Table of Contents

Alignment algorithms

Complete:

Goal of this page

Description of the configuration and operation common to all algorithms. Details on how to use the different algorithms can be found in the links section.

Known problems

Running on data from many different runs in >= CMSSW_2_2_X

- **Symptom**: Memory exhaustion.
- **Cause**: By default Framework keeps information corresponding to Runs or Lumi sections in memory until the last event is processed (in case of presence of an output module)
- **Solutions/Workarounds** (probably to be combined with each other):
    1. **Hacking shell solution**: Increase the memory budget that your shell allows, e.g. on lxplus the default is 1.5 GB, but you can increase to 3 GB. How to do that is shell dependent. In `zsh/bash` do `ulimit -v 3072000` (number interpreted in kB). Check current limit using `ulimit -v`.
    2. **CMSSW process options** `process.options.fileMode`:
        1. Tell framework not to cache anything, by using `process.options.fileMode = cms.untracked.string('NOMERGE')`. Note that beginRun(), beginLuminosityBlock() and corresponding endXy() are called at file boundaries, even if the same run/luminosity block continues in next/other file. But this should be OK for alignment validation.
        2. Tell framework to cache only luminosity blocks, but rely on events in files being ordered by run: `process.options.fileMode = cms.untracked.string('FULLLUMIMERGE')`
            - Let DBS sort your files by run using this query: `find run, file where dataset = /your/dataset/foo/bah order by run`
            - Remove possible file duplications by hand. Note that this **will not work in case file A contains parts of two runs and file B other parts of the same two runs**.
            - Needs more memory than `NOMERGE` mode.
        3. Tell framework to cache neither runs nor luminosity blocks, but rely on events in files being ordered by run and luminosity block: `process.options.fileMode = cms.untracked.string('MERGE')`
            - Ordering probably hard to achieve.
            - Memory consumption should be in between `NOMERGE` and `FULLLUMIMERGE`.
    3. **CMSSW drop-on-input** feature: DQM monitoring elements are stored in the `Runs` TTree that is stored in the files. These branches consume a significant amount of memory. You can deactivate reading them by setting `inputCommands = cms.untracked.vstring('keep *', 'drop *_MEtoEDMConverter_*_*')` in `PoolSource`.
- See also the following example or more details in sections *The options Parameter Set* and *PoolSource* of SWGuideEDMParametersForModules.

```
process.options = cms.untracked.PSet(
    Rethrow = cms.untracked.vstring("ProductNotFound") # make this exception fatal
#    , fileMode  =  cms.untracked.string('FULLMERGE') # any file order (default): caches all lumi
#    , fileMode  =  cms.untracked.string('FULLLUMIMERGE') # needs files sorted in run, caches lum
#    , fileMode  =  cms.untracked.string('MERGE') # needs files sorted in run and within run in l
    , fileMode  =  cms.untracked.string('NOMERGE') # no ordering needed, but calls endRun/beginRu
    )
process.source = cms.Source("PoolSource",
                            inputCommands = cms.untracked.vstring('keep *', 'drop *_MEtoEDMConver
                            fileNames = cms.untracked.vstring('file1.root', ..., 'fileN.root')
                           )
```

Links to algorithms

- HIP algorithm use in the *tracker*
- HIP algorithm use in the *muon chambers*
- Kalman Alignment Algorithm
- Millepede
- APE estimation tool

Versions

See SWGuideTrackAlignmentVersions for changes between releases and possible tags on top of releases.

Configuration

AlignmentProducer

The class AlignmentProducer has control of the alignment jobs, independent of the algorithm. It has the following responsibilities:

1. Selection of the input geometry (and probable misalignment).
2. Selection of the algorithm and probable monitoring classes to run.
3. Control of the event loop (for iterative algorithms running in one job).
4. Calling the initialisation, termination and event methods of the chosen algorithm.
5. Writing of the alignment constants and their uncertainties.

To allow point 3, i.e. iterations in one go, it is implemented as an `ESProducerLooper` and cannot be put explicitly into the path, but is called after all other modules.

Note that the alignment constants are written as absolute positions and orientations of the GeomDet used in tracking. Therefore changes of the input geometry for different run periods would not make sense and are, in contrast to normal reconstruction jobs, neither taken into account nor foreseen.

The alignment producer expects two collections to be present in the job and passes them on to the algorithm:

1. an association map between a collection of reco::Track and a collection of Trajectory,
2. and a collection of reco::BeamSpot.

