \equiv$ `eta`, $K^0_S(\to \pi^+\pi^-) \equiv$ `Ks`, $\pi^0(\to \gamma\gamma) \equiv$ `pi0`), the four-momenta of the decay particles are listed using `a` and `b` in the same order as above. No assumptions about ordering are made for identical particles.

For example, for the final state $\gamma K^+ K^0_S \pi^+ \pi^- \pi^- \pi^0$, the four-momenta are:

```
EnP1  PxP1  PyP1  PzP1      (for the gamma)
EnP2  PxP2  PyP2  PzP2      (for the K+)
EnP3  PxP3  PyP3  PzP3      (for the Ks)
EnP3a PxP3a PyP3a PzP3a     (for the pi+ from Ks)
EnP3b PxP3b PyP3b PzP3b     (for the pi- from Ks)
EnP4  PxP4  PyP4  PzP4      (for the pi+)
EnP5  PxP5  PyP5  PzP5      (for one pi-)
EnP6  PxP6  PyP6  PzP6      (for the other pi-)
EnP7  PxP7  PyP7  PzP7      (for the pi0)
EnP7a PxP7a PyP7a PzP7a     (for one gamma from pi0)
EnP7b PxP7b PyP7b PzP7b     (for the other gamma from pi0)
```

Every final state can be specified in three different ways:

(1) `pair<int,int> modeCode`: a pair of two integers (`modeCode1`, `modeCode2`) that count the number of particles in a decay mode:

```
modeCode1 = abcdefg
  a = # gamma       d = # Ks          g = # pi0
  b = # K+          e = # pi+
  c = # K-          f = # pi-
modeCode2 = abcdefghi
  a = # Lambda      d = # e-          g = # p+
  b = # ALambda     e = # mu+         h = # p-
  c = # e+          f = # mu-         i = # eta
```

(2) `TString modeString`: a string version of `modeCode1` and `modeCode2` in the format "`modeCode2_modeCode1`". It can contain a prefix (for example, `FS` or `EXC` or `INC` or anything longer) that isn't used here, but can help with organization elsewhere.

(3) `TString modeDescription`: a string with a list of space-separated particle names (for example, "K+ K- pi+ pi+ pi- pi-"). The final state particles can appear in any order.

For example, the final state $\gamma K^+ K^0_S \pi^+ \pi^- \pi^- \pi^0$ has `modeCode1 = 1101121`, `modeCode2 = 0`, `modeString = "0_1101121"`, and `modeDescription = "gamma K+ Ks pi+ pi- pi- pi0"`.

3.2   Mode Information: the `FSModeInfo` class

Information about an individual final state is carried by the `FSModeInfo` class. Here are examples of a few of its basic member functions:

```
> FSModeInfo mi("K+ K- K+ K- pi+ pi- pi0 eta");
> mi.modeString();    // ==> "1_220111"
> mi.modeCode1();      // ==> 220111
> mi.modeCode2();      // ==> 1
> mi.modeString("this is the code: (MODECODE1,MODECODE2)");
   // ==> "this is the code: (220111,1)"
> mi.modeString("MODESTRING corresponds to MODEDESCRIPTION");
   // ==> "1_220111 corresponds to  K+  K+  K-  K-  pi+  pi-  pi0  eta "
> mi.modeString("The K- mesons have indices LIST([K-]).");
   // ==> "The K- mesons have indices 4,5."
```

The `FSModeInfo` class also handles particle combinatorics within a given final state through the `modeCombinatorics` member function. This is done using place holders like [pi+], [pi-], [K+], [K+3], [tk+], [pi0], etc., which are replaced by particle indices. While the `modeCombinatorics` function is rarely used explicitly by the user, it can be useful for cross-checking the behavior of the combinatorics. For example:

```
> mi.modeCombinatorics("K+[K+], K-[K-], K+(again)[K+], K+(other one)[K+2]",true);
```

```
// ==>  *******************************
// ==>  *** MODE COMBINATORICS TEST ***
// ==>  *******************************
// ==>       mode = K+  K+  K-  K-  pi+  pi-  pi0  eta
// ==>      input = K+[K+], K-[K-], K+(again)[K+], K+(other one)[K+2]
// ==>   combinations:
// ==>        (1) K+2,K-4,K+(again)2,K+(otherone)3
// ==>        (2) K+2,K-5,K+(again)2,K+(otherone)3
// ==>        (3) K+3,K-4,K+(again)3,K+(otherone)2
// ==>        (4) K+3,K-5,K+(again)3,K+(otherone)2
// ==>  *******************************
```

