



# Manipulando dados com o ROOT - Parte II

Introdução à análise de dados em FAE e tecnologias associadas



# Parte II

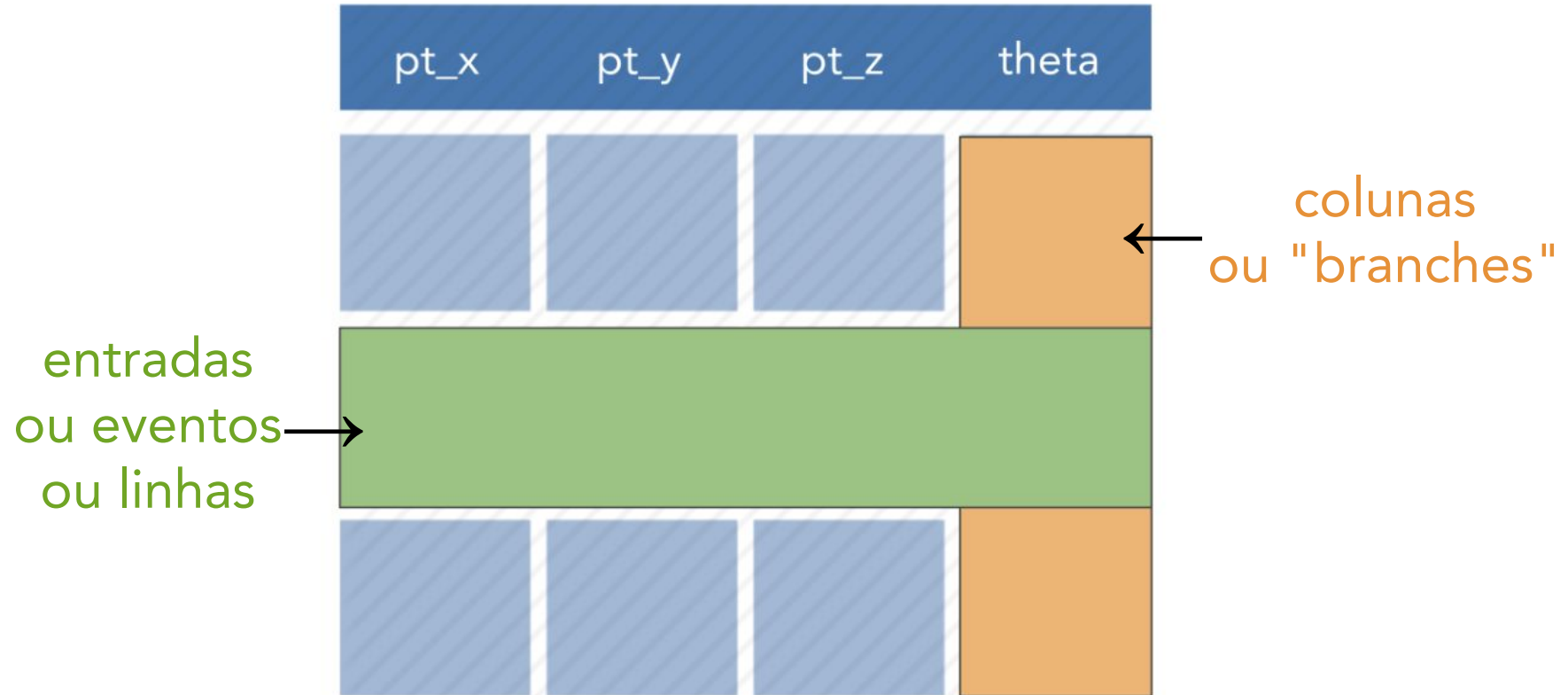
# ROOT - TTree

Uma **TTree** é uma estrutura de dados no ROOT usada para representar conjuntos de dados em formato **colunar**. Com ela podemos organizar e manipular várias variáveis ao mesmo tempo.

- Os dados em um TTree são organizados em colunas, também chamadas de branches .
- Cada branch representa uma variável ou um conjunto de variáveis relacionadas, e as colunas podem conter diferentes tipos de dados (como inteiros, floats, vetores, etc.).
- Um TTree pode armazenar qualquer tipo de objeto, permitindo organizar dados complexos (como vetores e classes personalizadas).
- Cada linha no TTree representa uma **entrada** (ou **evento**), com valores correspondentes em cada coluna.
- Uma maneira moderna de interagir com TTree é a interface **RDataFrame**.
  - Ela fornece uma maneira mais intuitiva de manipular e processar dados armazenados em TTrees, permitindo aplicar filtros, transformações e realizar operações de análise de forma mais eficiente.

