Analysis of the Invariant Mass Spectrum of Di-muon Events

September 20, 2025

Contents

1 Determining the Di-Muon Invarient Mass

1.1 Introduction

In this project, we analyze data from the CMS Open Data to study the production of muon pairs in high-energy proton–proton collisions at the LHC. Using ROOT and PyROOT, we reconstruct the invariant mass distribution of dimuon events. The invariant mass spectrum provides direct insight into the presence of intermediate resonances, such as the $J/\psi, \Upsilon$, and the Z boson, which decay into muon pairs ($\mu^+\mu^-$).

This exercise demonstrates how experimental high-energy physics identifies particles through their decay signatures, and how statistical patterns in collision data reveal fundamental properties of matter.

1.2 Data Acquisition

For this analysis, we use the DoubleMuParked dataset from 2012. Specifically, we download the file **"Run2012BC_DoubleMuParked_Muons.root"** from the CERN Open Data portal. This file contains muon-pair events collected during Runs 2012B and 2012C at the LHC, focusing on "parked" double-muon triggers. It serves as the raw data input for our dimuon invariant mass reconstruction. Download the ROOT file via the provided link.

Link: https://opendata.cern.ch/record/12341/files/Run2012BC_DoubleMuParked_Muons.root

1.3 Data Analysis

[1]:
```
import ROOT
import numpy as np
%jsroot off
```

[2]:
```
from ROOT import TFile, TH1D

file = TFile("Run2012BC_DoubleMuParked_Muons.root","READ")
tree = file.Get("Events")   # Access the TTree named "Events"

Hist = TH1D('h','h', 30000, 0.25, 160)

print(type(file))
print(type(tree))
print(type(Hist))

# Get total number of events in the TTree
print("\nTotal number of entries in the tree is",tree.GetEntries()) # \n is for␣
 ↪new line
```

```
<class cppyy.gbl.TFile at 0x618751b83310>
<class cppyy.gbl.TTree at 0x61874c52e4b0>
<class cppyy.gbl.TH1D at 0x618751cfe7c0>

Total number of entries in the tree is 61540413
```

`TFile` is a ROOT object used to open ROOT files. `"READ"` mode means you are opening the file just to read its contents, not to modify it.

Significance:

- You cannot accidentally overwrite or delete data in the ROOT file. Useful when analyzing large datasets where integrity is crucial.
- Faster for reading. ROOT optimizes read-only access because it doesn't need to allocate resources for writing or updating objects.
- Operations allowed in read-only mode. You can read trees, histograms, and other objects.

`file` now represents the opened ROOT **TFile**, which can contain trees, histograms, and other objects. `Get("Events")` retrieves **TTree** object named "Events" from the ROOT file. And it is stored in `tree` variable containing all the events.

```
Hist = TH1D('h','h', 30000, 0.25, 160)
```

A new Histogram is created. This histogram "Hist" is empty at first and can be filled with data from the TTree. Histogram's name and title is `'h'`. Number of bins are 30000. And $(0.25, 160)$ are lower and upper bound of the histogram. `tree.GetEntries()` returns the total number of entries (rows) in the TTree.

These outputs confirm that your objects are indeed ROOT objects/classes (`TFile`, `TTree`, `TH1D`) stored in respective pointers and is accessible in Python via PyROOT/cppyy. Here `cppyy` is a **Python-C++** binding library. It allows Python code to directly access C++ classes, functions, and objects as if they were native Python objects. ROOT is primarily written in C++, so cppyy is the bridge that lets Python use ROOT classes seamlessly. `cppyy.gbl` stands for Global C++ namespace.

[3]:
```
# Function to print all leaves in the tree
def print_leaves(tree):
    for branch in tree.GetListOfBranches():
        print(branch.GetName())

# Print the leaves
print_leaves(tree)
```

```
nMuon
Muon_pt
Muon_eta
Muon_phi
Muon_mass
Muon_charge
```

The function `print_leaves` loops over all branches and prints their names. Each **TTree** is made up of **Branches** (think of them like columns in a table). Each branch can contain one or more **Leaves** (actual variable like the numbers inside that column).

`tree.GetListOfBranches()` returns all branches in the tree. And `branch.GetName()` gets the name of each branch.

