

Numpy for High Energy Physics

Romain Madar¹

¹Laboratoire de Physique de Clermont-Ferrand

January 3, 2019

Abstract

These notes describe the material presented in a numpy tutorial in the context of a working group at Laboratoire de Physique de Clermont related to machine learning and applications in physics. This tutorial is split into three parts, going from first principles to some limitations for High Energy Physics (HEP), and some possible workarounds. This tutorial reflects my current understanding and some newer/better approach might exist (feel free to [contact me!](#)). This tutorial assumes some basic knowledge of python. The code and these notes are accessible in https://github.com/MLatCezeaux/intro_numpy.

Contents

1	Short introduction to numpy	3
1.1	Motivations	3
1.2	The core object: arrays	3
1.3	The three key features of numpy	5
1.4	Two powerfull tools: matplotlib and pandas	13
2	Typical use cases in high energy physics	21
2.1	Data model and goals	22
2.2	Mean over the differents axis	23
2.3	Distance computation	26
2.4	Pairing 3D vectors for each observation, without a loop	28
2.5	Selecting a subset of r_i based on (x, y, z) values, without loop	33
2.6	Play with two collections of vectors with different size $\{r_i\}_{10}$ and $\{q_i\}_6$	41
2.7	Supplement: explanation of <code>all_pairs_nd(a,b,axis)</code>	43

3 Analysis of typical collider data and numpy limitation	46
3.1 Loading a ROOT TTree as a pandas DataFrame	47
3.2 Variable-size arrays and “squared” arrays	50
3.3 Producing some non-trivial plots using numpy arrays	56
3.4 Perform event-by-event computations without explicit loop	62
3.5 Build up a system with several collections of objects (e.g. electrons and jets) . .	68
3.6 IO between pandas/numpy and ROOT	75
3.7 Other existing tools	76

1 Short introduction to numpy

```
import numpy as np
```

1.1 Motivations

Why numpy? Because it's very well optimized (and then fast) for numerical computations in python

Why python? Many tools available from machine learning to web or string processing.

1.2 The core object: arrays

1.2.1 Main differences with usual python lists

Addition of lists/arrays and multiplication by a number:

```
# Python list and numpy arrays to play with
l1, l2 = [1, 2, 3], [3, 4, 5]
a1, a2 = np.array([1, 2, 3]), np.array([3, 4, 5])
```

```
# helper function to print title
def print_title(title):
    print('-'*len(title))
    print(title)
    print('-'*len(title))
```

```
print_title('obj1+obj2')
print('  python lists: {}'.format(l1+l2))
print('  numpy arrays: {}'.format(a1+a2))
print('\n')
print_title('obj*2')
print('  python list: {}'.format(l1*2))
print('  numpy array: {}'.format(a1*2))
```

```
-----
obj1+obj2
-----
python lists: [1, 2, 3, 3, 4, 5]
```

```
numpy arrays: [4 6 8]
```

```
---
```

```
obj*2
```

```
---
```

```
python list: [1, 2, 3, 1, 2, 3]
```

```
numpy array: [2 4 6]
```

Slicing and indexing:

```
print_title('Indexing with an integer: obj[1]')
print('  python list: {}'.format(l1[1]))
print('  numpy array: {}'.format(a1[1]))
print('\n')

print_title('Indexing with a slicing: obj[slice(1,3)]')
print('  python list: {}'.format(l1[slice(1,3)]))
print('  numpy array: {}'.format(l1[slice(1,3)]))
print('\n')

print_title('Indexing with a list of integers: obj[[0,2]]')
print('  python list: IMPOSSIBLE')
print('  numpy array: {}'.format(a1[[0,2]]))
```

```
-----
```

```
Indexing with an integer: obj[1]
```

```
-----
```

```
python list: 2
```

```
numpy array: 2
```

```
-----
```

```
Indexing with a slicing: obj[slice(1,3)]
```

```
-----
```

```
python list: [2, 3]
```

```
numpy array: [2, 3]
```

```
-----
```

```
Indexing with a list of integers: obj[[0,2]]
```

```
-----
```

```
python list: IMPOSSIBLE
```

```
numpy array: [1 3]
```

1.2.2 Main characteristics of an array

- `a.dtype`: type of data contained in the array
- `a.shape`: number of elements along each dimension
- `a.size`: total number of elements
- `a.ndim`: number of dimensions

```
a = np.array([[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
              [ 8,  9, 10, 11]])

print('a.dtype = {}'.format(a.dtype))
print('a.shape = {}'.format(a.shape))
print('a.size  = {}'.format(a.size))
print('a.ndim  = {}'.format(a.ndim))
```

```
a.dtype = int64
a.shape = (3, 4)
a.size  = 12
a.ndim  = 2
```

1.3 The three key features of numpy

1.3.1 Vectorization

The *Vectorization* is a way to make computations on numpy array **without explicit loops**, which are very slow in python. The idea of vectorization is to compute a given operation *element-wise* while the operation is called on the array itself. An example is given below to compute the inverse of 100000 numbers, both with explicit loop and vectorization.

```
a = np.random.randint(low=1, high=100, size=100000)

def explicit_loop_for_inverse(a):
    inv_a = []
    for element in a:
        inv_a.append(1./element)
    return np.array(inv_a)
```

```
# Using explicit loop
%timeit explicit_loop_for_inverse(a)
```