The configuration parameters and their defaults in AlignmentProducer_cff.py are given in the following table where *untracked parameters* are marked with a '*':

Name	Type	Description	Default
`maxLoops`	`uint32` *	number of loops on the data	1
`doTracker`	`bool` *	align and provide geometry for the Tracker?	True
`doMuon`	`bool` *		False

		align and provide geometry for DT and CSC?		
`useSurvey`	`bool`	Read survey info from DB	False	see Alignment/SurveyA for an example `Pool`
`applyDbAlignment`	`bool`	(Mis-)alignment from database	False	i.e. default is *ideal g*
`doMisalignmentScenario`	`bool`	apply random scenario?	False	applied on top of ide alignment
`MisalignmentScenario`	`PSet`	scenario to dice	`NoMovementsScenario`	used if `doMisalignm`
`randomShift`	`double`	additional random translational misalignment	0.0	deprecated, use misa reproducability
`randomRotation`	`double`	additional random rotational misalignment	0.0	deprecated, use misa reproducability
`parameterSelectorSimple`	`string`	select rigid-body parameters to apply additional misalignment	'-1'	deprecated, '-1' mear
`ParameterBuilder`	`PSet`	selection of alignment parameters		see section *Selection*
`nFixAlignables`	`int32`	number of alignables to be removed after selection	0	deprecated since dif algorithms (HIP: fix
`tjTkAssociationMapTag`	`InputTag`	source of reco::Track-Trajectory association map for algorithm	"TrackRefitter"	
`beamSpotTag`	`InputTag`	source of the reco::BeamSpot for algorithm	"offlineBeamSpot"	
`algoConfig`	`PSet`	choice and configuration of algorithm	`HIPAlignmentAlgorithm`	must contain algorit system
`monitorConfig`	`PSet`	selection of AlignmentMonitors		see section *Monitori*
`RunRangeSelection`	`VPSet`	selects time dependent alignment parameters	VPSet()	only for Millepede I see ?
`ParameterStore`	`PSet`	configuration of AlignmentParameterStore		for (Kalman) experts
`saveToDB`	`bool`	save alignment constants for Tracker and/or DT/CSC?	False	requires configuratio
`saveApeToDB`	`bool`	save APE for Tracker and/or DT/CSC?	False	requires configuratio
`saveDeformationsToDB`	`bool`	save SurfaceDeformations for Tracker?	False	requires configuratio

Selection of what to align

The selection of objects to be aligned, their AlignmentParameters classes and which of their (usually) six parameters is configured by the `ParameterBuilder` PSet of the AlignmentProducer, passed to the AlignmentParameterBuilder. Its default in AlignmentProducer_cff.py looks like this:

```
ParameterBuilder = cms.PSet(parameterTypes = cms.vstring('Selector,RigidBody'),
                            Selector = cms.PSet(alignParams = cms.vstring('PixelHalfBarrelLayers,
                            )
```

Selection of AlignmentParameter Type

The `ParameterBuilder.parameterTypes` vstring contains strings that are separated into two parts each by a commas. The second part (e.g. `RigidBody`) selects the AligmmentParameters class to choose among those inheriting from their base class and known to the AlignmentParametersFactory. Concrete AligmmentParameters are (mathematically) defined by the derivatives they provide and the way they apply themselves to an Alignable:

- The default given above is the `RigidBody` parameterisation where the first three parameters are the translations in local x, y, z (u,v,w) coordinates and the remaining three the rotations around local x, y, z (alpha, beta, gamma) (see CMS CR-2003/022).
- The `RigidBody4D` parameterisation is identical in u, v, w, alpha, bet and gamma, but provides a 6x4 instead of a 6x2 matrix of derivatives as needed for 4D hits (e.g. segments in the muon system) instead of 2D hits used e.g. in the Tracker.
- The `BowedSurface` parameterisation basically adds three parameters to desccribe non-flat sensor, i.e. the derivatives matrix is 9x2, see this presentation.
- The `TwoBowedSurfaces` is thought for strip tracker modules with two sensors where each sensor is aligned independently with 9 parameters, i.e. the derivatives matrix is 18x2, see the same presentation.
- Further parameterisations could be implemented for internal structures of higher level objects etc.

Selection of Active Parameters of the Choosen AlignmentParameter Type

The first part of `ParameterBuilder.parameterTypes` strings (`Selector`) must match a further PSet within `ParameterBuilder`. This PSet is further passed to the class AlignmentParameterSelector that interpretes all strings in `alignParams` to select Alignables and the active parameters of the choosen AlignmentParameters. This PSet might look like

```
Selector = cms.PSet(alignParams = cms.vstring("TIBRods,101001", "TOBLayers,11111"))
```

Each string in `alignParams` consists of two (or three, cf. below) comma separated parts. The first is a string determining a subdetector and the Alignable level. The second determines which of the parameters are active. The order depends on the AlignmentParameters type (see above), e.g. for `RigidBody` parameters 101001 selects u, w and gamma and deselects v, alpha and beta. Other letters than 0 and 1 also select the parameter, but may be interpreted in a special way, e.g. f, r and c in Millepede.