The `modeCuts` member function uses the results of `modeCombinatorics` to combine combinatorics into a single string using the keywords `AND`, `OR`, `MAX`, `MIN`, and `LIST`. It is also called by the `modeString` member function. Examples:

```
> cout << mi.modeCuts("OR((ABC[K+]+DEF[K-])>0)") << endl;
  // ==>  "((((ABC2+DEF4)>0)||((ABC2+DEF5)>0)||((ABC3+DEF4)>0)||((ABC3+DEF5)>0))"
> cout << mi.modeCuts("AND((ABC[K+]+DEF[K-])>0)") << endl;
  // ==>  "((((ABC2+DEF4)>0)&&((ABC2+DEF5)>0)&&((ABC3+DEF4)>0)&&((ABC3+DEF5)>0))"
> cout << mi.modeCuts("MAX(ABC[K+])") << endl;
  // ==>  "((((ABC[K+])>=(ABC2))&&((ABC[K+])>=(ABC3)))"
> cout << mi.modeCuts("MIN(ABC[K+])") << endl;
  // ==>  "((((ABC[K+])<=(ABC2))&&((ABC[K+])<=(ABC3)))"
> cout << mi.modeCuts("LIST(ABC[K+])") << endl;
  // ==>  "ABC2,ABC3"
```

Note that most of the functionality of the `FSModeInfo` class is rarely used explicitly. It is more often combined with other functions and used in higher-level classes (like `FSModeHistogram`), often producing large strings (used only internally), like:

```
> FSTree::expandVariable(mi.modeCuts("AND(MASS2([K+],[K-])>MASS2([pi+],[K-]))"))
  // ==>  "(((pow(((EnP2+EnP4)),2)-pow(((PxP2+PxP4)),2)
                             -pow(((PyP2+PyP4)),2)-pow(((PzP2+PzP4)),2))>
           (pow(((EnP4+EnP6)),2)-pow(((PxP4+PxP6)),2)
                             -pow(((PyP4+PyP6)),2)-pow(((PzP4+PzP6)),2)))&&
           ((pow(((EnP2+EnP5)),2)-pow(((PxP2+PxP5)),2)
                             -pow(((PyP2+PyP5)),2)-pow(((PzP2+PzP5)),2))>
           (pow(((EnP5+EnP6)),2)-pow(((PxP5+PxP6)),2)
                             -pow(((PyP5+PyP6)),2)-pow(((PzP5+PzP6)),2)))&&
           ((pow(((EnP3+EnP4)),2)-pow(((PxP3+PxP4)),2)
                             -pow(((PyP3+PyP4)),2)-pow(((PzP3+PzP4)),2))>
           (pow(((EnP4+EnP6)),2)-pow(((PxP4+PxP6)),2)
                             -pow(((PyP4+PyP6)),2)-pow(((PzP4+PzP6)),2)))&&
           ((pow(((EnP3+EnP5)),2)-pow(((PxP3+PxP5)),2)
                             -pow(((PyP3+PyP5)),2)-pow(((PzP3+PzP5)),2))>
           (pow(((EnP5+EnP6)),2)-pow(((PxP5+PxP6)),2)
```

8

```
                                      -pow(((PyP5+PyP6)),2)-pow(((PzP5+PzP6)),2))))"
```

The `FSModeInfo` object also contains a list of "categories" that are used by the `FSModeCollection` class (next section) for organization. The `display` method shows information about a given mode, including a list of categories, some of which are added by default. In the following, the first use of `display` will show the default list of categories; the second will also show the added categories:

```
> mi.display();
  // ==>   0    1_220111    K+  K+  K-  K-  pi+  pi-  pi0  eta
  // ==>                     Hadronic  HasGammas  HasEtas  HasKaons
  // ==>                     HasPions  HasPi0s  1Eta4K2Pi1Pi0  8Body
  // ==>                     4Gamma  CODE=1_220111  CODE1=220111  CODE2=1
> mi.addCategory("TEST1");
> mi.addCategory("TEST2");
> mi.display();
  // ==>   0    1_220111    K+  K+  K-  K-  pi+  pi-  pi0  eta
  // ==>                     Hadronic  HasGammas  HasEtas  HasKaons
  // ==>                     HasPions  HasPi0s  1Eta4K2Pi1Pi0  8Body
  // ==>                     4Gamma  CODE=1_220111  CODE1=220111  CODE2=1
  // ==>                     TEST1  TEST2
```