156 ms \pm 1.45 ms per loop (mean \pm std. dev. of 7 runs, 10 loops each)

```
# Using list comprehension
%timeit [1/x for x in a]
```

14 ms \pm 407 μ s per loop (mean \pm std. dev. of 7 runs, 100 loops each)

```
# Using vectorization
%timeit 1.0/a
```

111 μ s \pm 6.88 μ s per loop (mean \pm std. dev. of 7 runs, 10000 loops each)

Many standard functions are implemented in a vectorized way, they are call the *universal functions*, or ufunc. Few examples are given below but the full description can be found in [numpy documentation](#).

```
a = np.random.randint(low=1, high=100, size=3)
print('a          : {}'.format(a))
print('a^2        : {}'.format(a**2))
print('a/(1-a^a) : {}'.format(a/(1-a**a)))
print('cos(a)     : {}'.format(np.cos(a)))
print('exp(a)      : {}'.format(np.exp(a)))
```

```
a          : [31 94 48]
a^2         : [ 961 8836 2304]
a/(1-a^a) : [6.67813344e-18 9.40000000e+01 4.80000000e+01]
cos(a)      : [ 0.91474236  0.96945937 -0.64014434]
exp(a)      : [2.90488497e+13 6.66317622e+40 7.01673591e+20]
```

All these ufunc can work for n-dimension arrays and can be used in a very flexible way depending on the axis you are referring too. Indeed the mathematical operation can be performed over a different axis of the array, having a totally different meaning. Let's give a simple concrete example with a 2D array of shape (5,2), i.e. 5 vectors of three coordinates (x,y,z) Much more examples will be discussed in the section 2.

```
# Generate 5 vectors (x,y,z)
positions = np.random.randint(low=1, high=100, size=(5, 3))

# Average of the coordinate over the 5 observations
```

```
pos_mean = np.mean(positions, axis=0)
print(pos_mean)

# Distance to the origin sqrt(x^2+ y^2 + z^2)for the 5 observations
distances = np.sqrt(np.sum(positions**2, axis=1))
print(distances)
```

```
[49.  39.6 44.4]
[ 47.01063709  50.03998401 135.15176654 114.50764167  81.14801291]
```

1.3.2 Broadcasting

The *broadcasting* is a way to compute operation between arrays of having different sizes in a implicit (and consice) manner. Few examples are given below but more details are give in [this documentation](#).

```
# operation between shape (3) and (1)
a = np.array([1, 2, 3])
b = np.array([5])
a+b
```

```
array([6, 7, 8])
```

```
# operation between shape (3) and (1,2)
a = np.array([1, 2, 3])
b = np.array([
    [4],
    [5],
    ])
a+b
```

```
array([[5, 6, 7],
       [6, 7, 8]])
```

```
# Operation between shapes (5,3) and (3), e.g. adding an origin r0 to 10 2D
↪ vectors
data = np.random.normal(size=(5, 2))
r0 = np.array([1, 4])
print('data:\n {}'.format(data))
print('data+r0:\n {}'.format(data+r0))
```

```
data:
[[ 1.21830426 -0.18087267]
 [ 0.42314018 -1.14068326]
 [ 0.17526473  1.28811731]
 [ 0.99439608 -0.58580567]
 [ 0.96291976 -1.40336803]]
```

```
data+r0:
[[2.21830426 3.81912733]
 [1.42314018 2.85931674]
 [1.17526473 5.28811731]
 [1.99439608 3.41419433]
 [1.96291976 2.59663197]]
```

1.3.3 Working with sub-arrays: slicing, indexing and mask (or selection)

Slicing and indexing are ways to access sub-arrays in a smart way. Python allows slicing with `Slice()` object but numpy allows to push it much further with fancy indexing. Few examples are given below and for more details, please have a look to [this documentation page](#).