There are two separate ways to specify the Alignables. Both use as entry point the method `unsigned int AlignmentParameterSelector::addSelection(const std::string &nameInput, const std::vector &paramSel)`.

1. The first method, valid for Tracker and Muon, is to use pre-defined strings, e.g. `BarrelRods`, `TIBDets`, `PXECLayers` or `MuonCSCChambers`, which can be looked up in the source code. If an unknown string is found, and exception is thrown.
2. The second (Tracker only) method uses the tracker hierarchy. All hierarchy levels can be accessed generically: If a string starts with `Tracker`, this is stripped off. The remaining part should be

something like `TIBHalfShell`. All possible strings can be taken from the source code by prepending "Tracker". If the string (after stripping off "Tracker") is unkown, an exception is thrown, telling all known (stripped) strings. Here `TIBModuleUnit` means the 1D modules that form the 2D AlignableDets (and analogous for TID, TOB and TEC). In case of TOB and TEC with both 1D and 2D modules, `TOB/TECModuleUnit` means all 1D units and `TOB/TECModule` means all modules. For the pixel `TPB/TPEModule` and `TPB/TPEModuleUnit` are equivalent.

For both methods there are additional ways to further restrict the selection of Alignables given by the first part (before first comma) of the string:

1. Special string extensions:
   `LayersNM`
   : If the first string *ends* with `Layers` followed by two digits, e.g. `24`, only the layers from the first to the second digit are selected. The numbering starts at 1. E.g. `TIBRodsLayers24` selects TIB rods of layers 2, 3 and 4, `TOBDetsLayers66` selects TOB Dets of the sixth (and last) layer. For TID and TEC selects wheels and for PixelEndcap disks.

   `SS` or `DS` (*valid only for TIB and TOB, ignored elsewhere, see below how to select TID/TEC 1D/2D-dets geometrically*)
   : If the first string part *contains* at any place `DS`, only Dets/Rods/Layers with stereo (double sided) modules are selected, whereas `SS` at any place selects single sided modules (based on the layer). `SS` and `DS` are removed from the string before it is further interpreted.
2. The 1D modules that are part of an AlignableDet in the 2D strip modules can be separately selected by the selection strings `TrackerTIBModuleUnitRphi` and `TrackerTIBModuleUnitStereo` and analog for TID, TOB and TEC. (The previous `LayersNM` selection can be give on top of that.)
3. Geometrical selection in eta, phi, r, x, y and z by use of extra PSets: If an item in `alignParams` contains a third comma-separated part, a PSet with that name is expected within `Selector`, defining arrays for ranges in eta, phi, r, x, y and z. These arrays must have an even number of entries and the alignable positions are required to fulfil ('x'Range[0] <= 'x' < 'x'Range[1]) || ('x'Range[2] <= 'x' < 'x'Range[3]) || etc. Note that currently the alignable positions which have to fulfil the selection are the nominal positions (or those read from the database). As of CMSSW version 3_10 the geometrical selection has been extended by the possibility to also select by detector IDs and detector ID ranges ('detId'Range[0] <= 'detId' <= 'detId'Range[1]). Here is an example that e.g. assigns different parameters for 1D and 2D modules in TEC and different parameters to pixel endcap disk 1 modules :

```
PSet Selector = {
  vstring alignParams = { "TrackerTIBString,101001,aSelection", "TECDets,101001,endCapSS","
  PSet aSelection = { # name defined above
    vdouble etaRanges = {-2., -0.4, 0., 1.5} # either (-2 <= eta < -0.4) or (0 <= eta < 1.5
    vdouble phiRanges = {1.5, -0.5}          # a phi-slice covering the +-pi sign flip
    vdouble rRanges = {} # empty arrays mean no restriction in that variable
    vdouble xRanges = {}
    vdouble yRanges = {}
    vdouble zRanges = {}
  }
  PSet endCapSS = { # valid for TID and TEC
    vdouble etaRanges = {}
    vdouble phiRanges = {}
    vdouble rRanges = {40., 60., 75., 999.}
    vdouble zRanges = {}
    vdouble xRanges = {}
    vdouble yRanges = {}
  }
  PSet endCapDS = { # valid for TID and TEC
    vdouble etaRanges = {}
    vdouble phiRanges = {}
    vdouble rRanges = {0., 40., 60., 75.}
    vdouble zRanges = {}
    vdouble xRanges = {}
    vdouble yRanges = {}
  }
```

```
    PSet detIdSelection = {
      vdouble etaRanges = {}
      vdouble phiRanges = {}
      vdouble rRanges = {}
      vdouble zRanges = {}
      vdouble xRanges = {}
      vdouble yRanges = {}
      vint detIds = {}
      vint detIdRanges = {}
      vint excludedDetIds = {}
      vint excludedDetIdRanges = {}
      PSet pxbDetId = {
        vint ladderRanges = {}
        vint layerRanges = {}
        vint moduleRanges = {}
      }
      PSet pxfDetId = {
        vint bladeRanges = {}
        vint diskRanges = { 1, 1 }  # 1 <= disk number <= 1
        vint moduleRanges = {}
        vint panelRanges = {}
        vint sideRanges = {}
      }
      PSet tibDetId = {
        vint layerRanges = {}
        vint moduleRanges = {}
        vint stringRanges = {}
        vint sideRanges = {}
      }
      PSet tidDetId = {
        vint diskRanges = {}
        vint moduleRanges = {}
        vint ringRanges = {}
        vint sideRanges = {}
      }
      PSet tobDetId = {
        vint layerRanges = {}
        vint moduleRanges = {}
        vint sideRanges = {}
        vint rodRanges = {}
      }
      PSet tecDetId = {
        vint wheelRanges = {}
        vint petalRanges = {}
        vint moduleRanges = {}
        vint ringRanges = {}
        vint sideRanges = {}
      }
    }
  }
```

