Table of Contents

Documentation for Onia2MuMuPAT

The term Onia2MuMuPAT can be used to name two objects:

- the full chain from AOD/RECO to PAT di-muon candidates using the standard configuration available in the python files under **`HeavyFlavorAnalysis/Onia2MuMu`**
- the individual CMSSW Module that produces the PAT di-muon candidate objects starting from PAT muons.

This page will describe both.

Purpose of Onia2MuMuPAT

The aim of Onia2MuMuPAT is to provide to the CMS analysis community:

- A **flexible module** that can be used to make PAT di-muon candidate objects starting from PAT muons filling in some analysis level variables which would be tricky or tedious for everybody to compute by themselves
- A **standard configuration** to produce PAT di-muon candidates using the above mentioned module, so that it can be saved in group skims and used consistently among the different analyzes.

The Onia2MuMuPAT output under the standard configuration has a physics content similar to the previously used Onia2MuMu ntuples, which will become obsolete once Onia2MuMuPAT will be fully validated, documented and accepted within the group.

Code and tags

The CMSSW module for Onia2MuMuPAT is defined in the **`HeavyFlavorAnalysis/Onia2MuMu`** package (files **`Onia2MuMuPAT.h`**, **`Onia2MuMuPAT.cc`** , **`onia2MuMuPAT_cfi.py`**)

The python code that defines the standard Onia2MuMuPAT workflow is **`onia2MuMuPAT_cff.py`** .

The current working tag is **`V00-10-00`** (`CMSSW_3_5_6` and later releases; it requires you to check out also this tag `MuonAnalysis/MuonAssociators V01-06-00` to do the association to L1 objects)
The previous working tag is **`V00-07-03`** (`CMSSW_3_4_2` release; requires also tag `MuonAnalysis/MuonAssociators V01-03-00`)

Configuration files

A simple configuration file to run Onia2MuMuPAT is provided under
`HeavyFlavorAnalysis/Onia2MuMu/test/onia2MuMuPAT_cfg.py`

This configuration file:

1. loads all the Onia2MuMu modules and sequences from `onia2MuMuPAT_cff.py`
2. defines a filter to select events with at least one di-muon candidate, accepting global or tracker muons.
3. defines a path that produces the `pat::Muons`, runs Onia2MuMuPAT and then the event selector
4. defines an output module that writes out just the pat muons, pat di-muons. It will by write out only events that pass the filter.

Parameter settings

The parameters of the Onia2MuMuPAT module are:

- **muons**: a `cms.InputTag` to the input collection of `pat::Muons` (`patMuons` in the standard configuration)
- **beamSpotTag**, **primaryVertexTag** : inputs for computing the vertex and lifetime-related information (standard values are `offlineBeamSpot`, `offlinePrimaryVertices`)
- **lowerPuritySelection** : a selection that **BOTH** muons must pass for the candidate to be accepted. The cut is specified as a `cms.string`, using the CMS.PhysicsTools string cut parser.
- **higherPuritySelection** : a selection that **AT LEAST ONE** muon must pass for the candidate to be accepted.
- **addCommonVertex**, **addMuonlessPrimaryVertex**, **addMCTruth**, **resolvePileUpAmbiguity** : toggles the addition of analysis-level information into the PAT di-muons. In the standard configuration these values are set to `cms.bool(True)` except for `addMCTruth` (true on mc, false on data) and `resolvePileUpAmbiguity` (set to false, i.e. not needed) the description of these features is provided below in the next section.

In the full Onia2MuMuPAT workflow there are also other parameters that are worth tuning:

- preselection applied to all PAT Muons (parameter `cut` in the module `patMuons`); by default is empty, but it can be set to any string cut.
- parameters of the matching to generator level particles; the module is called **muonMatch**.
- parameters of the PATMuonProducer, e.g. to add extra user data: the module is called **patMuonsWithoutTrigger**.

Output format

A sample analyzer for the J/psi analysis, showing how to access most of the information stored in the PAT-tuple, is provided in the same package **HeavyFlavorAnalysis/Onia2MuMu**: (files **JPsiAnalyzerPAT.cc**, **jpsianalyzerpat_cfg.py**).

Details on the output format follow.

PAT di-muons

The output of the Onia2MuMuPAT producer is a collection of **pat::CompositeCandidate** objects that have as daughters the two **pat::Muons** .

The four-momentum of the di-muon is defined from the 4-momenta of the `reco::Muon` object, which is computed from the inner tracker track even if a global track is available (for all momenta below 200 GeV).

