SELF-FILLING HISTOGRAMS: AN OBJECT ORIENTED ANALYSIS FRAMEWORK

Jenny List*, University of Wuppertal
Benno List†, ETH Zurich

Abstract

We present a set of RooT based histogram classes that allow to define the histogrammed quantity, its weight and the cuts to be applied at the time of the booking. We use lightweight function object classes to define plotted quantities and cut conditions; the ”self-filling” histograms hold pointers to these objects, and evaluate them in a fill method that thus needs no parameters. The use of function objects rather than strings to define plotted quantities and cuts permits error detection at compile rather than run time. Arithmetic, logical and comparison expressions are implemented by operator overloading. A caching mechanism is available to prevent repeated evaluation of complicated expressions. Histograms can be grouped in collections. We apply the visitor pattern to perform operations such as fitting or attribute setting on such a group, without having to extend the collection class each time a new functionality is needed.

INTRODUCTION

Analyses in high-energy physics typically involve the filling of a large number of histograms from n-tuple like structures, where single records correspond to individual events, or to parts of events such as tracks or particle candidates. Often, hundreds or even thousands of histograms are to be filled by an analysis program from thousands or millions of records.

The RooT [1] framework provides powerful classes for the the storage of n-tuple like datasets in the form of RooT-trees, as well as classes for the creation and visualization of histograms. Traditionally, histograms are booked at the beginning of of an analysis program and the filled in a loop that iterates over the data records.

Our analysis framework is based on the RooT histogram classes. We aim for a declarative rather than procedural programming style, where the booking of a histogram, which defines *how* a quantity shall be plotted, is united with the filling, which defines *what* is to be plotted. Thus, in a single function call we completely define the contents of a histogram, which makes the resulting analysis code better maintainable. Since analysis programs often grow from a small nucleus to a large program over the course of months or even years, flexibility and maintainability are of paramount importance to make the lifespan of an analysis code large enough to complete the desired task (e. g. a PhD thesis) before software entropy renders the code useless and requires a complete or partial re-implementation of the program.

RooT already provides a mechanism to produce a histogram with a single statement, namely the `TTree::Draw` method; however, this method also entails a complete iteration over the data set at the time of the histogram definition, which makes it unsuitable for the filling of a large number of histograms, especially when complex algorithms such as jet finding are to be performed on each record prior to plotting.

Our toolkit ois based on the use of light-weight function objects. Such function objects overload the function call operator `operator()` such that it returns the value to be plotted. The “self-filling” histogram stores a pointer to such an object, which allows it to have a `Fill()` method without any arguments. During the histogram booking, i. e. during construction of the histogram object, the pointer to the function object is stored with the histogram; in the loop over the data records, just the argument-less `Fill()` method has to be called for all histograms that shall be filled, which is easily done by registering all such histograms in a collection of histograms.

Additional mechanisms exist for cases where cuts are to be evaluated before entries are made in a histogram, for cases where more than one entry per record is to be made (e. g. one entry for each track in an event), and when one quantity determines into which histogram an entry shall be made, for instance when one wants to plot a quantity differentially in another variable. All this will be discussed in the following. But before that, we shall discuss some considerations which guided our design.

DESIGN CONSIDERATIONS

A basic concept in object–oriented programming is the distribution of responsibility between objects, such that data and functionality is combined in a meaningful way. Applied to histogramming, this means that a histogram should not only administrate bin contents, but should take responsibility for the semantics, i. e. the filling of the bins, as well. Thus, instead of being filled passively, the histogram is just notified when new data is available, whereupon it performs the actual filling steps itself, in collaboration with other (function) objects.

This leads to a declarative approach, where procedural elements such as loops and conditional statements are replaced by iterator objects and logical expressions between function objects. Complicated algorithms that are not read-

* Jenny.Boehme@desy.de
† Benno.List@desy.de

ily formulated in arithmetic expressions (for example a jet parton association) can be encapsulated in small, reusable classes.

A further design consideration has been to the earliest possible detection of programming errors. Therefore we follow the strong typing philosophy of the C++ language and avoid string parsing wherever possible to allow error detection at compile rather than run time.

Whereas conceptually booking and filling are unified in our toolkit, the actual filling of the histograms is deferred until all histograms have been defined and is then performed in a single pass over the data. This reduces I/O load and permits to reuse the results of complicated calculations for the filling of multiple histograms per record.