[4]:
```
tree.Show(12500)  # Show the content of the 12500th entry
```

```
======> EVENT:12500
 nMuon           = 2
 Muon_pt         = 14.7748,
                   3.48163
 Muon_eta        = 1.36514,
                   -0.251159
 Muon_phi        = 0.12591,
                   2.57296
 Muon_mass       = 0.105658,
                   0.105658
 Muon_charge     = 1,
                   -1
```

[5]:
```
percent = 50  # percentage of events to process
n = int(percent*(tree.GetEntries())/100)
print("Number of events selected is", n)
```

```
Number of events selected is 30770206
```

Let take `percent = 10`. So, out of total 61540413 entries, it will only analyze the first 6154041 events.

[6]:
```
# Create numpy arrays to hold the 3-momentum components of the two muons
m1 = np.zeros((2*n, 3)) # For muon 1
m2 = np.zeros((2*n, 3)) # For muon 2
```

- `np.zeros((shape))` → This creates a NumPy array filled with zeros. The argument `(2*n, 3)` specifies the shape of the array.
- `(2*n, 3)` → The array will have: $(2n)$ rows and 3 columns. So it's basically a matrix with $2n$ rows and 3 columns, all initialized with 0. For example: if $n = 2$ then it is a matrix of dimension (4×3), with all elements 0.
- `m1` and `m2` are two independent arrays of the same size. Both start filled with zeros, but later in your analysis you'll probably store event data (like momentum components, energies, etc.) in them, separately for two muons.

[7]:
```
from ROOT import TLorentzVector # Import TLorentzVector class from ROOT
import time

# Start the timer
start = time.time()

# Loop over events in the tree
for i, ent in enumerate(tree):
    if i>n:
        break;
    if (i > 0 and i % 500000 == 0):
```

```
        print("Processing: ", i, "th event!")

    # Extract muon information
    charge = np.array(ent.Muon_charge)
    number = np.array(ent.nMuon)

    # Select only events with exactly 2 opposite-charge muons
    if len(charge) == 2 and number == 2 and (charge[0] + charge[1] == 0):

        pt = list(ent.Muon_pt)
        eta = list(ent.Muon_eta)
        phi = list(ent.Muon_phi)
        mass = list(ent.Muon_mass)

        # First muon 4-vector
        LV1 = TLorentzVector()
        LV1.SetPtEtaPhiM(pt[0], eta[0], phi[0], mass[0])

        # Second muon 4-vector
        LV2 = TLorentzVector()
        LV2.SetPtEtaPhiM(pt[1], eta[1], phi[1], mass[1])

        # Store momentum components in arrays
        m1[i][0] = LV1.Px()
        m1[i][1] = LV1.Py()
        m1[i][2] = LV1.Pz()

        m2[i][0] = LV2.Px()
        m2[i][1] = LV2.Py()
        m2[i][2] = LV2.Pz()

        # Calculate combined 4-vector and invariant mass
        LVT = LV1 + LV2
        mass = LVT.M()
        Hist.Fill(mass)

print ("Loop is completed")
end = time.time()

# Stop the timer and print the elapsed time
print("Time taken for processing", n, "events is", end-start, "seconds")
```

```
Processing:  500000 th event!
Processing:  1000000 th event!
Processing:  1500000 th event!
Processing:  2000000 th event!
Processing:  2500000 th event!
Processing:  3000000 th event!
```

```
Processing:  3500000 th event!
Processing:  4000000 th event!
Processing:  4500000 th event!
Processing:  5000000 th event!
Processing:  5500000 th event!
Processing:  6000000 th event!
Processing:  6500000 th event!
Processing:  7000000 th event!
Processing:  7500000 th event!
Processing:  8000000 th event!
Processing:  8500000 th event!
Processing:  9000000 th event!
Processing:  9500000 th event!
Processing:  10000000 th event!
Processing:  10500000 th event!
Processing:  11000000 th event!
Processing:  11500000 th event!
Processing:  12000000 th event!
Processing:  12500000 th event!
Processing:  13000000 th event!
Processing:  13500000 th event!
Processing:  14000000 th event!
Processing:  14500000 th event!
Processing:  15000000 th event!
Processing:  15500000 th event!
Processing:  16000000 th event!
Processing:  16500000 th event!
Processing:  17000000 th event!
Processing:  17500000 th event!
Processing:  18000000 th event!
Processing:  18500000 th event!
Processing:  19000000 th event!
Processing:  19500000 th event!
Processing:  20000000 th event!
Processing:  20500000 th event!
Processing:  21000000 th event!
Processing:  21500000 th event!
Processing:  22000000 th event!
Processing:  22500000 th event!
Processing:  23000000 th event!
Processing:  23500000 th event!
Processing:  24000000 th event!
Processing:  24500000 th event!
Processing:  25000000 th event!
Processing:  25500000 th event!
Processing:  26000000 th event!
Processing:  26500000 th event!
Processing:  27000000 th event!
```

```
Processing:  27500000 th event!
Processing:  28000000 th event!
Processing:  28500000 th event!
Processing:  29000000 th event!
Processing:  29500000 th event!
Processing:  30000000 th event!
Processing:  30500000 th event!
Loop is completed
Time taken for processing 30770206 events is 2778.36674284935 seconds
```