The basic syntax is `a[i]` to access the *i*th element. It is also possible to go from the last element using negative indices: `a[-1]` is the last element. Numpy also support array of indices. If the index array is multi-dimensional, the result will have the same dimension as the indices array.

```
a = np.random.randint(low=1, high=100, size=10)
print('a = {}'.format(a))
print('a[2] = {}'.format(a[2]))
print('a[-1] = {}'.format(a[-1]))
print('a[[1, 2, 5]] = {}'.format(a[[1, 2, 5]]))
```

```
a = [21 95 62  8 62 28 90  3  8 20]
a[2] = 62
a[-1] = 20
a[[1, 2, 5]] = [95 62 28]
```

```
# Playing with a small n-dimensional indices array: 5 arrays of 2 elements
↳ each
indices = np.arange(10).reshape(5,2)
print('indices = {}'.format(indices))
print('a[indices] = {}'.format(a[indices]))
print('\n')
```



```
# Playing with n-dimensional indices array: 2 arrays of (10, 10) arrays
indices_big = np.random.randint(low=0, high=10, size=(2, 10, 10))
print('indices_big = {}'.format(indices_big))
print('a[indices_big] = {}'.format(a[indices_big]))
```

```
indices = [[0 1]
           [2 3]
           [4 5]
           [6 7]
           [8 9]]
a[indices] = [[21 95]
             [62  8]
             [62 28]
             [90  3]
             [ 8 20]]
```

```
indices_big = [[[6 0 2 8 4 5 5 1 3 0]
               [5 4 4 6 7 3 6 1 4 9]
               [1 1 6 9 2 4 6 7 1 6]
               [6 1 9 3 1 4 6 9 0 3]
               [0 7 4 4 1 7 6 4 3 2]
               [8 5 3 0 7 2 0 2 3 3]
               [1 0 5 3 9 5 8 3 6 6]
               [4 3 8 4 9 6 4 7 5 2]
               [4 1 4 3 1 7 7 4 2 4]
               [9 3 0 4 0 9 1 4 9 9]]
              [[5 4 0 8 3 1 5 4 2 4]
               [1 0 5 2 5 5 5 9 5 1]
               [6 0 3 6 0 1 6 2 6 5]
               [6 0 8 9 8 0 4 8 0 8]
               [3 4 7 4 4 1 4 5 3 3]
               [2 6 0 3 5 5 9 5 9 9]
               [3 0 9 1 9 8 6 0 4 7]
               [6 8 9 6 9 3 7 5 3 1]
               [2 7 1 4 5 0 8 0 1 5]
               [8 7 0 1 1 3 8 2 9 6]]]
a[indices_big] = [[[90 21 62  8 62 28 28 95  8 21]
                  [28 62 62 90  3  8 90 95 62 20]
                  [95 95 90 20 62 62 90  3 95 90]
                  [90 95 20  8 95 62 90 20 21  8]
                  [21  3 62 62 95  3 90 62  8 62]
                  [ 8 28  8 21  3 62 21 62  8  8]]
```

```
[95 21 28  8 20 28  8  8 90 90]
[62  8  8 62 20 90 62  3 28 62]
[62 95 62  8 95  3  3 62 62 62]
[20  8 21 62 21 20 95 62 20 20]]

[[28 62 21  8  8 95 28 62 62 62]
 [95 21 28 62 28 28 28 20 28 95]
 [90 21  8 90 21 95 90 62 90 28]
 [90 21  8 20  8 21 62  8 21  8]
 [ 8 62  3 62 62 95 62 28  8  8]
 [62 90 21  8 28 28 20 28 20 20]
 [ 8 21 20 95 20  8 90 21 62  3]
 [90  8 20 90 20  8  3 28  8 95]
 [62  3 95 62 28 21  8 21 95 28]
 [ 8  3 21 95 95  8  8 62 20 90]]]
```

There is a quite powerful smart way to access sub-array with the synthax `a[min:max:step]`. In that way, it's very easy to take one element over two (`step=2`), or reverse the order of the array (`step=-1`). This synthax works also for n-dimensional array, where each dimension is sperated by a coma. An example is given for a 1D array and for a 3D array of (5, 2, 3) shapes (that can though of 5 observations containing each 2 3D vectors).

```
a = np.random.randint(low=1, high=100, size=10)
print('full array a          = {}'.format(a))
print('from 0 to 1: a[:2]    = {}'.format(a[:2]))
print('from 4 to end: a[4:]   = {}'.format(a[4:]))
print('reverse order: a[::-1] = {}'.format(a[::-1]))
print('all even elements: a[::2] = {}'.format(a[::2]))
```

```
full array a          = [56 40 21 48 50 83 23 61 63 29]
from 0 to 1: a[:2]    = [56 40]
from 4 to end: a[4:]   = [50 83 23 61 63 29]
reverse order: a[::-1] = [29 63 61 23 83 50 48 21 40 56]
all even elements: a[::2] = [56 21 50 23 63]
```

```
a = np.random.randint(low=0, high=100, size=(5, 2, 3))
print('a = {}'.format(a))

# Taking only the y,z values of the first vector for all observation
# - first dimension (=5 observations): `` means takes all
# - second dimension (=2 vectors): ``1`` means only the 2nd element
# - third dimensio (=3 coordinates): ``0:2`` means from 0 to 2-1, so only
↪ (x,y)
```

```

print('\nTaking only the y,z values of the first vector for all
↳ observation:')
print('a[:, 0, 0:2] = {}'.format(a[:, 0, 0:2]))

print('\nReverse the order of the 2 vector for each observation:')
print('a[:, ::-1, :] = {}'.format(a[:, ::-1, :]))

```

```

a = [[[ 6 90 70]
      [87 73 77]]

```

```

      [[40 77  5]
      [89 38 74]]

```

```

      [[45 89 45]
      [97 38  1]]

```

```

      [[69  1 81]
      [ 6 72 43]]

```

```

      [[46 45 75]
      [37 41 73]]]

```

Taking only the y,z values of the first vector for all observation:

```

a[:, 0, 0:2] = [[ 6 90]
               [40 77]
               [45 89]
               [69  1]
               [46 45]]

```

Reverse the order of the 2 vector for each observation:

```

a[:, ::-1, :] = [[[87 73 77]
                  [ 6 90 70]]

                  [[89 38 74]
                  [40 77  5]]

                  [[97 38  1]
                  [45 89 45]]

                  [[ 6 72 43]
                  [69  1 81]]

                  [[37 41 73]
                  [46 45 75]]]

