
RETURN
OF THE
DAVINCI

Recap

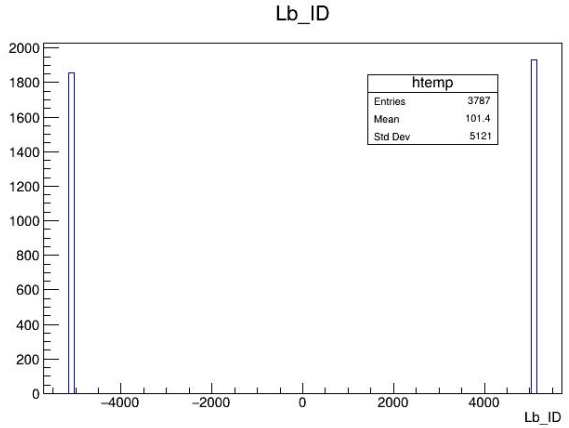
- We now have a very complete script we could perform a lovely analysis with!
- BUT I hear you cry, you want more, some more advanced topics…
- In this final part we will cover mysterious things such as TISTOS, DTF and MCTRUTH!

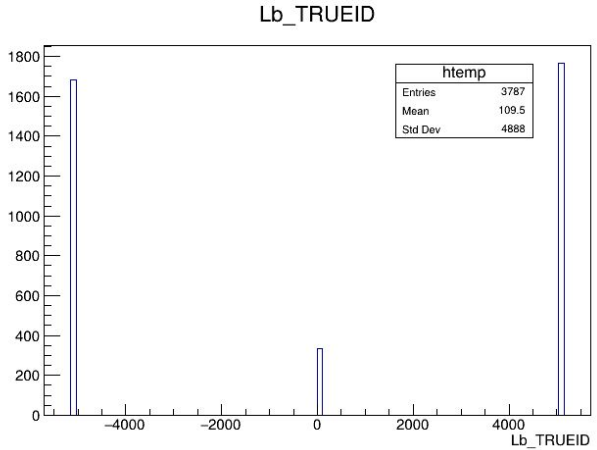
MC TRUTH

Truth or Dare

- MCTRUTH variables are only possible for MC simulation - it can retrieve properties of the particles that were actually generated.
- During simulation the detector is modelled and your sample is made from simulating PP collisions and reconstructing what you are looking for.
- But was that really a true candidate or was it mis-reconstructed from random track hits in the detector…
- When we want to use the MC e.g. for BDT training we want it to learn the kinematics of the signal not mis-reconstructed stuff!

Example from my work





- Can see in the TRUE variable we actually have this additional peak at 0. This is the value I set if there does not exist a TRUE_ID e.g. it wasn't really the signal!

Defining the Truth

```python
# Function to define truth MC variables
def make_mc_variables(data):
    mctruth = MCTruthAndBkgCat(data)
    # Helper lambda for truth information
    MCTRUTH = lambda func: F.MAP_INPUT(Functor=func, Relations=mctruth.MCAssocTable)

    trueid_bkgcat_info = FunctorCollection({
        "TRUEID": F.VALUE_OR(0) @ MCTRUTH(F.PARTICLE_ID),
        "TRUEKEY": F.VALUE_OR(-1) @ MCTRUTH(F.OBJECT_KEY),
        "TRUEPT": MCTRUTH(F.PT),
        "TRUEPX": MCTRUTH(F.PX),
        "TRUEPY": MCTRUTH(F.PY),
        "TRUEPZ": MCTRUTH(F.PZ),
        "TRUEENERGY": MCTRUTH(F.ENERGY),
        "TRUEP": MCTRUTH(F.P),
        "TRUEFOURMOMENTUM": MCTRUTH(F.FOURMOMENTUM),
        "BKGCAT": F.BKGCAT(Relations=mctruth.BkgCatTable),
    })
    return trueid_bkgcat_info
```