1.4 Explanation

1.4.1 Timing:

```
start = time.time()
end = time.time()
```

Here you are using the **time** module in Python. `time.time()` returns the current time in seconds since the Unix epoch (Jan $1, 1970, 00:00:00$ UTC). You store that value in the variable `start`. It's a **timestamp** marking the moment your loop starts. Later, after the loop finishes, you take another timestamp `end`. Then you calculate:

```
print(end - start)
```

which gives the total runtime of your code in seconds. Thus, you can measure how long the loop takes.

1.4.2 Loop Over Events

```
for i, ent in enumerate(tree):
    if i>n:
        break;
    if (i > 0 and i % 100000 == 0):
        print("Processing: ", i, "th event!")
```

- `for i, ent in enumerate(tree)` → `tree` is your TTree (here: "Events"). And `enumerate(tree)` allows you to loop through all events, while also keeping track of the index `(i)`. So, i is event number $(0, 1, 2, ...)$. And `ent` is the actual entry in the tree (an event object that holds branches like Muon_pt, Muon_eta, etc.). So, each iteration gives you one event and its index.
- `if i > n: break` → n is the number of events you decided to process. This ensures you don't read the whole dataset, just the first n events. When the loop index i exceeds n, the loop stops immediately.
- `if (i > 0 and i % 100000 == 0):` → This is a progress checkpoint. `i % 100000 == 0` means **"every** $100,000$ **th event"**. So, whenever you hit event 100000, 200000, 300000, etc., the condition becomes true.
- `print("Processing: ", i, "th event!")` → When the checkpoint condition is true, you print a message like:

```
Processing:  100000 th event!
Processing:  200000 th event!
```

This helps track progress for large datasets so you know the code hasn't frozen.

1.4.3 Muon Charge and Multiplicity Checks

```
charge = np.array(ent.Muon_charge)
number = np.array(ent.nMuon)
```

- `ent.Muon_charge` → This is a branch that contains the charges of all muons in the event (e.g., $[+1, -1], [+1, +1]$, etc.). Converted into a numpy array `charge` for easier manipulation.
- `ent.nMuon` → This branch stores the number of muons in the event. Converted into numpy array `number`, but since `nMuon` is just a single integer, it will look like 2, 3, etc.

```
if len(charge) == 2 and number == 2 and (charge[0] + charge[1] == 0):
```

- `len(charge) == 2` → Means the list of muon charges has exactly 2 entries. So the event has 2 reconstructed muons.
- `number == 2` → Double check the branch `nMuon` (official muon count stored in the dataset) must also be 2. This redundancy is often used to ensure data consistency.
- `(charge[0] + charge[1] == 0)` → Adds the charge of the first and second muon. If they are opposite charges $(+1, -1)$, the sum is 0, and the condition is true. If they are the same sign $(+1, +1)$, or $(-1, -1)$, the sum is $+2$ or -2, and the condition fails.

This condition keeps only **di-muon** events where, exactly 2 muons of opposite charge exist. And only those events satisfying this condition is selected.