```

The last part of indexing is about array *masking* or *selection*. This allows to get only element based on a give criteria, exploiting the indexing technics shown above. Indeed, a condition an array such as `a>0` will directly return an array of boolean for which each element is True or False depending on the condition. This is also very useful to replace all element with a given value. Some examples are given below.

```
a = np.random.randint(low=-100, high=100, size=(5, 3))
mask = a>0
print('a          = {}'.format(a))
print('mask       = {}'.format(mask))
print('a[mask]    = {}'.format(a[mask])) # this cannot keep the dimension, by
↳ construction
print('a*mask     = {}'.format(a*mask)) # this preserve the dimension
↳ (replacing False by 0)
print('a[~mask]   = {}'.format(a[~mask])) # The symbol ~make reverse the
↳ conditions
print('a*~mask    = {}'.format(a*~mask)) # which work for a produc too.
```

```
a          = [[ 86  63  61]
 [ -5 -56  48]
 [-53  76 -94]
 [ 66 -17  53]
 [ 13  65  27]]
mask       = [[ True  True  True]
 [False False  True]
 [False  True False]
 [ True False  True]
 [ True  True  True]]
a[mask]    = [86 63 61 48 76 66 53 13 65 27]
a*mask     = [[86 63 61]
 [ 0  0 48]
 [ 0 76  0]
 [66  0 53]
 [13 65 27]]
a[~mask]   = [ -5 -56 -53 -94 -17]
a*~mask    = [[  0  0  0]
 [ -5 -56  0]
 [-53  0 -94]
 [  0 -17  0]
 [  0  0  0]]
```

```
# Replacement of all negative values by their square
a = np.random.randint(low=-100, high=100, size=(5, 3))
print('Before: a={}'.format(a))

a[a<0] = a[a<0]**2
print('\nAfter: a={}'.format(a))
```

```
Before: a=[[-50  27  51]
 [-58 -49  93]
 [-15  11 -43]
 [ -2  44 -23]
 [-23 -67  19]]
```

```
After: a=[[2500   27   51]
 [3364 2401   93]
 [ 225   11 1849]
 [   4   44  529]
 [ 529 4489   19]]
```

1.4 Two powerfull tools: matplotlib and pandas

1.4.1 Plotting with matplotlib

matplotlib is an extremely rich package for data visualization and there is no way to cover all its features here. The goal of this section is just to give you short and practical example to plot data. Much more details can be obtained on the [webpage](#).

```
import matplotlib.pyplot as plt
%matplotlib inline
```

1.4.1.1 Example of 1D plots and histograms

```
# Data generation
x = np.random.normal(loc=-1, 1, scale=[0.5, 0.5], size=(1000,2))
y = np.sin(x)

# Figure creation
fig = plt.figure(figsize=(24, 10))

# First subplot (1 line, 2 column, 1st plot)
```

```
plt.subplot(121)
ax = plt.plot(x, y, marker='o', markersize=5, linewidth=0.0)

# Second subplot (1 line, 2 column, 2nd plot)
plt.subplot(122)
ax = plt.hist(x, bins=20) # x.shape = (1000, 2), interpreted as 2 histos of
↳ 1000 entries
```

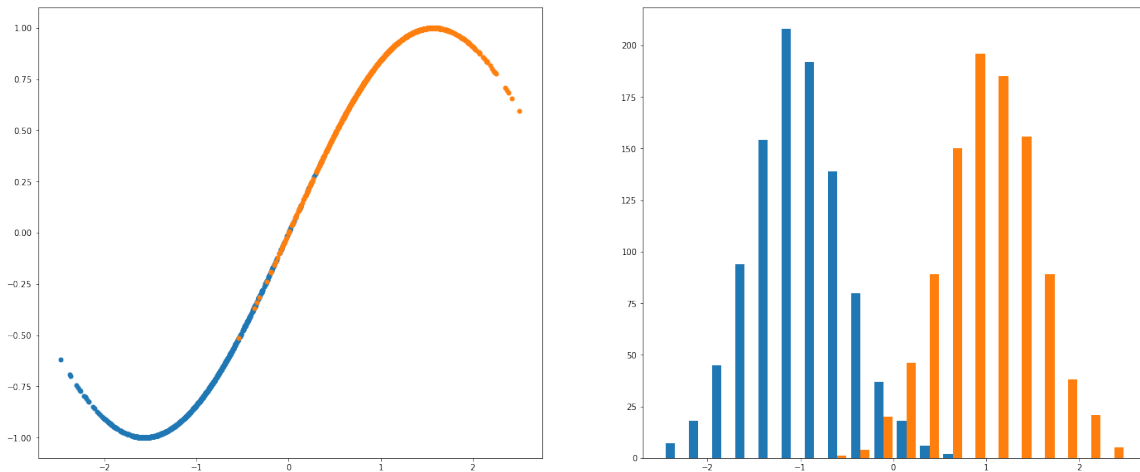


Figure 1: png

1.4.1.2 Example of 2D scatter plot

```
# Data generation
points = np.random.normal(loc=[0, 0.0], scale=[0.5, 0.8], size=(5000,2))

# plotting 2D points (x, y) with a circle size of 100*sin(x)^2
x, y = points[:, 0], points[:, 1]
fig = plt.figure(figsize=(10,7))
ax = plt.scatter(x, y, s=100*(np.sin(x))**2, marker='o', alpha=0.3)
ax = plt.xlim(-3, 3)
ax = plt.ylim(-3, 3)
```