Defining the Truth

Define our function and pass in our data!

```
def make_mc_variables(data):
    # Helper lambda for truth information
    MCTRUTH = lambda func: F.MAP_INPUT(Functor=func, Relations=mctruth.MCAssocTable)

    trueid_bkgcat_info = FunctorCollection({
        "TRUEID": F.VALUE_OR(0) @ MCTRUTH(F.PARTICLE_ID),
        "TRUEKEY": F.VALUE_OR(-1) @ MCTRUTH(F.OBJECT_KEY),
        "TRUEPT": MCTRUTH(F.PT),
        "TRUEPX": MCTRUTH(F.PX),
        "TRUEPY": MCTRUTH(F.PY),
        "TRUEPZ": MCTRUTH(F.PZ),
        "TRUEENERGY": MCTRUTH(F.ENERGY),
        "TRUEP": MCTRUTH(F.P),
        "TRUEFOURMOMENTUM": MCTRUTH(F.FOURMOMENTUM),
        "BKGCAT": F.BKGCAT(Relations=mctruth.BkgCatTable),
    })
    return trueid_bkgcat_info
```

Defining the Truth

Using this new import called MCTruthAndBkgCat. It is a bit fancy but basically defines an algorithm that builds a map -> a 1 to 1 relation table between the reconstructed particle and the truth MC particle.

```python
# Function to define truth MC variables
def make_mc_variables(data):
    mctruth = MCTruthAndBkgCat(data)
    # Helper lambda for truth information
    MCTRUTH = lambda func: F.MAP_INPUT(Functor=func, Relations=mctruth.MCAssocTable)

    trueid_bkgcat_info = FunctorCollection({
        "TRUEID": F.VALUE_OR(0) @ MCTRUTH(F.PARTICLE_ID),
        "TRUEKEY": F.VALUE_OR(-1) @ MCTRUTH(F.OBJECT_KEY),
        "TRUEPT": MCTRUTH(F.PT),
        "TRUEPX": MCTRUTH(F.PX),
        "TRUEPY": MCTRUTH(F.PY),
        "TRUEPZ": MCTRUTH(F.PZ),
        "TRUEENERGY": MCTRUTH(F.ENERGY),
        "TRUEP": MCTRUTH(F.P),
        "TRUEFOURMOMENTUM": MCTRUTH(F.FOURMOMENTUM),
        "BKGCAT": F.BKGCAT(Relations=mctruth.BkgCatTable),
    })
    return trueid_bkgcat_info
```

Defining the Truth

Here we define a helper lambda function. It takes in “func” (F.PX, F.PT) and returns a new functor MCTRUTH that knows how to go from the standard functor to the truth functor using this MCAssocTable.

```python
# Function to define truth MC variables
def make_mc_variables(data):
    mctruth = MCTruthAndBkgCat(data)
    MCTRUTH = lambda func: F.MAP_INPUT(Functor=func, Relations=mctruth.MCAssocTable)

    trueid_bkgcat_info = FunctorCollection({
        "TRUEID": F.VALUE_OR(0) @ MCTRUTH(F.PARTICLE_ID),
        "TRUEKEY": F.VALUE_OR(-1) @ MCTRUTH(F.OBJECT_KEY),
        "TRUEPT": MCTRUTH(F.PT),
        "TRUEPX": MCTRUTH(F.PX),
        "TRUEPY": MCTRUTH(F.PY),
        "TRUEPZ": MCTRUTH(F.PZ),
        "TRUEENERGY": MCTRUTH(F.ENERGY),
        "TRUEP": MCTRUTH(F.P),
        "TRUEFOURMOMENTUM": MCTRUTH(F.FOURMOMENTUM),
        "BKGCAT": F.BKGCAT(Relations=mctruth.BkgCatTable),
    })
    return trueid_bkgcat_info
```