Monitoring

The `AlignmentProducer` maintains a list of histogram modules implemented in the `CommonAlignmentMonitor` package. The idea is that each group working on alignment can create a module, fill it with histograms, and point `AlignmentProducer` to it. As `AlignmentProducer` runs the selected algorithm, it also calls the histogram module, filling plots. This "on-board monitoring" has access to the same data as the alignment algorithm itself, so it can create histograms/profile plots from

- the selected set of alignables and parameters,
- every iteration, and
- the aligned geometry after each iteration step.

The monitoring package manages the output ROOT file, including a directory structure based on iteration number and collecting and merging histograms from distributed jobs.

Configuring Alignment Monitors before `CMSSW_2_1_X`

The default configuration is to use no monitors: in `Alignment/CommonAlignmentProducer/data/AlignmentProducer.cff`, the relevant lines are

```
PSet monitorConfig = {
    untracked vstring monitors = {}
}
```

In your configuration file, you can replace the whole `PSet`. A typical configuration is:

```
replace AlignmentProducer.monitorConfig = {
    untracked vstring monitors = {"AlignmentMonitorHIP"}
    untracked PSet AlignmentMonitorHIP = {
        string outpath = "./"
        string outfile = "histograms.root"

        bool collectorActive = false
        int32 collectorNJobs = 0
        string collectorPath = "./"
    }
}
```

The `vstring monitors` is a list of selected monitors, only one element long in the above (typical case). A `PSet` for each follows; this `PSet` is passed to the alignment monitor for parsing. The above example contains the minimum parameters:

- an `outpath` for the output ROOT file (must end in slash)
- an `outfile` for the output ROOT file
- `collectorActive` is true if this invocation of `AlignmentProducer` is collecting and merging intermediate results. This parameter should be true if and only if `*AlignmentAlgorithm.collectorActive` is true.
- `collectorNJobs` is the number of ROOT files to merge
- `collectorPath` is the base directory for all intermediate alignment results, including histograms (must end in slash). The full directory hierarchy is `[collectorPath]/job[N]/[outfile].`

Configuring Alignment Monitors in `CMSSW_2_1_X` and beyond

The default configuration is still to have no monitors, with the same syntax.

To add a monitor, you now need a TFileService

```
process.TFileService = cms.Service("TFileService", fileName = cms.string('file.root'))
```

which puts all your histograms from different modules into the same file.