Obs.: **Tutoriais do RDataFrame**

# ROOT - TTree



# ROOT - TTree - RDataFrame

Importar o ROOT e Criar um RDataFrame

1. Criar o RDataFrame a partir de uma TTree em um arquivo ROOT
2. Aplicar transformações aos dados: filtros, definir novas colunas,...
3. Criar e preencher histogramas

1. Criar RDataFrame

```
import ROOT
```

```
df = ROOT.RDataFrame("t", "f.root")
```

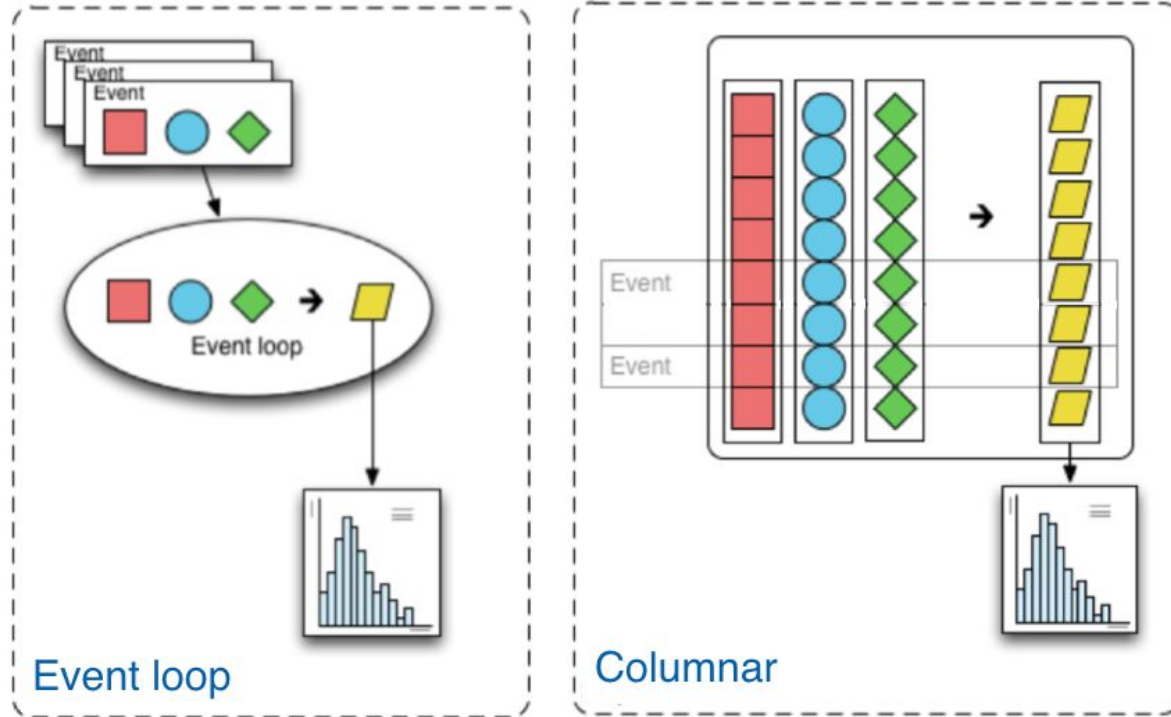
```
h = df.Filter("theta > 5").Histo1D("pt")
```

```
h.Draw()
```

2. Cortes em  
theta

3. preencher o  
histograma com o  
pt

# Técnicas de análise em FAE

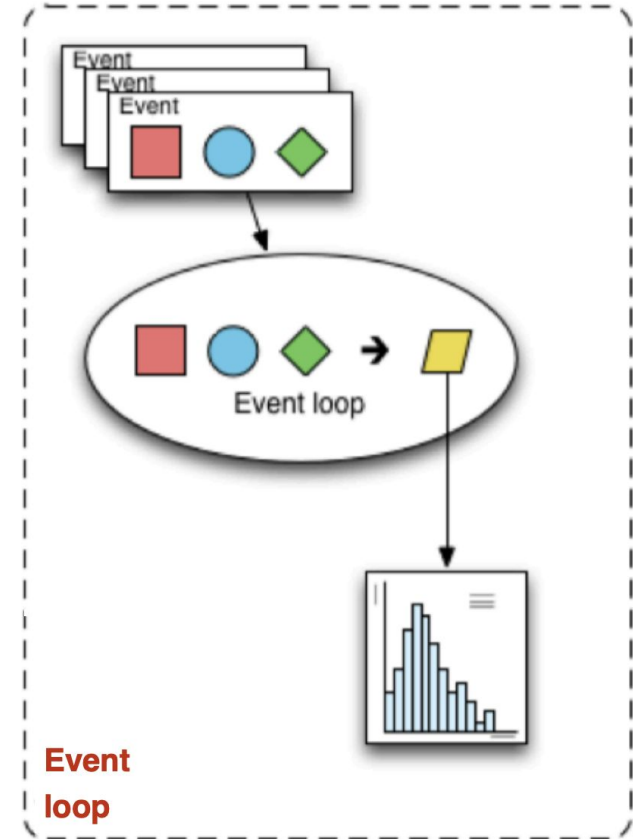


# Análise em loop de eventos - Análise orientada a registros

- Carregar valores relevantes
- Avaliar expressões
- Armazenar os valores derivado
- Repetir o processo

Os dados são processados evento a evento, realizando cálculos e armazenando resultados para análise posterior. Cada evento é processado individualmente.