Defining the Truth

Here our new little helper lambda function is in action!

```python
# Function to define truth MC variables
def make_mc_variables(data):
    mctruth = MCTruthAndBkgCat(data)
    # Helper lambda for truth information
    MCTRUTH = lambda func: F.MAP_INPUT(Functor=func, Relations=mctruth.MCAssocTable)

    trueid_bkgcat_info = FunctorCollection({
        "TRUEID": F.VALUE_OR(0) @ MCTRUTH(F.PARTICLE_ID),
        "TRUEKEY": F.VALUE_OR(-1) @ MCTRUTH(F.OBJECT_KEY),
        "TRUEPT": MCTRUTH(F.PT),
        "TRUEPX": MCTRUTH(F.PX),
        "TRUEPY": MCTRUTH(F.PY),
        "TRUEPZ": MCTRUTH(F.PZ),
        "TRUEENERGY": MCTRUTH(F.ENERGY),
        "TRUEP": MCTRUTH(F.P),
        "TRUEFOURMOMENTUM": MCTRUTH(F.FOURMOMENTUM),
        "BKGCAT": F.BKGCAT(Relations=mctruth.BkgCatTable),
    })
    return trueid_bkgcat_info
```

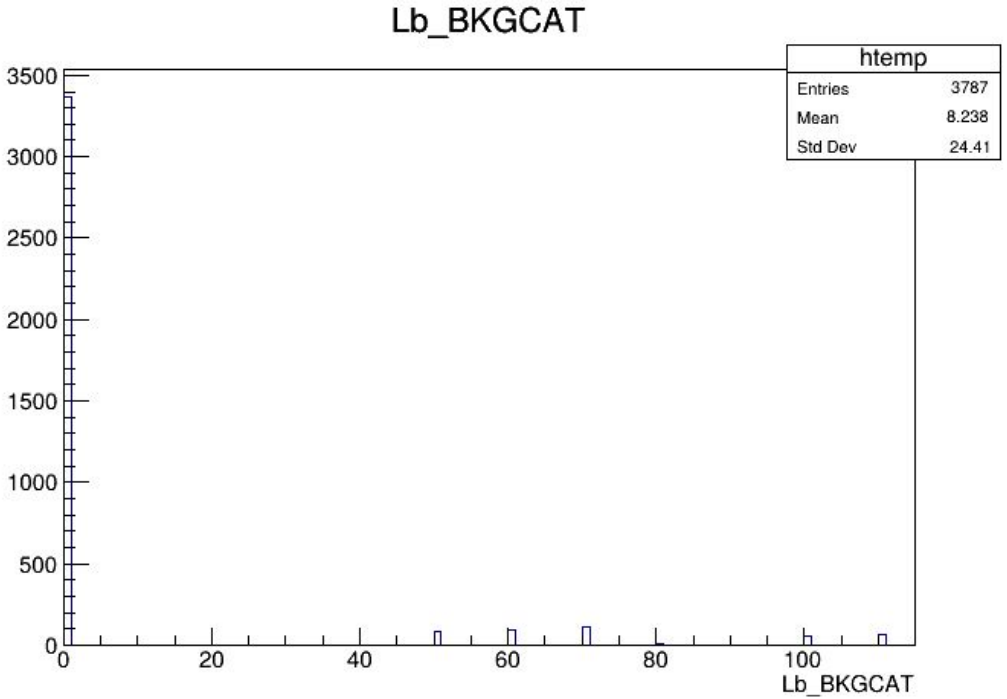
Defining the Truth

This guy differs though and tells you what type of background you have. e.g. 0 if pure signal, different values if its partially reconstructed or combinatorial etc

```python
# Function to define truth MC variables
def make_mc_variables(data):
    mctruth = MCTruthAndBkgCat(data)
    # Helper lambda for truth information
    MCTRUTH = lambda func: F.MAP_INPUT(Functor=func, Relations=mctruth.MCAssocTable)

    trueid_bkgcat_info = FunctorCollection({
        "TRUEID": F.VALUE_OR(0) @ MCTRUTH(F.PARTICLE_ID),
        "TRUEKEY": F.VALUE_OR(-1) @ MCTRUTH(F.OBJECT_KEY),
        "TRUEPT": MCTRUTH(F.PT),
        "TRUEPX": MCTRUTH(F.PX),
        "TRUEPY": MCTRUTH(F.PY),
        "TRUEPZ": MCTRUTH(F.PZ),
        "TRUEENERGY": MCTRUTH(F.ENERGY),
        "TRUEP": MCTRUTH(F.P),
        "TRUEFOURMOMENTUM": MCTRUTH(F.FOURMOMENTUM),
        "BKGCAT": F.BKGCAT(Relations=mctruth.BkgCatTable),
    })
    return trueid_bkgcat_info
```

Background Category



- Information on what each category stands for can be found here

Implementing

- Along with the function definition shown above we need to include this snippet to actually apply the TRUTH variables to each particle!
- This options.simulation reads your simulation status from the .yaml file.

```
if options.simulation == True:
    mc_variables = make_mc_variables(data)