Monitor packages no longer have `outpath`, `outfile`, `collectorActive`, `collectorNJobs`, or `collectorPath` parameters, so only those parameters directly needed by the monitoring package need to be specified (e.g. how to book the histograms, etc.). You still need a PSet, even if it's empty.

```
process.AlignmentProducer.monitorConfig = cms.PSet(monitors = cms.untracked.vstring("AlignmentMon
                                                   AlignmentMonitorHIP = cms.untracked.PSet()
                                                   )
```

Changes in capabilities

These changes were motivated by problems interfacing with new versions of ROOT. `TFileService` properly handles the new ROOT interface, so we made CommonAlignmentMonitor a shell around that. While it is convenient having histograms from different modules in the same output file, we have lost the following features:

- CommonAlignmentMonitor can only put histograms from multiple iterations into the same file if all of those iterations were performed in the same cmsRun process, using AlignmentProducer as a looper. If each of your iterations involves a separate cmsRun invocation, for example because you are parallel-processing, each iteration will be in a directory named `iter1`.
- CommonAlignmentMonitor no longer merges histograms in a collector step (combining histograms at the end of an iteration, from different sub-jobs in a parallel-processing step). To merge histograms in a file created by `TFileService`, see the documentation.

However, the following is now possible:

- Monitoring packages can now be run as an analyzer outside of AlignmentProducer, for example in an offline validation process. The analyzer is configured like this:

```
module AlignmentMonitorAsAnalyzer = AlignmentMonitorAsAnalyzer {
    InputTag tjTkAssociationMapTag = # same as you would have passed to AlignmentProducer
    PSet ParameterStore = # same as you would have passed to AlignmentProducer
    untracked vstring monitors = {"AlignmentMonitorHIP"}
    untracked PSet AlignmentMonitorHIP = {}
}
```

This also paves the way to turn monitoring packages into DQM--- the DQM source would be another analyzer with much the same interface.

Creating an alignment monitor

The full documentation on creating a monitoring module is located at SWGuideAlignmentMonitors.

Refitting

To obtain the `TrajectoryStateOnSurface` (TSOS) for each hit, the tracks have to be refitted using the TrackRefitter. During this refit constraints can be applied, see SWGuideRefitterWithConstraints. Note the different refitter settings for collision, cosmic and beam halo tracks. From CMSSW_2_0_0 on, the final tracks do not anymore contain `SiStripMatchedRecHit2D`, these are replaced in the final fit by their constituents.

ReferenceTrajectory

ReferenceTrajectorises are used in Kalman and Millepede algorithms. As long as no documentation is available here, please refer to this python snippet in CVS that predefines the following trajectory factory PSets to configure the trajectories below: `ReferenceTrajectoryFactory`, `BzeroReferenceTrajectoryFactory`, `DualReferenceTrajectoryFactory`, `DualBzeroReferenceTrajectoryFactory`, `TwoBodyDecayTrajectoryFactory`, `CombinedTrajectoryFactory`, `BwdBzeroReferenceTrajectoryFactory`, `CombinedFwdBwdBzeroTrajectoryFactory`, `CombinedFwdBwdDualBzeroTrajectoryFactory`, `BwdReferenceTrajectoryFactory`, `CombinedFwdBwdTrajectoryFactory`, `CombinedFwdBwdDualTrajectoryFactory` and `DualKalmanFactory`.

ReferenceTrajectory

Default trajectory with reference helix from first or last hit.

BzeroReferenceTrajectory

As ReferenceTrajectory, but using only four parameters, skipping the momentum (for B = 0T).

DualTrajectory and DualBzeroTrajectory

As ReferenceTrajectory and BzeroReferenceTrajectory, respectively, but with reference helix from point in the middle.

TwoBodyDecayTrajectory

Two-track trajectory providing mass and vertex constraints.

CombinedTrajectoryFactory

To combine several factories.

DualKalmanTrajectory

Trajectory with residuals from Kalman fit, but only 5 track parameters taken from DualTrajectory. In experimental status and mathematically not well founded.

Review Status

Editor/Reviewer and date	Comments
Main.cklae - 16 Mar 2007	Added links to algorithms
JennyWilliams - 28 Mar 2007	editing for inclusion in SWGuide
JennyWilliams - 24 Apr 2007	added millipede link
FredericRonga - 25 Apr 2007	Re-organize page
FredericRonga - 02 May 2007	Add Known Problems
FredericRonga - 01 Nov 2007	Added 1_6_X backport of new tracker hierarchy
GeroFlucke - 17 Apr 2009	Add more configuration details

Responsible: AndreasMussgiller
Last reviewed by: GeroFlucke

This topic: CMSPublic > SWGuideAlignmentAlgorithms
Topic revision: r43 - 2012-07-27 - GeroFlucke

Copyright &© 2008-2020 by the contributing authors. All material on this collaboration platform is the property of the contributing authors.
or Ideas, requests, problems regarding TWiki? use Discourse or Send feedback