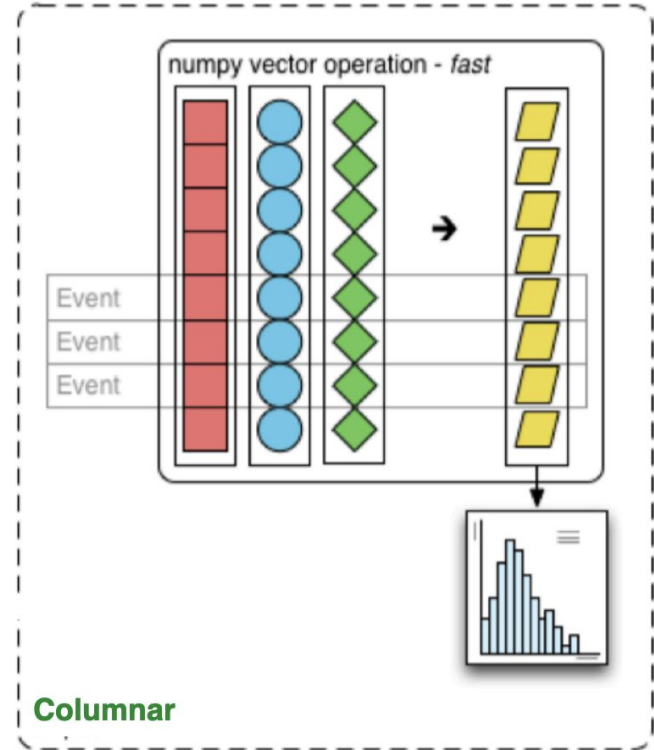
A análise é estruturada como um loop que percorre os eventos um por um. Cada "registro" ou evento é carregado na memória individualmente, e o código de análise opera sobre os campos ou propriedades desse evento, como os momentos das partículas reconstruídas.



# Análise de forma Colunar

- **Representação de dados em formato colunar:**
  - Os dados são carregados em colunas contíguas, onde cada coluna corresponde a uma variável ou observável e cada linha corresponde a um evento.
  - Você trabalha diretamente com colunas de dados, representando múltiplos eventos de uma vez.
- **Análise Colunar:**
  - Expressões vetorizadas, as operações de vetor podem ser aplicadas diretamente nas colunas de dados usando bibliotecas como NumPy.
  - Sem loops explícitos, as operações são aplicadas diretamente em vetores/matriz (ou colunas).
  - Armazenar valores derivados em novas colunas.

Aproveitando o processamento em vetores/matriz, torna o processo mais rápido e escalável.





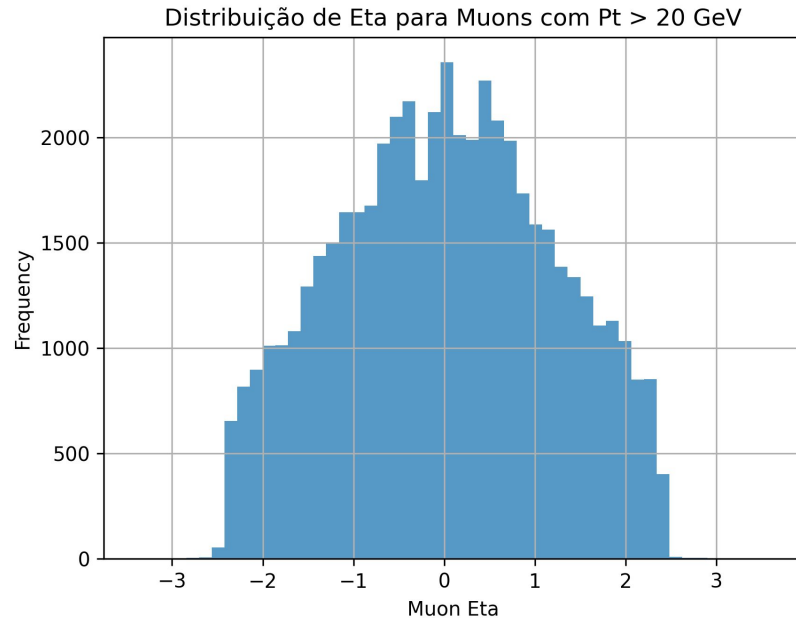
## Exemplos - Loop de eventos x Colunar

```
filtered_eta = []  
  
# Loop sobre cada evento  
for i in range(len(nMuon)):  
    for j in range(nMuon[i]):  
        if muon_pt[i][j] > 20.0:  
            filtered_eta.append(muon_eta[i][j])
```