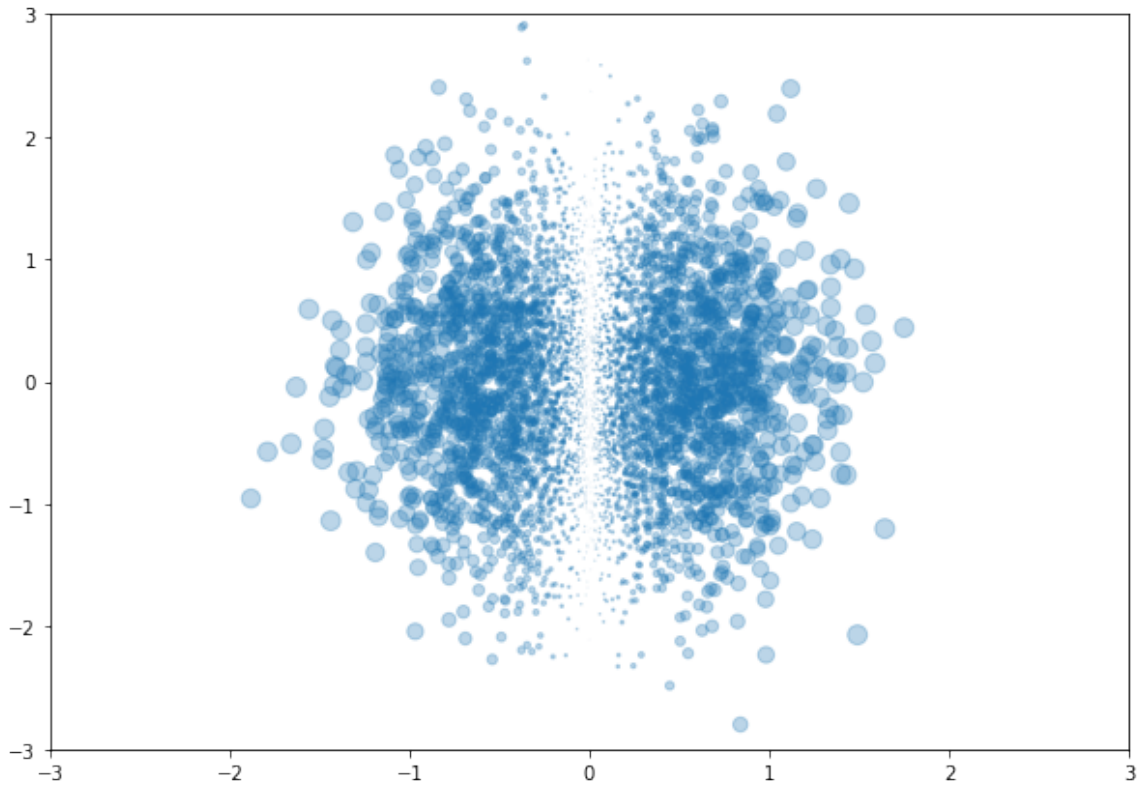


Figure 2: png

1.4.1.3 Example of 3D plots

```
# Making 3D positions and translate them using broadcasting
data = np.random.normal(size=(1000, 3))
r0 = np.array([1, 4, 2])
data_trans = data + r0

# import the needed extension to plot in 3D
from mpl_toolkits import mplot3d

# Plotting the data before and after translation
fi = plt.figure(figsize=(12,10))
ax = plt.axes(projection='3d')
_ = ax.scatter3D(data[:,0], data[:,1], data[:,2], alpha=0.4, label='before
↳ translation')
_ = ax.scatter3D(data_trans[:,0], data_trans[:,1], data_trans[:,2], alpha=0.4,
↳ label='after translation')
_ = ax.set_xlabel('x')
_ = ax.set_ylabel('y')
_ = ax.set_zlabel('z')
```

```
_ = ax.legend(frameon=False, fontsize=18)
```