The efficient handling of sets of similar histograms, as they occur e.g. in differential measurements and data Monte–Carlo comparisons, is of great importance. In our toolkit, such sets can be treated as single entities for booking, filling, and operations such as adding or fitting.

SOME EXAMPLES

In this section we will present a couple of simple examples how our toolkit can be used. We assume that a class `MyTree` exists, which holds the data of one record, either hand–written or generated by RooT's `TTree::MakeClass` method. Histograms are defined in the constructor of a class `AnalysisLoop`, which is derived from the base class `EventLoop` of our toolkit.

The first example shows how to produce a histogram of a quantity `MET` stored in a RooT tree:

```
class AnalysisLoop: public EventLoop {
  public:
    AnalysisLoop (MyTree& tree) {
      FloatFun& METFun = ntfloatfun (tree, &MyTree::MET);
      METHist = new SFH1F("methist", "Missing ET",
                          50, 0., 200., this, METFun);
    }
    void output (TFile *psfile, TFile *rootfile) {
      // do anything (plotting, fitting, etc.) here!
    }
  private:
    SFH1F *METHist;
};
```

Our toolkit provides a class `SFH1F`, derived from RooT's `TH1F`, for self–filling 1–dimensional histograms. As the example shows, the constructor of `SFH1F` takes the same arguments as for a `TH1F` object, plus 2 additional ones: The `FloatFun` object `METHist` defines what is to be plotted in the histogram, namely the missing transverse energy, stored in data member `MET` of the `MyTree` class. The pointer `this` to the `EventLoop` object tells the histogram object where to register itself for later filling.

The corresponding `main` program looks like this:

```
int main(int argc, const char *argv[]) {
  MyTree tree;
  AnalysisLoop theLoop (tree);
  for (int i = 0; i < tree.fChain->GetEntries(); ++i) {
    tree.fChain->GetEntry(i);
    theLoop.loop();
  }
  theLoop.output("out.root","out.ps");
  return 0;
}
```

The method `AnalysisLoop::loop` is inherited from class `EventLoop`, and notifies all registered self–filling objects to fill themselves. The method `AnalysisLoop::output` can contain any post–processing operation desired by the user, in the example the histogram is just plotted and stored in a RooT file.

Next, we add a cut, in this case we require at least one b–tagged jet:

```
IntFun& NBJet = ntintfun (tree, &MyTree::NBJet);
METHist = new SFH1F("methist", "Missing ET", 50, 0, 200,
                    this, METFun, NBJet >= 1);
```

The result of the expression "`NBJet >= 1`" is actually a function object of a subclass of `BaseCut`, which is the base class for function objects returning a `bool` value.

If several entries of the same quantity are stored per data record, iterators take care of treating each entry, as shown in the following example. Note that the arguments of the `SFH1F` constructor don't change. The knowledge of the iterator's existence is passed on by the `FloatFun` object.

```
FillIterator& jetIter =
                    ntfilliterator (tree, &MyTree::NJet);
FloatFun& EtJetsFun =
             ntfloatfun (tree, &MyTree::EtJets, jetIter);
EtjHist = new SFH1F("Etjhist", "Jet Energy", 50, 0, 200,
                    this, EtJetsFun);
```

Weights are defined like abscissa values with FloatFun objects. To plot the energy flow from jets as function of pseudorapidity η, we could therefore write:

```
FloatFun& etaJetsFun =
             ntfloatfun (tree, &MyTree::etaJets, jetIter);
etflowHist = new SFH1F("etflow","Energy flow", 50, -5, 5,
                       this, etaJetsFun, 0, EtJetsFun);
```

Caching of any function object needed several times per event can be achieved like this (`cachedObjects` is a collection of cached objects that belongs to class `EventLoop`):

```
FloatFun& ptJetFun = cached (cachedObjects,
            ntfloatfun(tree, &MyTree::ptJets, jetiter));
```

BASIC ABSTRACTIONS

After these first impressions of how self–filling histograms can be used, we will discuss now the most important ideas which make it work inside.

