Table of Contents

Creating a Monitoring Module for `AlignmentProducer`

Complete: 🟩

Goal of this page

See Alignment algorithms::Monitoring for an introduction to the `CommonAlignmentMonitor` package. This page explains how to make your own histogram module.

Version

The latest version of the CommonAlignmentMonitor framework, which the documentation below describes, is V01-02-00. This requires CommonAlignmentProducer version V00-30-09. If you can check out this version of Alignment/!CommonAlignmentProducer, please do so and put your new monitor in the HEAD of CommonAlignmentMonitor. If not, note that the new feature added in this version is the inclusion of `const edm::Event&` in the `event()` method. Before version V01-02-00, there was no direct access to the `edm::Event`.

Your new monitor

You will make a new subclass of `AlignmentMonitorBase`, following `AlignmentMoniterTemplate` as a model. The name should conform to `AlignmentMonitorXXX` to keep all histogram modules in the same corner of the namespace. Change directories to `Alignment/CommonAlignmentMonitor/plugins/` and create a new file named `AlignmentMonitorXXX.cc`. (If using the old plugin manager, put the file in `/src/`.) Fill the new file with

```
// -*- C++ -*-
//
// Package:     CommonAlignmentProducer
// Class  :     AlignmentMonitorXXX
//
// Implementation:
//     <Notes on implementation>
//
// Original Author:  Who?
//         Created:  When?
// $Id: SWGuideAlignmentMonitors.txt,v 1.11 2011/05/06 19:19:56 yuriy_2epakhotin_40cern_2ech Exp
//

#include "Alignment/CommonAlignmentMonitor/interface/AlignmentMonitorPluginFactory.h"
#include "Alignment/CommonAlignmentMonitor/interface/AlignmentMonitorBase.h"

class AlignmentMonitorXXX: public AlignmentMonitorBase {
   public:
      AlignmentMonitorXXX(const edm::ParameterSet& cfg): AlignmentMonitorBase(cfg) { };
      ~AlignmentMonitorXXX() {};

      void book();
      void event(const edm::Event &iEvent, const edm::EventSetup &iSetup, const ConstTrajTrackPai
      void afterAlignment(const edm::EventSetup &iSetup);

   private:
};

void AlignmentMonitorXXX::book() {
}
```

```
void AlignmentMonitorXXX::event(const edm::Event &iEvent, const edm::EventSetup &iSetup, const Co
}

void AlignmentMonitorXXX::afterAlignment(const edm::EventSetup &iSetup) {
}

DEFINE_EDM_PLUGIN(AlignmentMonitorPluginFactory, AlignmentMonitorXXX, "AlignmentMonitorXXX");
```

Now it should compile.

Adding histograms

1. Declare your histogram in the `private:` field of your class definition, e.g.

```
   private:
      TH1F *m_hist;
```

The histogram may be anything that descends from `TH1` (that is, any ROOT histogram, including 2D histograms and profile plots).

2. Book it in the `book()` method like this (before `CMSSW_2_1_0`):

```
void AlignmentMonitorXXX::book() {
    m_hist = (TH1F*)(add("/", new TH1F("hist", "normal constructor", 100, 0., 1.)));
}
```

and like this (with `CMSSW_2_1_0` and later):

```
   m_hist = book1D("/", "hist", "normal constructor", 100, 0., 1.);
```

The `add()` function manages the ROOT file, handling iterations and job collection automatically. Let's step through the program flow:

- First, the ROOT constructor (in this case, `TH1F`) does its thing in the normal way.
- This is passed to `add()` or `book1D` with a string `"/"`. This is the internal ROOT-file path for the directory in which you wish to put this histogram.
- If this is a new ROOT file (first iteration), the booking function will create whatever parent directories are needed and place the histogram at that point in the file.
- If this is an old ROOT file (second or later iteration), `add()` will find the old histogram in the file and throw away the new one you just made (before `CMSSW_2_1_0` only).
- The booking function returns a histogram, either the new one or the old one.

Histograms that are reset with each iteration (e.g. residuals) should go into a directory staring with `"/iterN/"` `(a real letter ="N"`, not a number). Other histograms (e.g. something cumulative with iteration, such as X versus iteration number) should not. Here are some examples of setting up directory strings:

- `"/"` put it in the top-level directory
- `"/iterN/"` put it in the top-level of the `iter1`, `iter2`, `iter3`, etc.
- `"/some/other/dir/"` put it in `/some/other/dir` (you don't need to create the `TDirectory` first; CommonAlignmentMonitor does that)
- `"/iterN/some/other/dir/"` put it in `/iter1/some/other/dir`, etc.

Directory names always begin and end with slashes. Do not use the standard ROOT `TFile` and `TDirectory` tools: make all directories through the booking function. Don't make a directory named "/iter2/" or something--- that's just asking for trouble.

The booking function also merges histograms in a collection job after parallel processing. In this case, newly-constructed histograms are saved into the grand total ROOT file, and `add()` uses the names and directories to find all the subjob histograms that need to be merged into the grand total. All of that happens automatically.