1.4.4 Get Kinematics

```
pt   = list(ent.Muon_pt)    # transverse momentum
eta  = list(ent.Muon_eta)   # pseudorapidity
phi  = list(ent.Muon_phi)   # azimuthal angle
mass = list(ent.Muon_mass)  # rest mass
```

- `ent.Muon_pt` → It is a branch that stores the **transverse momentum** (p_T) of each muon in the event. For example, $(45.2, 38.7$ GeV). It's converted into a Python list so you can easily index like `pt[0], pt[1]`.
- `ent.Muon_eta` → Gives the **pseudorapidity** (η) of each muon. And η is a coordinate defined in terms of polar angle θ relative to the beam axis as:

$$\eta = -\ln\left(\tan\frac{\theta}{2}\right)$$

- `ent.Muon_phi` → Gives the **azimuthal angle** (ϕ) of each muon in the transverse plane. usually $(-\pi \leq \phi < \pi)$. It tells you the direction of the muon around the beamline.
- `ent.Muon_mass` → Gives the **rest mass** of the muons. For real muons, the mass is about 105.7 MeV/c^2.

So, converts the branches into Python lists so you can index them easily. In reconstructed events, the software stores this as part of the muon 4-vector information.

1.4.5 Build Lorentz Vectors

```
LV1 = TLorentzVector()
LV1.SetPtEtaPhiM(pt[0], eta[0], phi[0], mass[0])
```

- `LV1 = TLorentzVector()` → Creates an empty **Lorentz 4-vector** object (from ROOT's `TLorentzVector` class). A 4-vector in relativity has components: (E, p_x, p_y, p_z), where E is the energy and $\vec{p}$ is the 3-momentum.
- `LV1.SetPtEtaPhiM(pt[0], eta[0], phi[0], mass[0])` → This fills the Lorentz vector using the standard HEP coordinates: (p_T, η, ϕ, m).

Internally, ROOT converts these into the Cartesian form:

$$(p_T, \eta, \phi, m) \quad \longrightarrow \quad \text{4-vector: } (E, p_x, p_y, p_z)$$

```
m1[i][0] = LV1.Px()
m1[i][1] = LV1.Py()
m1[i][2] = LV1.Pz()
```

- `LV1.Px()`, `LV1.Py()`, `LV1.Pz()` → ROOT provides methods for **Lorentz vector** `LV1` to get its Cartesian components by `.Px()` as ($p_x = p_T \cos\phi$), for first Muon. Similarly with `.Py()` and `.Pz()`.

$$\begin{aligned} p_x &= p_T \cos\phi \\ p_y &= p_T \sin\phi \\ p_z &= p_T \sinh(\eta) \\ E &= \sqrt{p_T^2 \cosh^2(\eta) + m^2} \end{aligned}$$

- `m1[i][0], m1[i][1], m1[i][2]` → This means: `m1` has $2n$ **rows** (because there are 2 muons per event $\times n$ events). Each row has **3 columns**, one for p_x, p_y, p_z. For the i-th event, you are saving the first muon's momentum components:
  - Column 0 → p_x
  - Column 1 → p_y
  - Column 2 → p_z

You are transferring the muon's momentum information from ROOT into a numpy structure. So after this call, `LV1` now fully represents the 4-momentum of the first muon in the event. Similairy, `Lv2` for second muon.

1.4.6 Calculate the Invarient Mass

```
LVT = LV1 + LV2
mass = LVT.M()
Hist.Fill(mass)
```

- `LVT = LV1 + LV2` → So, `LV1` and `LV2` are the Lorentz 4-vectors of the two muons in the event. Adding them gives the combined 4-vector: $\mu_1^\mu + \mu_2^\mu$. So `LVT` is the total 4-momentum of the pair.

- `mass = LVT.M()` → `M()` is a ROOT method that calculates the **invariant mass** of a 4-vector. Since `LVT` is the sum of the two muons' 4-vectors, `LVT.M()` gives the dimuon invariant mass. This is exactly the quantity you're interested in to find resonances (like the Z boson peak near 91 GeV).

$$M^2 = E^2 - |\vec{p}|^2$$

- `Hist.Fill(mass)` → Takes the calculated dimuon invariant mass and increments the bin that corresponds to that value. By looping over all events, you build up a distribution of invariant masses which should show a peak around the resonances.