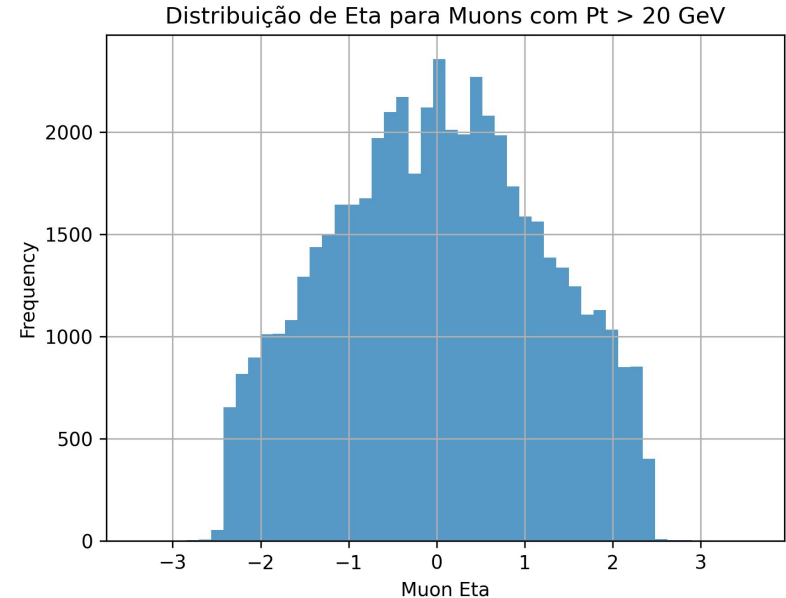
```
# Máscara para selecionar os múons com pt > 20 GeV  
mask = muon_pt > 20.0  
  
# Filtrando os eventos que atendem à condição  
filtered_eta = ak.flatten(muon_eta[mask])
```

# Exemplos - Tempo de processamento

(Loop de Eventos): 14.8031 segundos



(Análise Colunar): 0.0094 segundos



# Exemplos

```
void MyClass::Loop() {  
    size_t nEvents;  
    // load...  
  
    for (Long64_t iEvent=0; iEvent<nEvents; iEvent++) {  
        double MET_pt;  
        int nElectron;  
        double * Electron_pt;  
        double * Electron_eta;  
        // load...  
  
        if ( MET_pt > 100. ) continue;  
  
        for(size_t iEl=0; iEl<nElectron; ++iEl) {  
            if ( Electron_pt[iEl] > 30. ) {  
                hist->Fill(Electron_eta[iEl]);  
            }  
        }  
    }  
}
```

Event loop

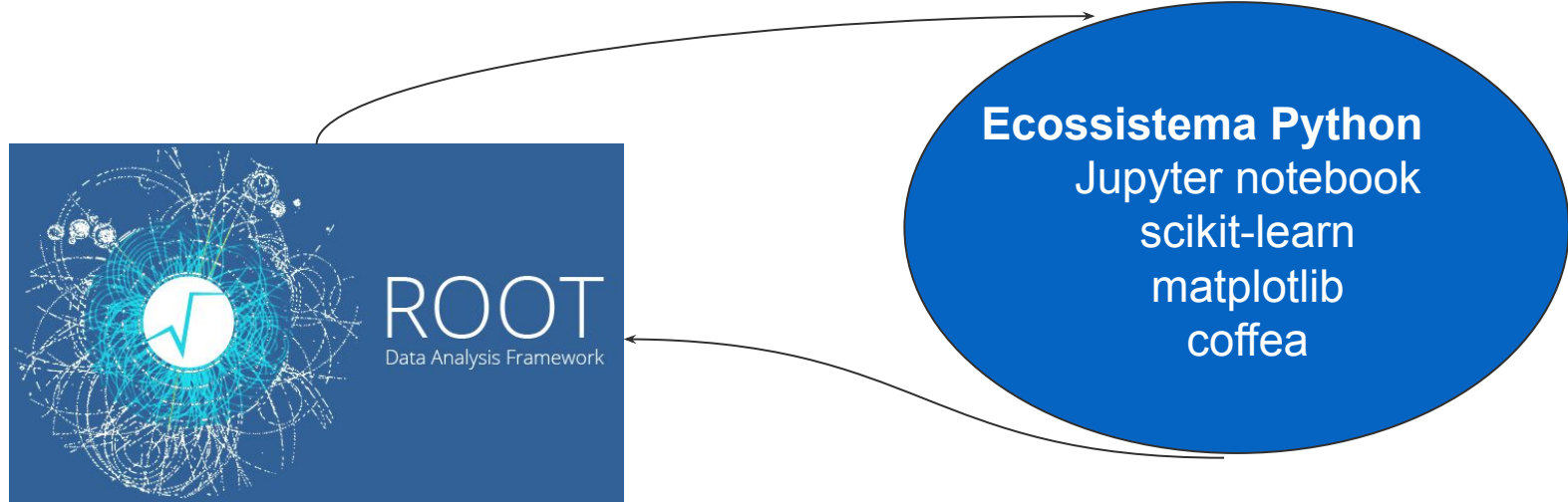
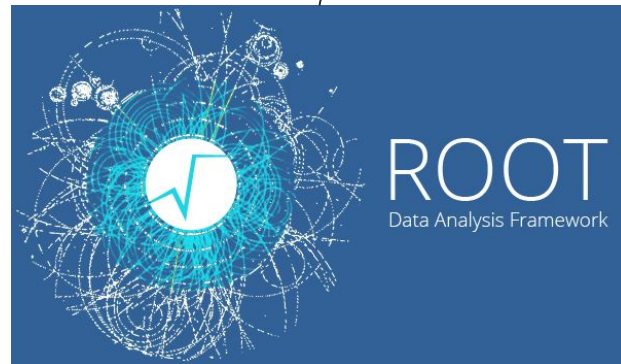
```
cut = (events.MET.pt < 100.) & (events.Electron.pt > 30.)  
hist.fill(eta=events.Electron.eta[cut].flatten())
```