    all_vars += mc_variables
```

Trigger and TISTOS

Trigger Info

- It can be extremely useful to know whether a candidate fired a particular trigger line in HLT1. These are crucial for efficiency studies and computing data-MC trigger corrections.
- These come in the form of HLT1 TISTOS variables…

HLT1 Lines

- There are a variety of HLT1 lines you might be interested in.
- For this example I will mention a handful, loads listed here:

  “Hlt1TrackMVA” - Uses MVA to select single high quality track

  “Hlt1TwoTrackMVA” - Uses MVA to select a pair of good quality tracks

  “Hlt1DiMuonLowMass” - Looks for a pair of muons with low mass

  “Hlt1DiMuonHighMass” - Looks for a pair of muons with high mass

TICTAC noooo TISTOS

- This classically confusing topic I will attempt to break down for you.
- First what does this acronym stand for?

  TIS = Trigger Independent Signal

  TOS = Trigger On Signal

- We want to know whether the HLT1 lines were fired by the particles in the event as TIS, TOS or TOB? (will discuss in a minute) - why? Because the trigger efficiencies vary if it was TIS or TOS!

TOS - Trigger On Signal

- Did ***MY*** signal particles cause the trigger to fire? If so that means enough of your signal's tracks/hits overlap with what caused the trigger to fire.
- So for our example were the J/psi muons what fired the DiMuon trigger line or was it two different muons that happened to be in the same event?
- Did the K+ fire the TrackMVA trigger or some other track?
- If the answer to these is YES then that was a TOS ! 👍

TIS - Trigger Independent Signal

- Would the event have triggered even without your signal?
- So other particles in your event caused the trigger line to fire.
- So for our example there could have been some other decay that happened to decay to 2 muons which fired the trigger.
- This sample still contains your event ***BUT*** it wasn't your event that caused it to trigger!
- That is why the efficiencies are different!

TOB - Trigger On Both

- These are events that are neither TIS nor TOS
- The presence of the signal alone nor the rest of the event alone are sufficient to generate a positive trigger decision, but rather both are necessary.