[8]:
```
m1
```

[8]:
```
array([[ 0.        ,  0.        ,  0.        ],
       [10.14310332, -2.85958612, -4.64691517],
       [ 0.        ,  0.        ,  0.        ],
       …,
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ]], shape=(61540412, 3))
```

[9]:
```
m2
```

[9]:
```
array([[  0.        ,   0.        ,   0.        ],
       [-13.45861397,   9.24336543,   5.8184379 ],
       [  0.        ,   0.        ,   0.        ],
       …,
       [  0.        ,   0.        ,   0.        ],
       [  0.        ,   0.        ,   0.        ],
       [  0.        ,   0.        ,   0.        ]], shape=(61540412, 3))
```

[10]:
```
m1m2 = np.vstack((m1, m2))
m1m2
```

[10]:
```
array([[ 0.        ,  0.        ,  0.        ],
       [10.14310332, -2.85958612, -4.64691517],
       [ 0.        ,  0.        ,  0.        ],
       …,
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ]], shape=(123080824, 3))
```

Now, `m1` stores the 3-momentum (p_x, p_y, p_z) of the first muon in each event, `m2` stores the 3-momentum of the second muon in each event. So after the event loop, you have two separate NumPy arrays.

`np.vstack` means "vertical stack" in NumPy. It takes two arrays and stacks them row-wise (one on top of the other). `m1` has shape $(2n, 3)$. `m2` has shape $(2n, 3)$. After stacking `m1m2` has shape $(4n, 3)$.

`m1m2` is just a single combined array of muon momentum components: The first $2n$ rows contain momenta from 'm1' (first muon of each event). And the next $2n$ rows contain momenta from 'm2' (second muon of each event). So instead of keeping two separate containers, now you have all muons' 3-momenta together in one big matrix.

1.5 Viewing the Four Momentum

[11]:
```
# Print full 4-vector of muon 1
print("Muon 1:")
print("  Px =", LV1.Px())
print("  Py =", LV1.Py())
print("  Pz =", LV1.Pz())
print("  E  =", LV1.E())

# Print full 4-vector of muon 2
print("Muon 2:")
print("  Px =", LV2.Px())
print("  Py =", LV2.Py())
print("  Pz =", LV2.Pz())
print("  E  =", LV2.E())
```

```
Muon 1:
  Px = -6.613884075296045
  Py = 5.392224089836168
  Pz = 7.39200086673741
  E  = 11.290366854132163
Muon 2:
  Px = -14.167305287011093
  Py = 11.46780781617979
  Pz = 16.507223149313972
  E  = 24.591110893898822
```

This prints the four momentum components of the two muons for the current event (maybe the last event from the `n`th entries).

[12]:
```
# Print details
print("Muon pair invariant mass = ", mass, " GeV")

# Also print 4-vector components if needed
print("Resonance candidate 4-vector:")
print("  Px =", LVT.Px())
print("  Py =", LVT.Py())
print("  Pz =", LVT.Pz())
print("  E  =", LVT.E())
```

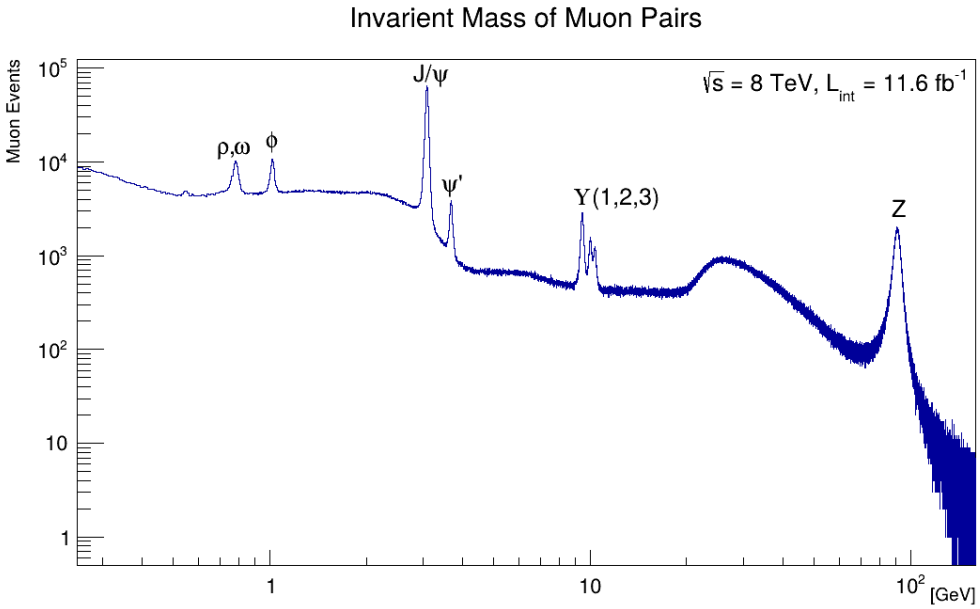
```
Muon pair invariant mass =  0.43477537774366537  GeV
Resonance candidate 4-vector:
  Px = -20.78118936230714
  Py = 16.86003190601596
```

```
Pz = 23.899224016051384
E  = 35.88147774803099
```