The two daughter muons can be accessed through the `daughter(i)` method:

```
const pat::Muon *mu1 = dynamic_cast<const pat::Muon *>(jpsi.daughter(0));
const pat::Muon *mu2 = dynamic_cast<const pat::Muon *>(jpsi.daughter(1));
```

In the standard configuration, both daughters are guaranteed to be `pat::Muons`. The daughters of the di-muon object are **not** sorted by quality; they happen to be sorted by p_T, but you shouldn't rely on this feature as it might change in the future.

More information about the content of the **pat::Muons** when they're produced from the standard Onia2MuMu configuration is provided below.

Vertex-related variables

Muon-less primary vertex

By default, for each di-muon candidate the module tries to reconstruct the primary vertex excluding the two muons. If the reconstruction is successful, the corresponding `reco::Vertex` is added to the di-muon candidate, and can be retrieved as

```
const reco::Vertex *muonlessPV = jpsi.userData<reco::Vertex>("muonlessPV");
```

The method will return a null pointer if such vertex is not available, which can happen if:

- no non-fake primary vertex can be found after removing the two muons: this is the only possibility if you're using the standard configuration.
- at least one of the two muon doesn't have an inner tracker track: can happen if Onia2MuMuPAT is configured to accept also standalone muons (not the default)
- the feature was turned off by setting the `addMuonlessPrimaryVertex` parameter to `False` (to save disk space, or if the necessary AOD info is not available in the input file)

The muon-less primary vertex is computed using exactly the same tracks, beam spot and fitter configuration as for the input primary vertices (parameter `primaryVertexTag`).

Choice of primary vertex

By default, the first entry in the primary vertex collection (either normal or muonless) is taken as the primary vertex of the event: this corresponds to the vertex with the highest sumpT of the tracks forming it. If one want to select the vertex with the minimum distance in z from the J/psi vertex (see below) the option `resolvePileUpAmbiguity` must be set to "true". In case of no pile-up, this option is clearly not needed.

Di-muon vertex

If both muons have a tracker track, as it always happen in the standard configuration, the module will use the standard KalmanVertexFitter to compute a common vertex for the two muons, and save the associated information in the `pat::CompositeCandidate`.
The information on this vertex can be retrieved as:

```
float normalizedChi2 = jpsi.userFloat("vNChi2");
float probability = jpsi.userFloat("vProb");
const reco::Vertex *vertex= jpsi.userData<reco::Vertex>("commonVertex");
```

If the vertex is not found, chi2 and probability will return `-1` , and the pointer to the vertex object will be null. If one needs to save even more disk space, one can decide to save only the chi2 and probability but not the full vertex object by setting the parameter `addCommonVertex` to `False`.

Pseudo-proper decay length

If the di-muon vertex fit is successful, information about the pseudo-proper decay length is computed an added to the jpsi:

- **`jpsi.userFloat("cosAlpha")`** : cosine of the angle in the transverse plane between (v_{mm} - v_P) and the di-muon momentum. v_P is the position of the muon-less primary vertex if it's available, otherwise it's the position of the normal primary vertex (from the input collection specified through parameter `primaryVertexTag`).
- **`jpsi.userFloat("ppdlPV")`** : pseudo proper decay length, assuming J/Psi mass, $|v_{mm} - v_P| * \cos(\ ) * M / p_T$.
- **`jpsi.userFloat("ppdlBS")`** : pseudo proper decay length using the beam spot instead of the primary vertex.

If the normal primary vertex reconstruction fails, the collection of primary vertices will contain a fake vertex obtained from the beam spot, and such vertex will be used by this module.

Simulation truth information

On simulated samples, if both daughter `pat::Muons` are matched to generator level particles decaying from the same generator level mother, additional information is included in the pat di-muon.

The information can be accessed like in this example:

```
reco::GenParticleRef trueDiMuon = jpsi.genParticleRef();
if (trueDiMuon.isNonnull()) {
    const reco::GenParticle & mcjpsi = *trueDiMuon;
    int dimuonParentPdgId = jpsi.userInt("momPDGId"); // id of the mother of the J/Psi
    float trueDecayLength  = jpsi.userFloat("ppdlTrue"); // pseudo-proper decay length from mc tr
}
```

PAT Muons

The **`pat::Muons`** from Onia2MuMuPAT have the following features:

- They include CaloMuons. Note that the method `isCaloMuon()` will return a true value also for global, tracker or standalone muons that have energy deposit compatible with a MIP.
- Anything which is available in the `reco::Muon` object. In particular, the features most relevant to low mass di-muons are:
  - the variables of the tracker track: e.g. `muon->track()->pt()`, `muon->track()->normalizedChi2()`, the step of iterative tracking (`muon->track()->algo()`), and the quality flags (`if (muon->track()->quality(reco::Track::highPurity)) { ... }`)
  - the hit pattern of the tracker track, `muon->track()->hitPattern()`, to query for the number of strip hits (mono and 2D), pixel hits, and so on. One can also request information about the number of active layers without measurement available before the first hit or after the last one (methods `trackerExpectedHitsInner()` and `trackerExpectedHitsOuter()` of the `reco::Track` object), which can be useful to tag non-prompt muons.
  - the calorimeter compatibility (method `caloCompatibility()`)
  - the global track, if available (methods `outerTrack()` and `globalTrack()`)
- a shortcut to the Muon ID information, through the method `mu.muonID("name")`
- matching with generator level muons, using PAT defaults (match against generator muons of status 1, with $\Delta R < 0.5$ and $\Delta p_T/p_T < 0.5$, requiring the correct charge and with match arbitration)
- match with trigger level info (see below)

Compared to usal `pat::Muons` from the default PAT configuration used in higher energy analyses, some unneeded features have been disabled to save disk space: the high energy muon refits, the user-specified isolation and the user-specified isodeposits.

Trigger matching information

Matching with trigger level information is performed at the level of the PAT muon.

- Matching with L3 objects: PAT defaults, $\Delta R < 0.5$, $\Delta p_T/p_T < 0.5$ between the reco. muon momentum at vertex and the L3 momentum at vertex; simple ambiguity resolution (highest p_T reco. muon gets the first pick, next one can match anything except the one already taken - if any - and so on...)
- Matching with L2 objects: done as above, although perhaps the cuts should be retuned (especially $\Delta p_T/p_T$)

- Matching with L1 objects: reco. muon tracker track is propagated to muon station 2, taking into account the expected energy loss; a matching is then performed requiring $\Delta R < 0.3$ between the L1 (,) and the impact point of the extrapolated reco. track. Ambiguity resolution performed just like for L3.

The information can be accessed from the `pat::Muon` objects by using **`triggerObjectMatchesByFilter("last_filter_name")`** that will return the 4-vector of the corresponding trigger objects (as `pat::TriggerObjectStandAlone`), or an empty collection if the muon didn't fire the trigger. **DO NOT USE** `triggerObjectMatchesByPath("path_name")` as it can give false positive matches with the current matching code .

This table contains the list of triggers currently available in Onia2MuMuPAT, and the filter name to be used with `triggerObjectMatchesByFilter`.

Trigger	Filter	Notes
L1MuOpen	**`hltL1MuOpenL1Filtered0`**	
L2Mu0	**`hltL2Mu0L2Filtered0`**	GR menu only, not 8E29
L2Mu3	**`hltSingleMu3L2Filtered3`**	GR menu only, not 8E29
Mu3	**`hltSingleMu3L3Filtered3`**	
Mu5	**`hltSingleMu5L3Filtered5`**	
L1DoubleMuOpen	**`hltDoubleMuLevel1PathL1OpenFiltered`**	
L2DoubleMu0	**`hltDiMuonL2PreFiltered0`**	GR menu only, not 8E29
DoubeMu0	**`hltDiMuonL3PreFiltered0`**	
DoubleMu3	**`hltDiMuonL3PreFiltered`**	
Mu0_L1MuOpen	**`hltMu0L1MuOpenL3Filtered0`** **`hltDoubleMuLevel1PathL1OpenFiltered`**	match to the L3 `Mu0` match to **both** the `L1MuOpen` (See below)
Mu3_L1MuOpen	**`hltMu3L1MuOpenL3Filtered3`** **`hltDoubleMuLevel1PathL1OpenFiltered`**	match to the L3 `Mu3` match to **both** the `L1MuOpen` (See below)
Mu5_L1MuOpen	**`hltMu5L1MuOpenL3Filtered5`** **`hltDoubleMuLevel1PathL1OpenFiltered`**	match to the L3 `Mu5` match to **both** the `L1MuOpen` (See below)
Mu0_L2Mu0	**`hltMu0L2Mu0L3Filtered0`** **`hltL2Mu0L2Filtered0`**	match to the L3 `Mu0` (GR menu only, not 8E29) match to **both** L2s (same as `L2DoubleMu0`)
Mu3_L2Mu0	**`hltMu3L2Mu0L3Filtered3`** **`hltL2Mu0L2Filtered0`**	match to the L3 `Mu3` (GR menu only, not 8E29) match to **both** L2s (same as `L2DoubleMu0`)
Mu5_L2Mu0	**`hltMu5L2Mu0L3Filtered5`** **`hltL2Mu0L2Filtered0`**	match to the L3 `Mu5` (GR menu only, not 8E29) match to **both** L2s (same as `L2DoubleMu0`)
Mu0_Track0_JPsi	**`hltMu0TrackJpsiTrackMassFiltered`**	*See below*
Mu3_Track0_JPsi	**`hltMu3TrackJpsiTrackMassFiltered`**	*See below*
Mu5_Track0_JPsi	**`hltMu5TrackJpsiTrackMassFiltered`**	*See below*