TOB….LERONE

Don’t all laugh at once…

The Overall Picture

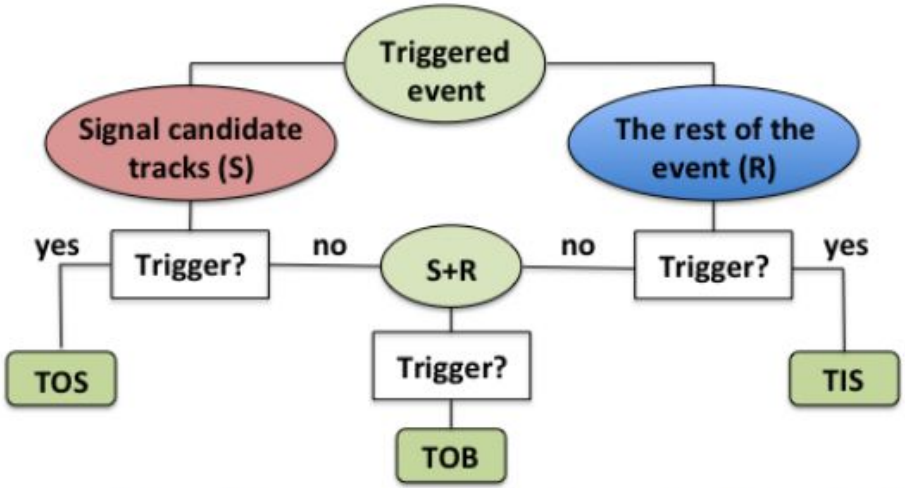


Figure 1: Diagram explaining the logic behind categorizing events into Trigger On Signal (TOS), Trigger Independent of Signal (TIS) and Trigger On Both (TOB) trigger categories. Note that an event can be both TIS and TOS simultaneously.

Implementing

```
Hlt1_decisions = ["Hlt1TrackMVA", "Hlt1TwoTrackMVA",          # Have just chosen 4 to look at as example
                  "Hlt1DiMuonLowMass", "Hlt1DiMuonHighMass"]

all_vars += FC.HltTisTos(                                     # Use the functorcollection HltTisTos
    selection_type="Hlt1",
    trigger_lines=[f"{x}Decision" for x in Hlt1_decisions],   # Simple for loop for each decision
    data=data)

evt_variables += FC.SelectionInfo(selection_type="Hlt1", trigger_lines=Hlt1_decisions) # Did each HLT1 line fire at all yes or no for the whole event
```

- It is easily implemented in our script as above!
- Also included this event variable selection to say if each HLT1 line fired at all for the whole event.

Decay Tree Fitter (DTF)

What is DTF

- The purpose of DTF is to perform a kinematic fitting of the decay to improve the mass resolution.
  - Applying vertex constraints
  - Applying mass constraints

How to Implement

```python
# DTF with PV constraint only
dtf_pv = DecayTreeFitter(
    name="PV_Fit",
    input_particles=data,
    input_pvs=pvs,
)
dtf_pv_vars = FC.DecayTreeFitterResults(
    DTF=dtf_pv,
    prefix="DTF_PV_",
    decay_origin=True,
    with_lifetime=True,
    with_kinematics=True,
)

# DTF with PV and J/psi mass constraint
dtf_pv_jpsi = DecayTreeFitter(
    name="PV_Jpsi_Fit",
    input_particles=data,
    input_pvs=pvs,
    mass_constraints=["J/psi(1S)"],
)
dtf_pv_jpsi_vars = FC.DecayTreeFitterResults(
    DTF=dtf_pv_jpsi,
    prefix="DTF_PV_JpsiM_",
    decay_origin=True,
    with_lifetime=True,
    with_kinematics=True,
)

variables = {
    "ALL": all_vars,                                                    # Variables for all particles
    "Bplus": composite_variables + dtf_pv_vars + dtf_pv_jpsi_vars,      # Variables specific to B+
    "Jpsi": composite_variables,                                        # Variables specific to J/ψ
    "muplus": track_variables,                                          # Variables for μ+
    "muminus": track_variables,                                         # Variables for μ-
    "Kplus": track_variables                                            # Variables for K+
}
```