Columnar

from:Lindsey Gray

## Análise de forma Colunar - Ferramentas

As ferramentas como **uproot** e **awkward array** são usadas para ler e processar os dados nesse formato. Essas bibliotecas permitem trabalhar com arquivos ROOT (TTree) como se fossem arrays nativos no Python, possibilitando operações colunares diretamente.



```
graph LR; Python([Ecosystema Python  
Jupyter notebook  
scikit-learn  
matplotlib  
coffea]) --> ROOT[ROOT Data Analysis Framework]; ROOT --> Python;
```

**Ecosystema Python**  
Jupyter notebook  
scikit-learn  
matplotlib  
coffea

# Análise de forma Colunar - Ferramentas



Visualização

Algoritmos

Array API

Ingestão de Dados

Agendador de tarefas

Provisão de recursos

Coffea

 Coffea	matplotlib		
 SciPy		 Coffea	
APACHE ARROW 	 NumPy	Awkward Array	
<a href="#">Laurelin</a> <a href="#">ServiceX</a>			
 <b>SPARK</b>	 <b>DASK</b>	 <b>Striped</b>	 <b>Parsl</b>
 <b>kubernetes</b>	<b>HTCondor</b>	 <b>slurm</b> workload manager	etc.

# Análise de forma Colunar - Ferramentas



## Análise de forma Colunar - uproot

- O uproot é um módulo Python que permite ler e escrever arquivos no formato ROOT.
- A dependência obrigatória, além do próprio Python, é o NumPy, que é a biblioteca mais popular para manipulação de arrays de dados em Python.
- Ele é capaz de processar grandes volumes de dados de forma rápida.



```
import uproot
file = uproot.open("data.root")
file
```

**Saída:** <ReadOnlyDirectory '/' at 0x(some hexadecimal number here)>

# Análise de forma Colunar - uproot

Como navegar e acessar o conteúdo do arquivo ROOT usando o uproot?

Ex.:

Listar o conteúdo do arquivo:

```
file.keys()
```

```
# Saída: ['Events;1']
```

Verificar o tipo de objeto no arquivo:

```
file.classnames()
```

```
# Saída: {'Events;1': 'TTree'}
```

Acessar o conteúdo do arquivo (file['key']):

```
file['Events']
```

```
# Saída: <TTree 'Events' (6 branches) at 0x(hexadecimal number)>
```



## Como acessar e manipular a TTree do arquivo ROOT usando o uproot?

```
tree = file['Events']
```

```
tree.keys()
```

```
#Saída: ['nMuon', 'Muon_pt', 'Muon_eta', 'Muon_phi', 'Muon_mass',  
'Muon_charge']
```

```
tree.arrays()
```

```
#Saída: <Array [{nMuon: 2, Muon_pt: [10.8, ... -1, 1]}] type='100000 *  
uint32,...'>
```

```
branches = tree.arrays()
```

```
branches['nMuon']
```

```
#Saída: <Array [2, 2, 1, 4, 4, 3, ... 0, 3, 2, 3, 2, 3] type='100000 *  
uint32'>
```

Quando usamos `tree.arrays()`, ele retorna um objeto Awkward Array.

Como acessar e manipular a TTree do arquivo ROOT usando o uproot?

```
branches['Muon_pt']
```

```
#Saída: <Array [[10.8, 15.7], ... 11.4, 3.08, 4.97]] type='100000 * var * float32'>
```

São arrays irregulares (jagged array), pois o número de entradas ('nMuon') varia por evento.

Para acessar os dados de um evento específico, você pode indexar o array como faria com qualquer array normal.

```
branches['Muon_pt'][0]
```

```
#Saída:<Array [10.8, 15.7] type='2 * float32'>
```

# Análise de forma Colunar - Awkward Array

Os dados em altas energias são estruturados de forma irregulares:

Awkward  
Array

- Não pode ser representado como uma tabela “retangular”.
  - Em diferentes eventos - números variáveis de objetos, como  $\mu/e$ /jatos/...

A representação de Arrays Irregulares (Jagged Arrays):

Muon pt: table

<b>Event 1</b>	40.2	25.6	10.2
<b>Event 2</b>	71.1	35.7	
<b>Event 3</b>	52.3		
<b>Event 4</b>	34.5	15.7	

Muon pt: jagged array

[ [ 40.2	25.6	10.2 ]	[ 71.1	35.7 ]	[ 52.3 ]	[ 34.5	15.7 ]]
	Event 1		Event 2		Event 3		Event 4

# Análise de forma Colunar - Awkward Array

Como aplicamos seleções em arrays irregulares (jagged arrays)?