You may be wondering, I see a "new", where's the "delete"? (This whole paragraph is pre-=CMSSW_2_1_0= and later only.) It happens at the end of the iteration in AlignmentMonitorBase, but only for histograms that *should* be deleted at the end of each iteration (histograms in the "iterN" directories). That is to say, your plugin creates references to histograms, and the AlignmentMonitor framework "steals" those references; you are no longer responsible for deleting them. So don't delete them, or you'll cause a segmentation fault!

3. Fill your histograms. You may do this in `event()` or `afterAlignment()`, whichever is appropriate.

The `event(const edm::Event &iEvent, const edm::EventSetup &iSetup, const ConstTrajTrackPairCollection &tracks)` function is called in the event loop, and it supplies a list of trajectory-track pairs. Here's how to iterate over them:

```
for (ConstTrajTrackPairCollection::const_iterator iter = tracks.begin();  iter != tracks.end();
    const Trajectory *traj = iter->first;
    const reco::Track *track = iter->second;

    std::vector<TrajectoryMeasurement> trajectoryIntersections = traj->measurements();
    for (std::vector<TrajectoryMeasurement>::const_iterator interIter = trajectoryIntersections.b
        const TrajectoryMeasurement trajectoryIntersection = *interIter;
        const TransientTrackingRecHit *hit = &(*interIter.recHit());
        ...
    }
}
```

Before version V01-02-00, there was no direct access to `edm::Event` and all tracks and trajectories had to be accessed through the `ConstTrajTrackPairCollection`, just as for alignment algorithms. Access to the `edm::Event` was added for supplimentary data, like the `BeamSpot` and run/event numbers.

The `afterAlignment(const edm::EventSetup &iSetup)` function is called at the end of an iteration, after updating the alignable geometry (`AlignableTracker` and `AlignableMuon`) but before updating the production geometry (`TrackerGeometry`, `DTGeometry`, and `CSCGeometry`). You can walk through the `AlignableTracker`, `AlignableMuon`, or `AlignableParameterStore` hierarchies to get the new orientations.

You have access to the current `AlignableTracker` through `pTracker()`, the `AlignableMuon` through `pMuon()`, the `AlignmentParameterStore` through `pStore()`, and the `AlignableNavigator` through `pNavigator()` (all are pointers, NULL if not defined). You can also access the current iteration number through `iteration()` (it starts with 1).

Ntuples or `TTrees`

Before creating thousands of static histograms, it's good to explore the data with an ntuple, at least to get the binning right. The booking function described above places any `TObject` in the file, including `TTrees`, so you can do the following in `book()`:

```
m_tree = (TTree*)(add("/", new TTree("tree", "tree")));
```

or

```
m_tree = directory("/")->make<TTree>("tree", "tree");
```

with `CMSSW_2_1_0` and later. Never put a `TTree` in an `/iterN/` directory. Afterward, add the branches as usual:

```
m_tree->Branch("x", &m_x, "x/F");
m_tree->Branch("y", &m_y, "y/F");
m_tree->Branch("z", &m_z, "z/F");
```

Configuring `AlignmentProducer` to use your new monitoring package

That was described in Alignment algorithms::Monitoring.

Recipes

Histogram code will involve a lot of similar constructions, such as calculating residuals. Here are some code snippets that you can use to quickly make your plots. I encourage others to add to this list (that's what wikis are for)!

Calculating residuals

```
#include "TrackingTools/TrackFitters/interface/TrajectoryStateCombiner.h"
```

and in your loop over hits,

```
const DetId id = hit->geographicalId();
if (hit->isValid()  &&  pNavigator()->detAndSubdetInMap(id)) {
    TrajectoryStateOnSurface combinedTrajInter = tsoscomb.combine(trajectoryIntersection.forwardP
    double residual = combinedTrajInter.localPosition().x() - hit->localPosition().x());
}
```

Booking histograms from selected `Alignables`

In the class declaration,

```
      std::map<Alignable*, TH1F*> m_residuals;
```

and in `book()`,

```
   std::vector<Alignable*> alignables = pStore()->alignables();
   for (std::vector<Alignable*>::const_iterator it = alignables.begin();  it != alignables.end();
      char name[256], title[256];
      sprintf(name,  "xresid%d", (*it)->geomDetId().rawId());
      sprintf(title, "x residual for DetId %d (cm)", (*it)->geomDetId().rawId());

      m_residuals[*it] = (TH1F*)(add("/iterN/", new TH1F(name, title, 100, -5., 5.)));
   }
```

and then in `event()`,

```
const DetId id = hit->geographicalId();
Alignable *alignable = pNavigator()->alignableFromDetId(id);
std::map<Alignable*, TH1F*>::const_iterator search = m_residuals.find(alignable);
while (search == m_residuals.end()  &&  (alignable = alignable->mother())) search = m_residuals.f
if (search != m_residuals.end()) {
    search->second->FIll(residual);
}
```

Review Status

Reviewer/Editor and Date	Comments

Main.pivarski - 07 May 2007	Page created

Last reviewed by: Reviewer