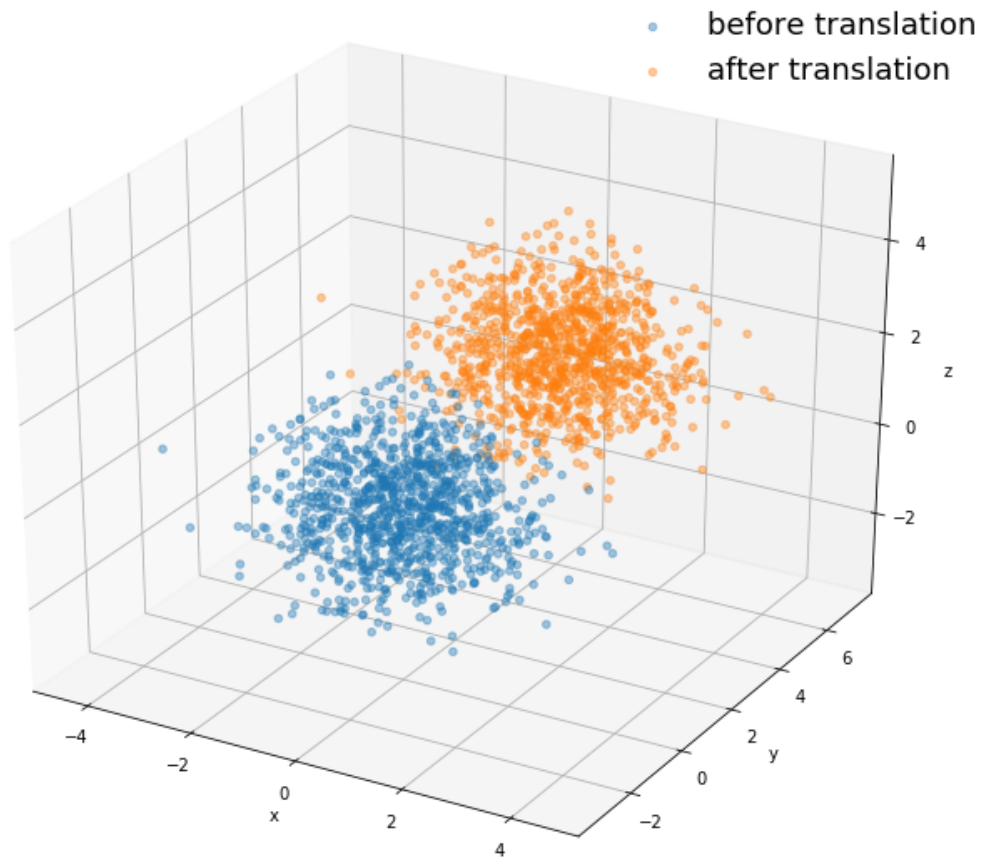


Figure 3: png

1.4.2 Import and manipulate data as numpy array via pandas

The package pandas is an very rich interface to read data from different format and produce a `pandas.dataframe` that can be based on numpy (but containing a lot more features). There is no way to fully describe this package here, the goal is simply to give functional and concrete example easily usable. More more details, please check the [pandas webpage](#)

```
import pandas as pd
cols_to_keep = ['HT', 'nlep', 'njet', 'pt_1st_bjet', 'pt_2nd_bjet',
               ↪ 'pt_3rd_bjet', 'pt_4th_bjet']
df = pd.read_csv('ttW.csv', usecols=cols_to_keep)
df.head()
```



```

<tr style="text-align: right;">
  <th></th>
  <th>HT</th>
  <th>njet</th>
  <th>nlep</th>
  <th>pt_1st_bjet</th>
  <th>pt_2nd_bjet</th>
  <th>pt_3rd_bjet</th>
  <th>pt_4th_bjet</th>
</tr>

<tr>
  <th>0</th>
  <td>262.100311</td>
  <td>2</td>
  <td>2</td>
  <td>48.112684</td>
  <td>-99.000000</td>
  <td>-99.0</td>
  <td>-99.0</td>
</tr>
<tr>
  <th>1</th>
  <td>447.937225</td>
  <td>4</td>
  <td>4</td>
  <td>118.460391</td>
  <td>28.788586</td>
  <td>-99.0</td>
  <td>-99.0</td>
</tr>
<tr>
  <th>2</th>
  <td>1287.348022</td>
  <td>6</td>
  <td>6</td>
  <td>89.715039</td>
  <td>42.138535</td>
  <td>-99.0</td>
  <td>-99.0</td>
</tr>
<tr>
  <th>3</th>
  <td>453.677887</td>
  <td>6</td>

```

```

        <td>6</td>
        <td>88.535555</td>
        <td>82.532266</td>
        <td>-99.0</td>
        <td>-99.0</td>
    </tr>
    <tr>
        <th>4</th>
        <td>268.445099</td>
        <td>2</td>
        <td>2</td>
        <td>116.625023</td>
        <td>-99.000000</td>
        <td>-99.0</td>
        <td>-99.0</td>
    </tr>

```

```

# Get a numpy arrays
ht = df['HT'].values

# Get derived quantities
ht_mean = np.mean(ht)
ht_rms = np.sqrt(np.mean((ht-ht_mean)**2))

# Add them into the pandas dataframe
df['HT_centered'] = ht-ht_mean
df['HT_normalized'] = (ht-ht_mean)/ht_rms
df.head()

```

```

<tr style="text-align: right;">
    <th></th>
    <th>HT</th>
    <th>njet</th>
    <th>nlep</th>
    <th>pt_1st_bjet</th>
    <th>pt_2nd_bjet</th>
    <th>pt_3rd_bjet</th>
    <th>pt_4th_bjet</th>
    <th>HT_centered</th>
    <th>HT_normalized</th>
</tr>

<tr>
    <th>0</th>