Note about `MuX_Track0_JPsi` paths:
For these triggers, the `triggerObjectMatchesByFilter` will return a collection containing both the matches to the L3 muons and the matches to the CKF tracks. In order to see to exactly what trigger was matched, use this code

```
bool matchedMu3 = false, matchedTrack = false;
pat::TriggerObjectStandAloneCollection mu0tkMatch = mu->triggerObjectMatchesByFilter("hltMu0Track
for (unsigned k = 0; k < mu0tkMatch.size(); ++k) {
   if (mu0tkMatch[k].collection() == "hltL3MuonCandidates::HLT") matchedMu3 = true;
   if (mu0tkMatch[k].collection() == "hltMuTrackJpsiCtfTrackCands::HLT") matchedTrack = true;
}
```

Note about `MuX_L1MuOpen` matches to L1:
The correct trigger filter for matching the two L1s of `Mu<X>_L1MuOpen` is **`hltMu<X>L1MuOpenL1Filtered0`**. However, this does not work in `CMSSW_3_5_6` which is used in Spring10 production, because of a misconfiguration in the trigger table used there. Luckily `L1DoubleMuOpen` has the same L1s as `MuX_L1MuOpen`, so using the filter labels for `L1DoubleMuOpen` works (at least as long as `L1DoubleMuOpen` is unprescaled)

Note about L1 matching in general:
The matching with L1 requires for the muon to be successfully propagated to station 2, and this can fail if the muon is too soft; normally, if the muon can't be propagated then we don't expect it to fire any trigger, so this is not a problem. However, if you want, you can check explicitly if the propagation succeeded or not (this can be also useful if you're interested in comparing the L1 (,) with the impact point of the propagated reco. track):

```
pat::TriggerObjectStandAloneCollection propagated = mu->triggerObjectMatchesByFilter("propagatedT
if (propagated.empty()) {
   std::cout << "Muon didn't reach station 2, according to CMS.SteppingHelixPropagator" << std::e
} else {
   std::cout << "Propagation succeeeded; eta = " << propagated[0].eta() << ", phi = " << propagat
}
```

Example macro to read trigger matching:
An example macro that prints out the triggers for a Onia2MuMuPAT file is HeavyFlavorAnalysis/Onia2MuMu/test/printTriggers.cxx. It needs to be compiled in order to check the `MuX_Track0_JPsi` paths (CINT is not able to understand the code). You can execute it with

```
root.exe -b -l -q myfile.root printTriggers.cxx+ | tee trigger.txt
```

Note that the trigger can print the same trigger more than once for a given muon; this is a feature in how trigger matches are embedded but it's *not* a problem and does *not* mean that that muon was matched to multiple trigger primitives from the same path.

How to run Onia2MuMuPAT

Running Onia2MuMuPAT on simulated events

The starting point is **`HeavyFlavorAnalysis/Onia2MuMu/test/onia2MuMuPAT_cfg.py`**. Depending on the sample used, however, the configuration might need some changes due to the different trigger process name.

- **Summer10 redigi 3.7.X** samples (re-digi done in 3.7.X): the 3.6.X trigger is called `REDIGI37X` instead of `HLT` NEW
- **Summer10 redigi 3.6.X** samples (re-digi done in 3.6.X): the 3.6.X trigger is called `REDIGI36X` instead of `HLT` NEW
- **Spring10 new** samples (gen-sim done in 3.5.X): no changes needed
- **Spring10 redigi** samples, 3.5.X (gen-sim from Summer09, done in 3.1.X): the 3.5.X trigger is called `REDIGI` instead of `HLT`
- **Summer09** samples: the trigger of interest is called `HLT8E29` instead of `HLT` , and it does not include onia triggers, only the plain ones (`L1SingleMuOpen`, `L1DoubleMuOpen`, `Mu3`, `Mu5`, `DoubleMu0`, `DoubleMu3`)

To determine if a Summer10 is 3.7.X or 3.6.X, just look in the dataset name for the global tag (e.g. START36_V9 is 3.6.X, START37_V5 is 3.5.X), or look for the relase in DBS. To determine if a Spring10 sample is new or redigi, look at the `GEN-SIM-RAW` file in DBS and see if it has as parents a Summer09 sample.