- Usando máscaras.

mu_pt	=	[[ 40.2    25.6    10.2 ] [ 71.1    35.7 ] [ 52.3 ] [ 34.5    15.7 ]]
mask = (mu_pt > 30)	=	[[ T    F    F ] [ T    T ] [ T ] [ T    F ]]
mu_pt[mask]	=	[[ 40.2 ] [ 71.1    35.7 ] [ 52.3 ] [ 34.5 ]]

A máscara foi aplicada a cada múon em cada evento de maneira vetorizada.

# Análise de forma Colunar - Awkward Array

```
✓ 4s [9] 1 !pip install awkward
      2 import awkward as ak
      3 import numpy as np

      Show hidden output

✓ 0s [10] 1 mu_pt = ak.Array([[40.2, 25.6, 10.2], # Evento 1: três múons
                             [71.1, 35.7],      # Evento 2: dois múons
                             [52.3],            # Evento 3: um múon
                             [34.5, 15.7]])      # Evento 4: dois múons

✓ 0s [11] 1 print("Array original:", mu_pt)
      2 print("\nMáscara aplicada:")
      3 print(mask_mupt)
      4 print("\nArray após a aplicação da máscara:")
      5 print(mu_pt_sel)

      Array original: [[40.2, 25.6, 10.2], [71.1, 35.7], [52.3], [34.5, 15.7]]

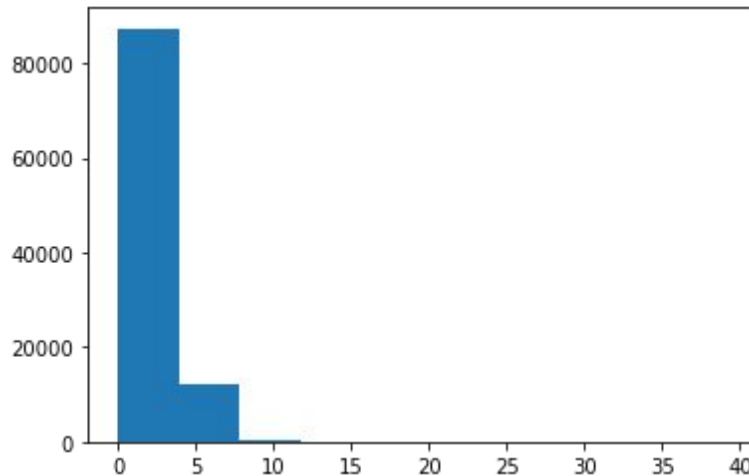
      Máscara aplicada:
      [[True, False, False], [True, True], [True], [True, False]]

      Array após a aplicação da máscara:
      [[40.2], [71.1, 35.7], [52.3], [34.5]]
```

# Análise de forma Colunar - Matplotlib

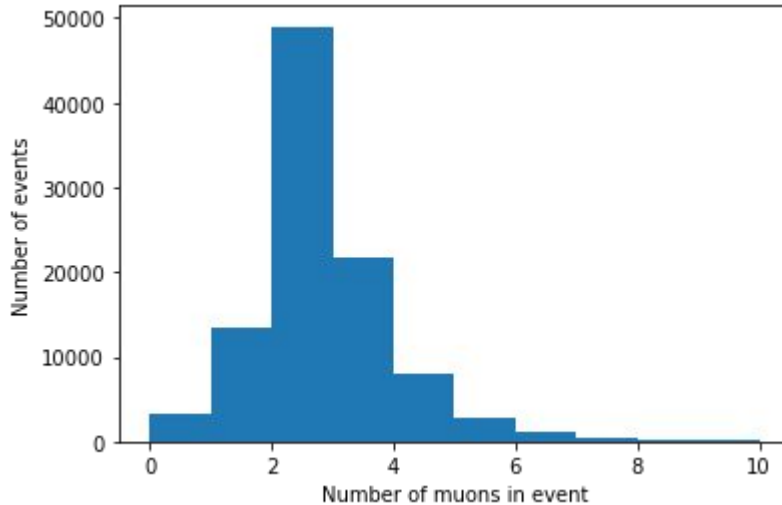
```
import matplotlib.pyplot as plt  
plt.hist(branches['nMuon'])
```

```
#Saída:(array([8.7359e+04, 1.2253e+04, 3.5600e+02,  
2.8000e+01, 2.0000e+00,1.0000e+00, 0.0000e+00,  
0.0000e+00, 0.0000e+00, 1.0000e+00]),array([ 0.,3.9,  
7.8, 11.7, 15.6, 19.5, 23.4, 27.3, 31.2, 35.1, 39. ]),  
<a list of 10 Patch objects>)
```



# Análise de forma Colunar - Matplotlib

```
plt.hist(branches['nMuon'], bins=10, range=(0, 10))  
plt.xlabel('Number of muons in event')  
plt.ylabel('Number of events')  
plt.show()
```