3.3 Collections of Modes: the `FSModeCollection` class

A list of final states (`FSModeInfo` objects) is managed by the `FSModeCollection` class through static member functions. The `FSModeCollection` class uses the categories associated with different final states to produce sublists. The initial list of final states is empty. There are a few methods to add final states to the list. Here is one, where the optional additions to the end of the final state strings add categories:

```
> FSModeCollection::addModeInfo("K+ K-        2K  phi   EXA");
> FSModeCollection::addModeInfo("pi+ pi-      2pi rho   EXB");
> FSModeCollection::addModeInfo("pi+ pi- pi0  3pi omega EXA");
```

The `display` method will show final states associated with different combinations of categories. Boolean operators (and = `&` or `&&`; or = `,` or `||`; not = `!`) and wildcards (`*` and `?`) are allowed:

```
> FSModeCollection::display();              // shows all three
> FSModeCollection::display("2K");          // shows just the 2K mode
> FSModeCollection::display("2K,2pi");      // shows 2K and 2pi
> FSModeCollection::display("EXA");         // shows 2K and 3pi
> FSModeCollection::display("EX*");         // shows all three
> FSModeCollection::display("2K&2pi");      // shows none
> FSModeCollection::display("2K&!2pi");     // shows 2K
> FSModeCollection::display("HasPions");    // shows 2pi,3pi
```

9

```
> FSModeCollection::display("HasPions&!3pi"); // shows 2pi
```

The same list could be created from a text file (the format resembles that for EvtGen; blank lines are ignored; everything after a # is ignored):

```
     ---- file: ThreeModes.modes ----
Decay ThreeModes
  K+ K-          2K  phi    EXA
  pi+ pi-        2pi rho    EXB
  pi+ pi- pi0    3pi omega EXA
Enddecay
     ----------- end file -----------
> FSModeCollection::addModesFromFile("ThreeModes.modes");
```

Nested lists can also be used:

```
     ---- file: NestedModes.modes ----
Decay psi'
  pi+ pi- JPSI      pipiJpsi
  eta JPSI          etaJpsi
Enddecay
Decay JPSI
  mu+ mu-       MM
  e+ e-         EE
Enddecay
     ----------- end file -----------
> FSModeCollection::addModesFromFile("NestedModes.modes");
> FSModeCollection::display("");  // shows all four combinations
> FSModeCollection::display("MM");  // shows two modes with mu+ mu-
> FSModeCollection::display("etaJpsi");  // shows two modes with eta JPSI
> FSModeCollection::display("etaJpsi&MM");  // shows one mode
```

Other methods also operate on combinations of categories:

```
> FSModeCollection::addModesFromFile("NestedModes.modes");
> vector<FSModeInfo*> modes = FSModeCollection::modeVector("MM");
> TString FN("file.MODECODE.root");  TString NT("tree_MODECODE");
> for (unsigned int i = 0; i < modes.size(); i++){
>   cout << "a file name:  " << modes[i]->modeString(FN) << endl;
>   cout << "a tree name:  " << modes[i]->modeString(NT) << endl;
> }
```

3.4   Histograms for Multiple Modes: the `FSModeHistogram` class

The `FSModeHistogram` class combines features from the classes described above to make histograms for multiple final states and to manage the particle combinatorics within those final states.

The primary member function is `FSModeHistogram::getTH1F`, which closely resembles the `FSHistogram::getTH1F` function described in Sec. 2.2. It takes an additional argument that specifies the modes to loop over. In addition to `FSTree::expandVariable`, it also incorporates methods like `ModeInfo::modeString`, `ModeInfo::modeCombinatorics`, `ModeInfo::modeCuts`, and `ModeCollection::modeVector`, all illustrated above.

Here is an example that would plot the mass of the J/ψ given the decay modes specified by the `NestedModes.modes` file shown in the previous section. The first histogram is a sum of two histograms; the second and third histograms are a sum of four histograms.

```
> FSModeCollection::addModesFromFile("NestedModes.modes");
> TString FN("file.MODECODE.root");  TString NT("tree_MODECODE");
>     // plot the total mu+ mu- mass for modes containing muons
> FSModeHistogram::getTH1F(FN,NT,"MM","MASS([mu+],[mu-])","(100,3.0,3.2)",
    "Chi2DOF<5.0")->Draw();
>     // plot the total di-lepton mass for all four modes
> FSModeHistogram::getTH1F(FN,NT,"","MASS([l+],[l-])","(100,3.0,3.2)",
    "Chi2DOF<5.0")->Draw();
>     // plot the same using a cut on track quality
> TString cutTRACK("AND(TrackQualityParticle[tk]>10)");
> FSModeHistogram::getTH1F(FN,NT,"","MASS([l+],[l-])","(100,3.0,3.2)",
    "Chi2DOF<5.0"+AND+cutTRACK)->Draw();
```

To explicitly see how the histograms are constructed, use:

```
> FSControl::DEBUG = true;
```

The histogram caches also work here: methods like `FSHistogram::dumpHistogramCache` work as before.