Changing the trigger name: put, at the bottom of your cfg file

```
process.patTrigger.processName = 'REDIGI'  # or 'HLT8E29'
```

```
process.muonMatchHLTCtfTrack.collectionTags[0] = process.muonMatchHLTCtfTrack.collectionTags[0].r
```

Removing completely the trigger matching: e.g. if you're running on a sample without trigger information

```
process.patMuonSequence.remove(process.patTriggerMatching)
process.patMuons.src = 'patMuonsWithoutTrigger'
```

Running Onia2MuMuPAT on real data

The configuration file **`HeavyFlavorAnalysis/Onia2MuMu/test/onia2MuMuPATData_cfg.py`** provides an example to run on real data. The main differences with Onia2MuMuPAT on data with respect to running on MC

Configuration of Onia2MuMuPAT:
To configure Onia2MuMuPAT to run on real data, you have to remove the montecarlo matching module `muonMatch` from the `patMuons` sequence, set `addGenMatch = False` in the producer of pat muons (module `patMuonsWithoutTrigger`), and set `addMCTruth` to false in the Onia2MuMuPAT producer module. These technical tasks have been automated for you, so if you're running with the standard configuration you can just add or uncomment these two lines at the end of the cfg file

```
from HeavyFlavorAnalysis.Onia2MuMu.onia2MuMuPAT_cff import onia2MuMu_isNotMC
onia2MuMu_isNotMC(process)
```

Filtering on Collision events:
A set of filters is defined in the configuration and run in the following paths:

- **`L1MinBiasVetoBeamHalo`**: (40 OR 41) AND NOT (36 OR 37 OR 38 OR 39)
- **`PhysicsDeclared`**: same as HLT_PhysicsDeclared
- **`MinBiasBSC`**: same as HLT_MinBiasBSC
- **`PrimaryVertex`**: filters on good vertexScraping
- **`Scraping`**: rejects scraping (monster) events

By default the output module does not select events based on these path but later the user can use them as trigger bits to select desired events.

Other things to check:

- You might want to select on runs or lumisections (syntax: **`process.source.lumisToProcess = cms.untracked.VEventID('<run>:<lumi>-<run>:<lumi>', ..., '<run>-<run>', ...)`**
- The `CMS.GlobalTag`.

Running Onia2MuMuPAT via CRAB

The configuration file **`HeavyFlavorAnalysis/Onia2MuMu/test/onia2MuMuPAT_cfg.py`** can be used on the grid. An example CRAB configuration file is provided **`HeavyFlavorAnalysis/Onia2MuMu/test/onia2MuMuPATData.crab`**.

Things that you should consider reviewing are:

- the selection of event content, and whether event filtering is on or off
- the global tag; lists of valid global tags for each release and their contents can be found under SWGuideFrontierConditions
- the verbosity of the message logger (e.g. setting **`process.MessageLogger.cerr.FwkReport.reportEvery = 1000`**)

Unless you're just analyzing the di-muons on the fly in the same yob, you have to set **`get_edm_output = 1`** under the `CMSSW` section of your `crab.cfg` in order to get your output.

If in the output module you save only the pat di-muons (or also the pat muons) and you're running privately on few events, then you should collect the output in your sandbox (**`return_data = 1`**), while otherwise you should store them on some storage element and publish them in local dbs.

privately-produced samples

If you're running on samples that have been produced privately and that don't have trigger emulation included, you'll probably stumble in this error message

```
---- ProductNotFound BEGIN
getByLabel: Found zero products matching all criteria
Looking for type: std::vector
```

In order to remove trigger-matching from Onia2MuMuPAT, just add at the end of your cfg.py file these two lines:

```
process.patMuonSequence.remove(process.patTriggerMatching)
process.patMuons.src = 'patMuonsWithoutTrigger'
```

Contact

- **Group homepage**: B Physics
- **Conveners**: Paula Eerola, Carlos Lourenco
- **Hypernews fora**: https://hypernews.cern.ch/HyperNews/CMS/get/bphysics.html, (hn-cms-bphysics@cernNOSPAMPLEASE.ch) and https://hypernews.cern.ch/HyperNews/CMS/get/quarkonia.html, (hn-cms-quarkonia@cernNOSPAMPLEASE.ch)
- **Page responsible**: RobertoCovarelli
- **Developers**: RobertoCovarelli, GiovanniPetrucciani, LucaMartini, BorisMangano, ...

This topic: CMSPublic > SWGuideOnia2MuMuPAT
Topic revision: r23 - 2010-07-12 - GiovanniPetrucciani