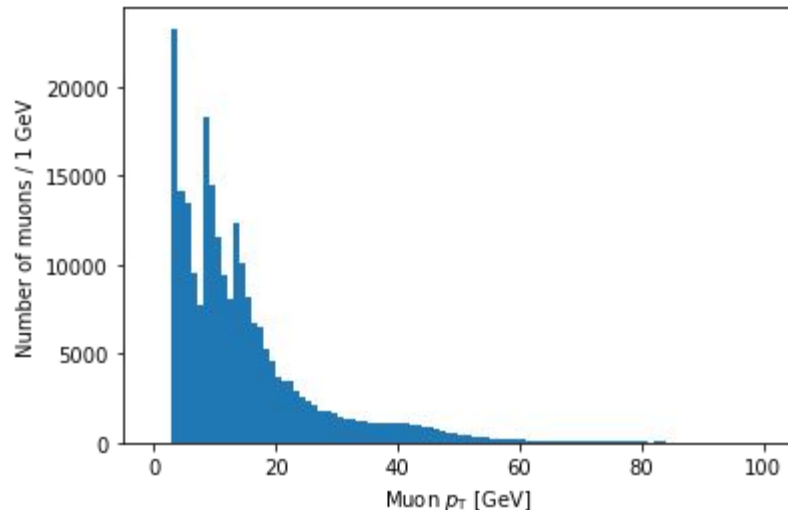


# Análise de forma Colunar - Matplotlib

Para fazer um histograma de um array irregular (jagged array), precisamos converter o array 2D para 1D usando o Awkward Array com a função `ak.flatten()`. Ex.:

```
import awkward as ak
```

```
plt.hist(ak.flatten(branches['Muon_pt']), bins=100, range=(0, 100))  
plt.xlabel('Muon  $p_T$  [GeV]')  
plt.ylabel('Number of muons / 1 GeV')  
plt.show()
```





# Análise de forma Colunar - Contagem

```
len(branches['Muon_pt']) #número total de eventos
```

```
#Saída: 100000
```

```
len(ak.flatten(branches['Muon_pt'])) #número total de múons
```

```
#Saída: 235286
```

# Análise de forma Colunar - Seleções

Para seleções:

```
branches['nMuon']== 1 # o resultado é array booleano
```

```
#Saída:<Array [False, False, True, ... False, False] type='100000 * bool'>
```

Para contar quantos eventos atendem a essa condição:

```
single_muon_mask = branches['nMuon']== 1
```

```
np.sum(single_muon_mask) #Saída: 13447
```

Para selecionar o  $p_T$  dos múons em eventos com exatamente um múon, usamos a máscara `single_muon_mask` como um índice:

```
branches['Muon_pt'][single_muon_mask]
```

```
#Saída:<Array [[3.28], [3.84], ... [13.3], [9.48]] type='13447 * var * float32'>
```

```
len(branches['Muon_pt'][single_muon_mask])
```

```
#Saída: 13447
```

# Análise de forma Colunar - Seleções



Construindo os 4-momenta dos múons:

```
two_muons_mask = branches['nMuon']== 2
```

Com o pacote [Vector](#), podemos construir os quadrimomentos dos múons a partir das variáveis  $p_T$ ,  $\eta$ ,  $\phi$ , e massa:

```
import vector  
  
muon_p4 = vector.zip({'pt': branches['Muon_pt'],  
                      'eta': branches['Muon_eta'],  
                      'phi': branches['Muon_phi'],  
                      'mass': branches['Muon_mass']})
```

```
two_muons_p4 = muon_p4[two_muons_mask]
```

Agora podemos acessar propriedades como  $p_T$ ,  $\eta$ ,  $\phi$ ,  $E$ , e  $mass$  dos múons filtrados.

# Análise de forma Colunar - Seleções

Para calcular a massa invariante dos dois múons em cada evento, somamos os quadrivetores. Primeiro, selecionamos o quadrimomento do primeiro múon em cada evento usando o slicing[:, 0]:

```
first_muon_p4 = two_muons_p4[:, 0] #primeiro múon de cada evento
```

```
second_muon_p4 = two_muons_p4[:, 1] #segundo múon de cada evento
```

Agora podemos somar os quadrivetores e calcular a massa invariante dos dois múons para cada evento.

```
sum_p4 = first_muon_p4 + second_muon_p4 #o resultado é um array 1D
```

```
two_muons_charges = branches['Muon_charge'][two_muons_mask]
```