1.6 Plotting the Mass Spectrum

This prints the four momentum components of the invarient dimuon of current event. And also prints the invarient mass of the dimuon event.

[13]:
```
from ROOT import gStyle, TCanvas, TLatex

# Produce plot
gStyle.SetOptStat(0)
gStyle.SetTextFont(42)

# Create canvas
canvas = TCanvas("c1", "", 1200, 700)
canvas.SetLogx(); canvas.SetLogy()

# Draw histogram
Hist.SetTitle("Invarient Mass of Muon Pairs")
Hist.GetXaxis().SetTitleSize(0.03)
Hist.GetXaxis().SetTitle("[GeV]")
Hist.GetYaxis().SetTitleSize(0.03)
Hist.GetYaxis().SetTitle("Muon Events")
Hist.Draw()

# Add resonance labels
label = TLatex()
label.SetNDC(True) # Normalized Device Coordinates
label.SetTextSize(0.040)
label.DrawLatex(0.225, 0.750, "#rho,#omega")
label.DrawLatex(0.268, 0.760, "#phi")
label.DrawLatex(0.400, 0.865, "J/#psi")
label.DrawLatex(0.425, 0.690, "#psi'")
label.DrawLatex(0.543, 0.670, "#Upsilon(1,2,3)")
label.DrawLatex(0.825, 0.650, "Z")

# CMS style text
label.SetTextSize(0.040)
label.DrawLatex(0.655, 0.850, "#sqrt{s} = 8 TeV, L_{int} = 11.6 fb^{-1}")

canvas.Draw()
```



1.7 Conclusion

In the dimuon invariant mass histogram, each peak corresponds to a different particle resonance that can decay into a muon–antimuon pair:

Peak	Mass (approx)	Meaning	Branching to $\mu^+\mu^-$
η	0.55 GeV	Eta meson $\to \mu^+\mu^-$ (rare, mostly background)	$\sim 5.8 \times 10^{-6}$
ρ/ω	0.77/0.78 GeV	Light vector mesons, occasional muon decays	$\rho^0 : \sim 4.6 \times 10^{-5}$; $\omega : \sim 9 \times 10^{-5}$
ϕ	1.02 GeV	Phi meson $\to \mu^+\mu^-$	$\approx 2.9 \times 10^{-4}$
J/ψ	3.10 GeV	Charmonium, $\bar{c}c \to \mu^+\mu^-$	$\approx 5.96\%$
ψ'	3.68 GeV	Excited charmonium $\to \mu^+\mu^-$	$\sim 0.8\%$
$\Upsilon(1,2,3S)$	9.46/10.02/10.36 GeV	Bottomonium states $\to \mu^+\mu^-$	$1S : \approx 2.49\%$; $2S : \approx 2.03\%$; $3S : \approx 2.18\%$
Z	91 GeV	Z boson $\to \mu^+\mu^-$	$\approx 3.37\%$

The dataset contains muons from many different resonances, not just Z bosons. That's why in the histogram you see multiple peaks at low mass (0.5–10 GeV) and the large Z peak at 91 GeV. Low-mass peaks are usually mesons produced in hadronic collisions or through secondary decays. The Z peak dominates at high mass because it is a primary electroweak process at 8 TeV collisions.