How to Implement

Using DecayTreeFitter we name it PV_Fit since this function will only fix the PV. We also give it our particles and pvs.

```
# DTF with PV constraint only
dtf_pv = DecayTreeFitter(
    name="PV_Fit",
    input_particles=data,
    input_pvs=pvs,
)
                                    esults(
    DTF=dtf_pv,
    prefix="DTF_PV_",
    decay_origin=True,
    with_lifetime=True,
    with_kinematics=True,
)

# DTF with PV and J/psi mass constraint
dtf_pv_jpsi = DecayTreeFitter(
    name="PV_Jpsi_Fit",
    input_particles=data,
    input_pvs=pvs,
    mass_constraints=["J/psi(1S)"],
)
dtf_pv_jpsi_vars = FC.DecayTreeFitterResults(
    DTF=dtf_pv_jpsi,
    prefix="DTF_PV_JpsiM_",
    decay_origin=True,
    with_lifetime=True,
    with_kinematics=True,
)

variables = {
    "ALL": all_vars,                                                    # Variables for all particles
    "Bplus": composite_variables + dtf_pv_vars + dtf_pv_jpsi_vars,      # Variables specific to B+
    "Jpsi": composite_variables,                                        # Variables specific to J/ψ
    "muplus": track_variables,                                          # Variables for μ+
    "muminus": track_variables,                                         # Variables for μ-
    "Kplus": track_variables                                            # Variables for K+
}
```

How to Implement

Now we use this model with only a PV constraint and ask for:
- decay_origin=True to give us info on the PV.
- with_lifetime=True to give us info related to lifetimes.
- with_kinematics=True to give mass, P, PT etc

```python
# DTF with PV constraint only
dtf_pv = DecayTreeFitter(
    name="PV_Fit",
    input_particles=data,
    input_pvs=pvs,
)
dtf_pv_vars = FC.DecayTreeFitterResults(
    DTF=dtf_pv,
    prefix="DTF_PV_",
    decay_origin=True,
    with_lifetime=True,
    with_kinematics=True,
)
# DTF with PV and J/psi mass constraint
dtf_pv_jpsi = DecayTreeFitter(
    name="PV_Jpsi_Fit",
    input_particles=data,
    input_pvs=pvs,
    mass_constraints=["J/psi(1S)"],
)
dtf_pv_jpsi_vars = FC.DecayTreeFitterResults(
    DTF=dtf_pv_jpsi,
    prefix="DTF_PV_JpsiM_",
    decay_origin=True,
    with_lifetime=True,
    with_kinematics=True,
)

variables = {
    "ALL": all_vars,                                                    # Variables for all particles
    "Bplus": composite_variables + dtf_pv_vars + dtf_pv_jpsi_vars,      # Variables specific to B+
    "Jpsi": composite_variables,                                        # Variables specific to J/ψ
    "muplus": track_variables,                                          # Variables for μ+
    "muminus": track_variables,                                         # Variables for μ-
    "Kplus": track_variables                                            # Variables for K+
}
```

How to Implement

Do the same as the start but also add in an additional constraint of requiring the J/psi mass to be fixed to the PDG value

```python
# DTF with PV constraint only
dtf_pv = DecayTreeFitter(
    name="PV_Fit",
    input_particles=data,
    input_pvs=pvs,
)
dtf_pv_vars = FC.DecayTreeFitterResults(
    DTF=dtf_pv,
    prefix="DTF_PV_",
    decay_origin=True,
    with_lifetime=True,
    with_kinematics=True,
)

# DTF with PV and J/psi mass constraint
dtf_pv_jpsi = DecayTreeFitter(
    name="PV_Jpsi_Fit",
    input_particles=data,
    input_pvs=pvs,
    mass_constraints=["J/psi(1S)"],
)
dtf_pv_jpsi_vars = FC.DecayTreeFitterResults(
    DTF=dtf_pv_jpsi,
    prefix="DTF_PV_JpsiM_",
    decay_origin=True,
    with_lifetime=True,
    with_kinematics=True,
)

variables = {
    "ALL": all_vars,                                          # Variables for all particles
    "Bplus": composite_variables + dtf_pv_vars + dtf_pv_jpsi_vars,  # Variables specific to B+
    "Jpsi": composite_variables,                              # Variables specific to J/ψ
    "muplus": track_variables,                                # Variables for μ+
    "muminus": track_variables,                               # Variables for μ-
    "Kplus": track_variables                                  # Variables for K+
}
```