Registered Objects: A registered object enters itself into the collection given in its constructor. The base class for such collections in the SFH toolkit is called `ROList`. It can notify all its members, not only for filling, but also for any other common operation (see also section on visitors below). The class `EventLoop` is derived from `ROList`,

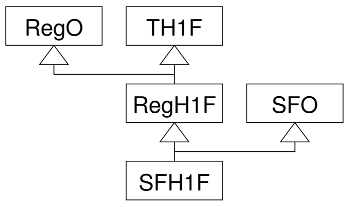

Figure 1: Inheritance tree for class SFH1F.

as are the classes managing groups of histograms. A special case of a RegO is a registered histogram, which is derived from RegO and the desired RooT histogram class, e.g. TH1F, as shown in figure 1.

Self–Filling Objects: Self–fillingness can be added to any registered object by deriving it additionally from the class SFO. The SFO interface requires that any derived class implements an argument–less Fill method. In case of a 1–dimensional self–filling histogram, for which the inheritance diagram is shown in fig 1, the Fill method looks roughly like this:

```
class SFH1F: public RegH1F,  public SFO {
  public:
    void Fill() const {
      if (!xfun) return;
      if (!iter || iter->reset())
        do {
          if (!cut || (*cut)()) {
            float w = wfun ? (*wfun)() : 1.;
            this->TH1F::Fill ((*xfun)(), w);
          }
        } while (iter && iter->next());
    }
    // The different constructors are not shown here
  private:
    FloatFun *xfun, *wfun;
    BaseCut *cut;
    FillIterator *iter;
};
```

The information needed for filling is accessed via the pointers to function objects, which are given to the self–filling histogram object in the constructor.

Function Objects: In the SFH toolkit, several types of function objects are used, which differ by the return value of their operator(). A FloatFun object returns a float, and can be used for filling or weighting, while a BaseCut returns a bool and is used to express cuts. Our framework provides common functions such as sin or sqrt and arithmetic, logical and comparison operators between function objects and numbers, so that complicated expressions can be built from simple objects without the need to write dedicated classes. Comparison operators can be used with FloatFun or IntFun objects to define cuts, as shown in the examples.

The interface of the abstract base class FloatFun is rather minimal:

```
class FloatFun {
  public:
    virtual float operator() () const = 0;
    virtual const FillIterator *getIterator() const
      { return 0; }
    virtual void destroy() { delete this; }
  protected:
    virtual ~FloatFun() {};
};
```

The operator() is the central method, it returns the value of the function object. If multiple values, such as the p_t of several jets, are to be returned, a FillIterator object will be involved, which is returned by getIterator.

Only pointers or references to function objects are stored and passed around, the objects are never copied, otherwise caching would be impossible. In most cases it would be a mistake to create automatic instances of such function objects, which are destroyed automatically when they go out of scope, while they may still be needed. To prevent this mistake, all classes derived from class FloatFun should make their destructor protected, which leads to a compiler error when the user tries to create automatic instances of that class. Then, function objects can only be created on the heap. The method destroy replaces operator delete.

Since FloatFun is an abstract class, generally the user has to provide a derived class that implements operator(). However, a common case, namely the access to a data member in a tree object, is handled by objects of the template class NTFloatFun, which can be generated by using the global template function ntfloatfun (the code for iterator support has been taken out for clarity):

```
template<class MyTree, class VarType = Float_t>
class NTFloatFun: public FloatFun {
  public:
    NTFloatFun (const MyTree& tree,
                VarType MyTree::* p_mem)
      : branchAddress (&(tree.*p_mem)) {}
    virtual float operator() () const
      { return *branchAddress; }
  protected:
    virtual ~NTFloatFun() {}
  private:
    const VarType *branchAddress;
};

template<class MyTree, class T>
NTFloatFun<MyTree, T>& ntfloatfun (const MyTree& tree,
                                   T MyTree::* p_mem)
  { return *new NTFloatFun<MyTree, T> (tree, p_mem); }
```

Additionally, there exists a base class IntFun, which is useful to express conditions on integer quantities, and also serves as base class for two other important concepts that will be discussed next.

Iterators: As already mentioned, iterators are used in cases where potentially more than one entry shall be made in a histogram for a given data record, for instance the energy of several jets per event. The interface of the abstract base class FillIterator is again simple:

```
class FillIterator: public IntFun {
  public:
    virtual int  operator() () const = 0;
    virtual bool next() = 0;
    virtual bool reset() = 0;
    virtual const FillIterator *getIterator() const
      { return this; }
  protected:
    virtual ~FillIterator() {}
};
```

The return value of `operator()` is an index that typically is used as array index; a returned `bool` always indicates whether the index is within the valid range.