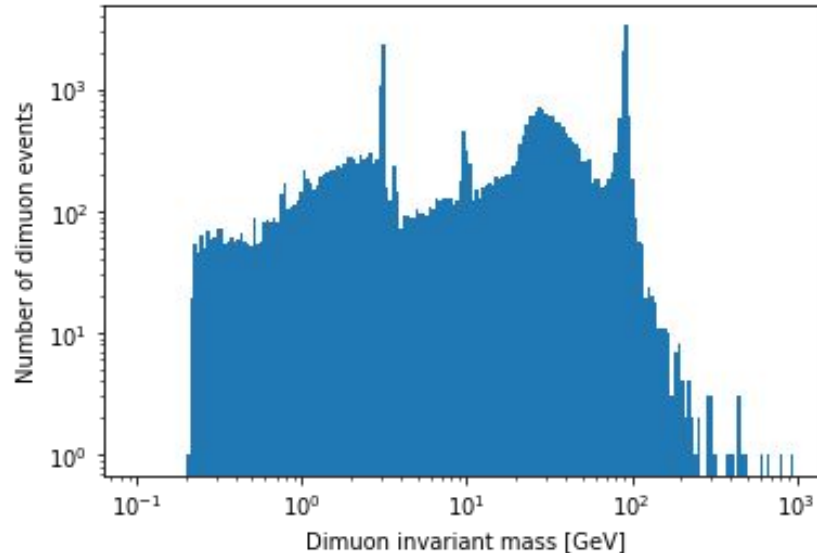
```
opposite_sign_muons_mask = two_muons_charges[:, 0] != two_muons_charges[:, 1]
```

Por fim, aplicamos essa máscara ao somatório dos quadrivetores para obter os eventos de múons de sinais opostos:

```
dimuon_p4 = sum_p4[opposite_sign_muons_mask]
```

## Massa invariante dos pares de múons de cargas opostas

```
plt.hist(dimuon_p4.mass, bins=np.logspace(np.log10(0.1), np.log10(1000), 200))  
plt.xlabel('Dimuon invariant mass [GeV]')  
plt.ylabel('Number of dimuon events')  
plt.xscale('log')  
plt.yscale('log')  
plt.show()
```



## exercício

Usando a idéia inicial do exercício da aula anterior que você calculou a  $M$  com a amostra do seu grupo:

**Aplique cortes de seleção antes de plotar a massa, vamos exigir um limiar de  $p_T$  e  $\eta$  em cada objeto e depois salvar a figura da  $M$  no formato png.**

1. O número de eventos é afetado?
2. Compare os plots de  $p_T$  e  $\eta$  antes e depois do corte .
3. Adicione as figuras e o código no git.

# Referências

- Particle physics with the computer
- The HEP Software Foundation facilitates (HSF)

# Backup

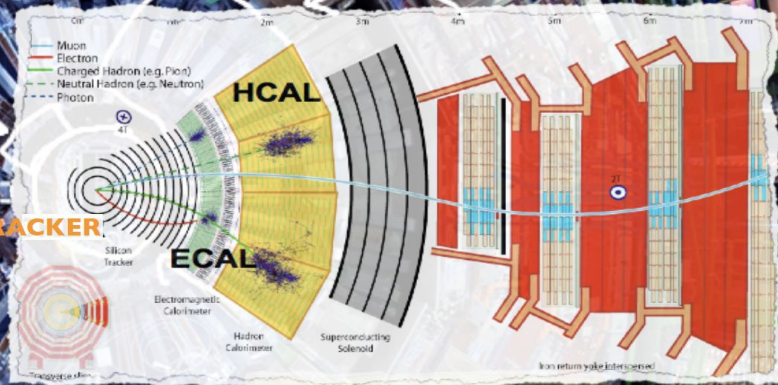


**CMS**

3.8T Superconducting Solenoid

Hermetic ( $|\eta| < 5.2$ )  
Hadron Calorimeter (HCAL)  
[scintillators & brass]

Lead tungstate  
E/M Calorimeter (ECAL)



All Silicon Tracker  
(Pixels and Microstrips)

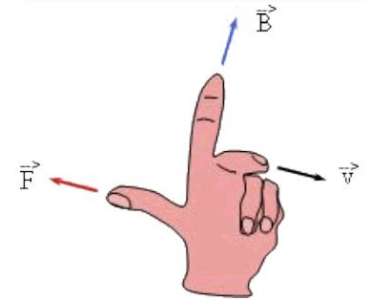
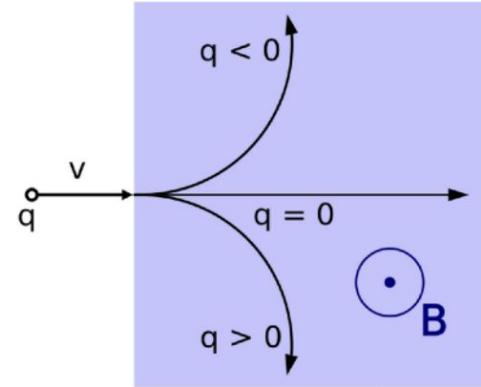
Redundant Muon System  
(RPCs, Drift Tubes,  
Cathode Strip Chambers)

# Campo magnético: no coração de uma experiência

Cargas elétricas em movimento são sensíveis aos campos magnéticos

A partir da trajetória de uma partícula sujeita a  $B$ :

- *direcção*  
⇒ medição da carga eléctrica
- *raio de curvatura*  
⇒ medida de momento (conhecida a massa)  
⇒ medida de massa (conhecida a velocidade)



$$p \cos \lambda = 0.3 z B R$$

momento linear [GeV/c]      ângulo de inclinação      carga eléctrica [e]      intensidade de campo magnético [T]      Raio de curvatura [L]