```

```

        <td>262.100311</td>
        <td>2</td>
        <td>2</td>
        <td>48.112684</td>
        <td>-99.000000</td>
        <td>-99.0</td>
        <td>-99.0</td>
        <td>-254.826585</td>
        <td>-0.895919</td>
    </tr>
    <tr>
        <th>1</th>
        <td>447.937225</td>
        <td>4</td>
        <td>4</td>
        <td>118.460391</td>
        <td>28.788586</td>
        <td>-99.0</td>
        <td>-99.0</td>
        <td>-68.989671</td>
        <td>-0.242554</td>
    </tr>
    <tr>
        <th>2</th>
        <td>1287.348022</td>
        <td>6</td>
        <td>6</td>
        <td>89.715039</td>
        <td>42.138535</td>
        <td>-99.0</td>
        <td>-99.0</td>
        <td>770.421127</td>
        <td>2.708646</td>
    </tr>
    <tr>
        <th>3</th>
        <td>453.677887</td>
        <td>6</td>
        <td>6</td>
        <td>88.535555</td>
        <td>82.532266</td>
        <td>-99.0</td>
        <td>-99.0</td>
        <td>-63.249009</td>
        <td>-0.222371</td>
    </tr>

```

```

</tr>
<tr>
  <th>4</th>
  <td>268.445099</td>
  <td>2</td>
  <td>2</td>
  <td>116.625023</td>
  <td>-99.000000</td>
  <td>-99.0</td>
  <td>-99.0</td>
  <td>-248.481797</td>
  <td>-0.873612</td>
</tr>

```

```

plt.figure(figsize=(20, 6))

plt.subplot(121)
ax = plt.hist(df['HT'], bins=100, alpha=0.5)
ax = plt.hist(df['HT_centered'], bins=100, alpha=0.5)

plt.subplot(122)
ax = plt.hist(df['HT_normalized'], bins=100, alpha=0.5)

```

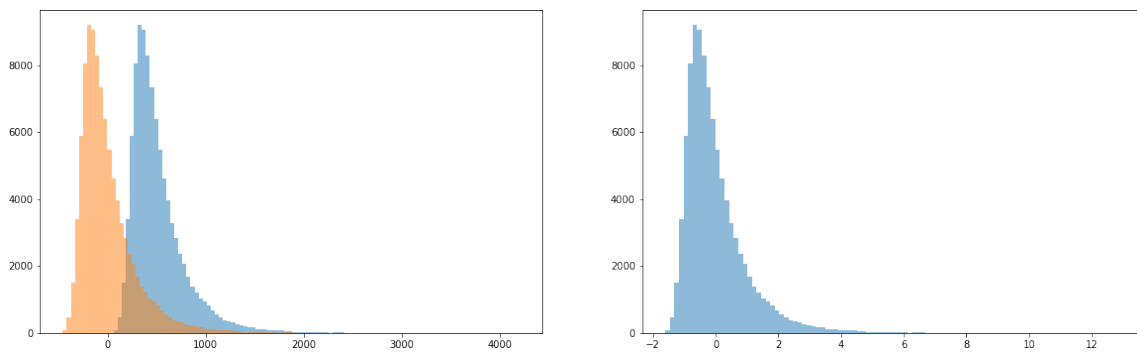


Figure 4: png

```

from pandas.plotting import scatter_matrix
df_scatter = df[['HT_normalized', 'njet', 'nlep', 'pt_1st_bjet']]
_ = scatter_matrix(df_scatter, alpha=0.2, figsize=(12, 12), diagonal='hist')