In our framework, iterators have two clients that use different parts of the interface: self-filling objects need just `next` and `reset`, whereas `FloatFun` and `BaseCut` objects normally need only the index value provided by `operator()`, and hence often demand just an `IntFun` instead of a `FillIterator`. The method `getIterator` enables self–filling objects to determine whether they have to use an iterator during filling; it also helps to detect the potential error to mix objects that depend on different iterators in the same histogram.

The iterator concept offers a number of interesting possibilities. Similar to the case of `FloatFun` objects, simple iterators that run over a range defined by a tree variable can be generated by a global function `ntfilliterator`. In that case, the `next` method of the iterator will simply increment an index variable by one. More complicated `next` methods are possible that select only certain index values, for instance only numbers that correspond to bottom quarks in a list of generated particles. This is a possibility to sort of incorporate a cut into an iterator, which makes it unnecessary to explicitly add the respective cut everywhere this iterator is used. For instance, in an energy resolution study, one could simply replace a “uds–jet–iterator” by a “b–jet–iterator” to study their different properties.

Another nice technique is to write an `IntFun` subclass that translates the value of an iterator into another index. For instance, a jet parton association can thus be implemented: The iterator value would signify the jet, and the result of the `IntFun` subclass (here called `JetPartonFun`) would be the index of the best matching parton.

Binning functions: The second special case of an `IntFun` is a `BinningFun`, where the returned integer is a bin number. It also provides name and title strings for each bin, which are used to book groups of histograms with the same plotted quantity, as discussed in the next section.

Cached Objects: The caching mechanism makes use of a special list of registered objects, whose members are notified to invalidate themselves when a new data record is read in. Any function object can be cached using the global function `cached`.

Groups of Histograms: A powerful tool are groups histograms which show the same quantity for several non–overlapping data subsets. These occur in data Monte Carlo comparisons as well as in differential measurements. A `SetOfHistograms`, which is differential in one quantity, can be booked by just adding a `BinningFun` object to the argument list of the `TH1F` constructor, whereas a 2–dimensional `MatrixOfHistograms` takes two `BinningFun` objects. Self-filling versions of both types of collections exist. Such a self–filling collection of histograms is considerably more efficient than the same number of single self–filling histograms. Most operations possible for a `TH1`, e.g. addition, multiplication, are also implemented for sets and matrices of histograms. Furthermore, they provide methods to generate summary histograms containing per–histogram–information such as total content or mean value in each bin.

Visitors: An important consequence of grouping similar histogram into sets and matrices is that they can be acted on together very conveniently by applying the visitor pattern [2] to the list. The SFH toolkit comes with many ready to use visitor classes for standard cases like fitting a `TF1` function to each histogram or changing histogram attributes. They all inherit from the abstract base class `HVisitor`, from which the user can derive other visitor classes tailored to the tasks at hand.

CONCLUSIONS

Although originally written to facilitate our own analysis within the H1 experiment with its object–oriented analysis framework, the SFH toolkit was easily used in and proved valuable to several analyses at ATLAS and D0.

The function objects were originally just introduced to make the self–fillingness possible. From this simple starting point, further benefits emerged. The lightweight interface of these classes encourages the encapsulation of algorithms, making the analysis code more modular. The possibility to form expressions of function objects and their internal, thus hidden, collaboration with iterators leads to a very concise way of performing complicated calculations and maintaining a declarative programming style. This has turned out to be far more powerful than we had originally anticipated.

The subclasses of `EventLoop` provide a good way to structure the analysis, because several objects of different such classes can be employed concurrently within the same program. The tree class, which can be conveniently generated by `TTree::MakeClass`, is used just as a store for the data, and does not need to be edited by the user. This makes it easy to deal with changes in the tree structure.

Documentation and further resources for this software are available on the web[3].

ACKNOWLEDGMENTS

We cordially thank family Berger for their generous hospitality in Interlaken.

REFERENCES

[1] R. Brun, F. Rademakers *et al.*, “RooT. An object–oriented analysis framework.” `http://root.cern.ch/`.

[2] E. Gamma *et al.*, “Design Patterns,” Reading, MA (Addison-Wesley) 1995.

[3] B. List, J. List, “SFH: Self–Filling Histograms.” `http://www.desy.de/~blist/sfh/`.