3.5   Information about MC Components

The `FSModeHistogram` class also includes methods that operate on Monte Carlo truth information. These methods assume trees include variables called `MCDecayCode1`, `MCDecayCode2`, and `MCExtras` that contain truth information about the final state contents. `MCDecayCode1` and `MCDecayCode2` have the same format as `modeCode1` and `modeCode2` described in Sec. 3.1. `MCExtras` has the format:

```
MCExtras = abcd
```

```
    a = # neutrinos        c = # neutrons
    b = # K long           d = # anti-neutrons
```

The `FSModeHistogram::drawMCComponents` method takes the same arguments (file name, tree name, category, etc.) as the `FSModeHistogram::getTH1F` method, but draws the histogram with different colors representing different MC components.

Other methods, like `FSModeHistogram::getMCComponents`, return lists of MC components, where the components are labeled by a single string with format `MCExtras_MCDecayCode2_MCDecayCode1`.

3.6   Operations on Multiple Trees: the `FSModeTree` class

The `FSModeTree` class contains static member functions that operate on multiple trees.

The `FSModeTree::skimTree` method is the same as the `FSTree::skimTree` method (Sec. 2.4), except it takes an argument to specify a combination of categories. To skim trees for the final states listed in `NestedModes.modes`, and to make track quality cuts on tracks, for example:

```
> FSModeCollection::addModesFromFile("NestedModes.modes");
> TString inFN("file.MODECODE.root");  TString NT("tree_MODECODE");
> TString outFN("skim.MODECODE.root");
> TString cutTRACK("AND(TrackQualityParticle[tk]>10)");
> FSModeTree::skimTree(inFN,NT,"EE,MM",outFN,cutTRACK);
```

It often happens in particle physics experiments that a given event can be reconstructed multiple times under different hypotheses. This can happen within a single final state – for example, an event from the final state $K^+K^-\pi^+\pi^-\pi^0$ could be reconstructed once correctly and again one or more times incorrectly by misidentifying pions as kaons and vice versa. It can also happen across several final states – for example, the same event from the $K^+K^-\pi^+\pi^-\pi^0$ final state could also be reconstructed as $K^+K^-\pi^+\pi^-$ by missing the π^0.

The different hypotheses in the above scenarios would lead to different values of the χ^2 from kinematic fits. The `createChi2RankingTree` method uses the `TTree` variables `Run`, `Event`, and `Chi2` to rank hypotheses by χ^2. It does this by creating a friend tree in a new file – the name of the new file name is the same as the old file name except with a ".`Chi2Rank`" appended. The friend tree contains the variables:

```
// Chi2RankCombinations:  number of combinations within a final state
// Chi2Rank:  rank of this combination within a final state
// Chi2RankCombinationsGlobal:  number of combinations in all final states
// Chi2RankGlobal:  rank of this combination in all final states
// Chi2RankVar:  the value of the variable used for ranking
// Chi2RankVarBest:  the best value of Chi2RankVar within a final state
// Chi2RankVarBestGlobal:  the best value of Chi2RankVar in all final states
```

The friend tree is used automatically by `FSModeHistogram` by setting:

12

```
> FSTree::addFriendTree("Chi2Rank");
```

A generalized version of the `createChi2RankingTree` method is called `createRankingTree` and works in the same way. In this version, the variable ranked, its name, and the variables used to group combinations (like `Run` and `Event`) can be customized.

4   Fitting Utilities: the `FSFit` directory

The fitting utilities (contained in the directory `FSFit`) work, but are still under development. See the examples in the directory `Examples/Fitting` for the general idea.

5   Organizing Data and Data Sets: the `FSData` directory

The `FSData` directory contains utilities to manipulate data and data sets. The `FSXYPoint` classes are general, while the `FSEEDataSet` and `FSEEXS` classes are specific to e^+e^- data sets and cross sections, respectively.

Points can be read from files using the `FSXYPointList::addXYPointsFromFile` method. Data sets (for e^+e^-) can be read from files using the `FSEEDataSetList::addDataSetsFromFile` method. Cross sections (for e^+e^-) are read using the `FSEEXSList::addXSFromFile` method. The resulting lists can then be manipulated using "categories" (in a way similar to the way final states are manipulated by the `FSMode` classes). A selection of data sets and reactions from BESIII can be found in the files `BESLUMINOSITIES.txt` and `BESREACTIONS.txt`, respectively.