How to Implement

Same again but for the PV
+ J/psi constraint fit

```python
# DTF with PV constraint only
dtf_pv = DecayTreeFitter(
    name="PV_Fit",
    input_particles=data,
    input_pvs=pvs,
)
dtf_pv_vars = FC.DecayTreeFitterResults(
    DTF=dtf_pv,
    prefix="DTF_PV_",
    decay_origin=True,
    with_lifetime=True,
    with_kinematics=True,
)

# DTF with PV and J/psi mass constraint
dtf_pv_jpsi = DecayTreeFitter(
    name="PV_Jpsi_Fit",
    input_particles=data,
    input_pvs=pvs,
    mass_constraints=["J/psi(1S)"],
)
dtf_pv_jpsi_vars = FC.DecayTreeFitterResults(
    DTF=dtf_pv_jpsi,
    prefix="DTF_PV_JpsiM_",
    decay_origin=True,
    with_lifetime=True,
    with_kinematics=True,
)

variables = {
    "ALL": all_vars,                                                    # Variables for all particles
    "Bplus": composite_variables + dtf_pv_vars + dtf_pv_jpsi_vars,      # Variables specific to B+
    "Jpsi": composite_variables,                                        # Variables specific to J/ψ
    "muplus": track_variables,                                          # Variables for μ+
    "muminus": track_variables,                                         # Variables for μ-
    "Kplus": track_variables                                            # Variables for K+
}
```

How to Implement

Finally make sure we add these variables to the B_plus!

```python
# DTF with PV constraint only
dtf_pv = DecayTreeFitter(
    name="PV_Fit",
    input_particles=data,
    input_pvs=pvs,
)
dtf_pv_vars = FC.DecayTreeFitterResults(
    DTF=dtf_pv,
    prefix="DTF_PV_",
    decay_origin=True,
    with_lifetime=True,
    with_kinematics=True,
)

# DTF with PV and J/psi mass constraint
dtf_pv_jpsi = DecayTreeFitter(
    name="PV_Jpsi_Fit",
    input_particles=data,
    input_pvs=pvs,
    mass_constraints=["J/psi(1S)"],
)
dtf_pv_jpsi_vars = FC.DecayTreeFitterResults(
    DTF=dtf_pv_jpsi,
    prefix="DTF_PV_JpsiM_",
    decay_origin=True,
    with_lifetime=True,
    with_kinematics=True,
)

variables = {
    "ALL": all_vars,
    "Bplus": composite_variables + dtf_pv_vars + dtf_pv_jpsi_vars,
    "Jpsi": composite_variables,
    "muplus": track_variables,
    "muminus": track_variables,
    "Kplus": track_variables
}

# Variables for all particles
# Variables specific to B+
# Variables specific to J/ψ
# Variables for μ+
# Variables for μ-
# Variables for K+
```

DTF

- You can of course add any constraints you might want, these are just two examples e.g. you can also change the particles hypothesis!
- More details about the maths behind DTF can be found here!

I’m Probably Tired Now 💀😴

- That’s it, enough advanced topics and enough of me yapping. Time for you guys to play around with these new features and explore for yourself!
- Please head over to the gitlab again, link here, read through the comments on the advanced script and try and run it again on MC.
- Check the HLT1 TISTOS branches make sense, TRUTH_ID’s look sensible and DTF’s do indeed constrain things!
- Tasks:
- Now try and use the data.yaml to run this script on data, how do variables change?
- Add some more HLT1 decisions and see if they are TIS or TOS.
- Add some more DTF’s and try and constrain other masses.
- Also you might’ve noticed we are looking at a “detached” line, what do you think that means and what cut might have been implemented to obtain detached events! Bonus points if you can find the exact cuts applied in HLT2 selections in Moore lol. (Does this match what you see in your tuple)!
- If we are good for time we will end with a quiz to see what you remember!

BACKUP