```

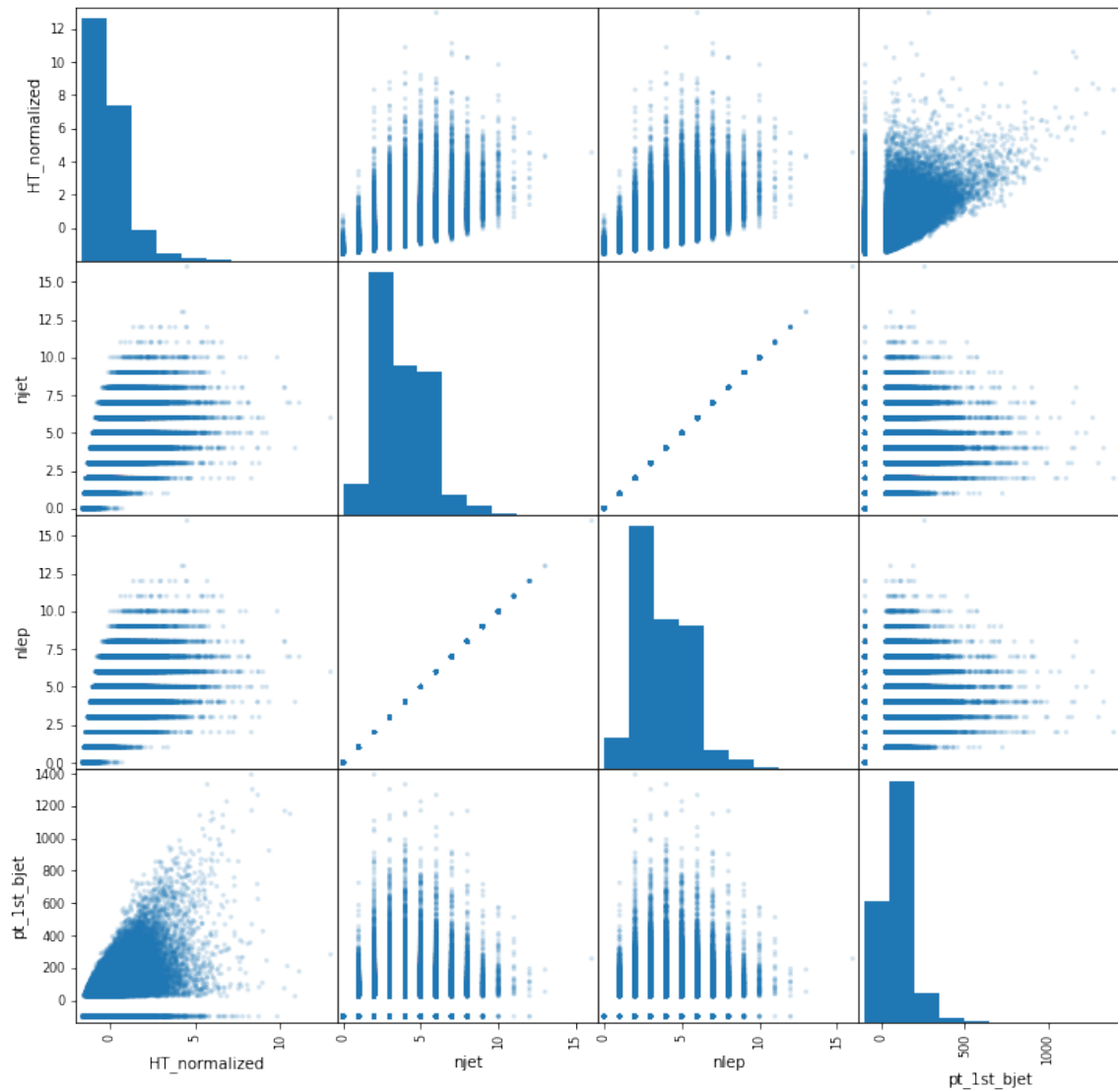


Figure 5: png

2 Typical use cases in high energy physics

```
# Imports
import numpy as np
import itertools
import matplotlib.pyplot as plt
import matplotlib as mpl
%matplotlib inline
```

```
# Plot settings
mpl.rcParams['legend.frameon'] = False
mpl.rcParams['legend.fontsize'] = 'xx-large'
mpl.rcParams['xtick.labelsize'] = 16
mpl.rcParams['ytick.labelsize'] = 16
mpl.rcParams['axes.titlesize'] = 18
mpl.rcParams['axes.labelsize'] = 18
mpl.rcParams['lines.linewidth'] = 2.5
mpl.rcParams['figure.figsize'] = (10, 7)
```

2.1 Data model and goals

We consider 1 millions of “observations”, each defined by ten 3D vectors (r_0, \dots, r_9) where $r_i = (x, y, z)$. These pseudo data can represent position in space or RGB colors. This is just an example to play with and apply numpy concepts for both simple computations (element-by-element functions, statistics calculations) and more complex calculations exploiting the multi-dimensional structure of the data. For example, one might want to compute the distance between all pairs (r_i, r_j) , which has to be done without loop.

Using the `np.random` module, it is possible to generate n-dimensional arrays easily. In our case, we want to generate pseudo-data as an array containing our observations with have 3 dimensions (or **axis** in numpy language), and the size along each of these axis will have the following value and meaning: + `axis=0`: over 1 million events + `axis=1`: over 10 vectors + `axis=2`: over 3 coordinates

```
r = np.random.random_sample((1000000, 10, 3))
print(r[0:2, ...])
```

```
[[[0.11565353 0.18948461 0.62984895]
  [0.42932845 0.32782561 0.27783779]
  [0.56957279 0.0189856 0.25712615]
  [0.70273917 0.95586083 0.26577568]
  [0.58092024 0.91899339 0.70095899]
  [0.25597834 0.28071807 0.34716292]
  [0.2790345 0.88324245 0.52832969]
  [0.65516038 0.80116358 0.09498912]
  [0.25620965 0.84625902 0.25135308]
  [0.34640387 0.29864917 0.30953939]]

 [[0.67849065 0.74025975 0.22319789]
  [0.56300543 0.73642676 0.39249035]]
```

```
[0.53774552 0.54752687 0.95504063]
[0.43430855 0.51938667 0.04538735]
[0.87616923 0.76191866 0.07106315]
[0.68232237 0.21571922 0.54843904]
[0.25458225 0.40148339 0.83364573]
[0.7959044  0.33925827 0.74188441]
[0.02103607 0.50218236 0.82084993]
[0.61883856 0.08677641 0.79843253]]]
```

2.2 Mean over the different axis

2.2.1 Mean over axis=0

This mean will average all observations over the first dimension, returning an array of dimension (10, 3) corresponding to the average $r_i = (x_i, y_i, z_i)$ over the 1 millions observations. The histogram distribution results into three separate histograms (one for each x, y, z) each having 10 entries (one per r_i)

```
%timeit np.mean(r, axis=0)
m0 = np.mean(r, axis=0)
print(m0.shape)
ax = plt.hist(m0, label=['<math>\langle x \rangle_{\text{evts}}</math>', '<math>\langle y \rangle_{\text{evts}}</math>', '<math>\langle z \rangle_{\text{evts}}</math>'])
ax = plt.title('10 entries, one for each $r_i$')
ax = plt.legend()
```

32.1 ms \pm 1.26 ms per loop (mean \pm std. dev. of 7 runs, 10 loops each)
(10, 3)

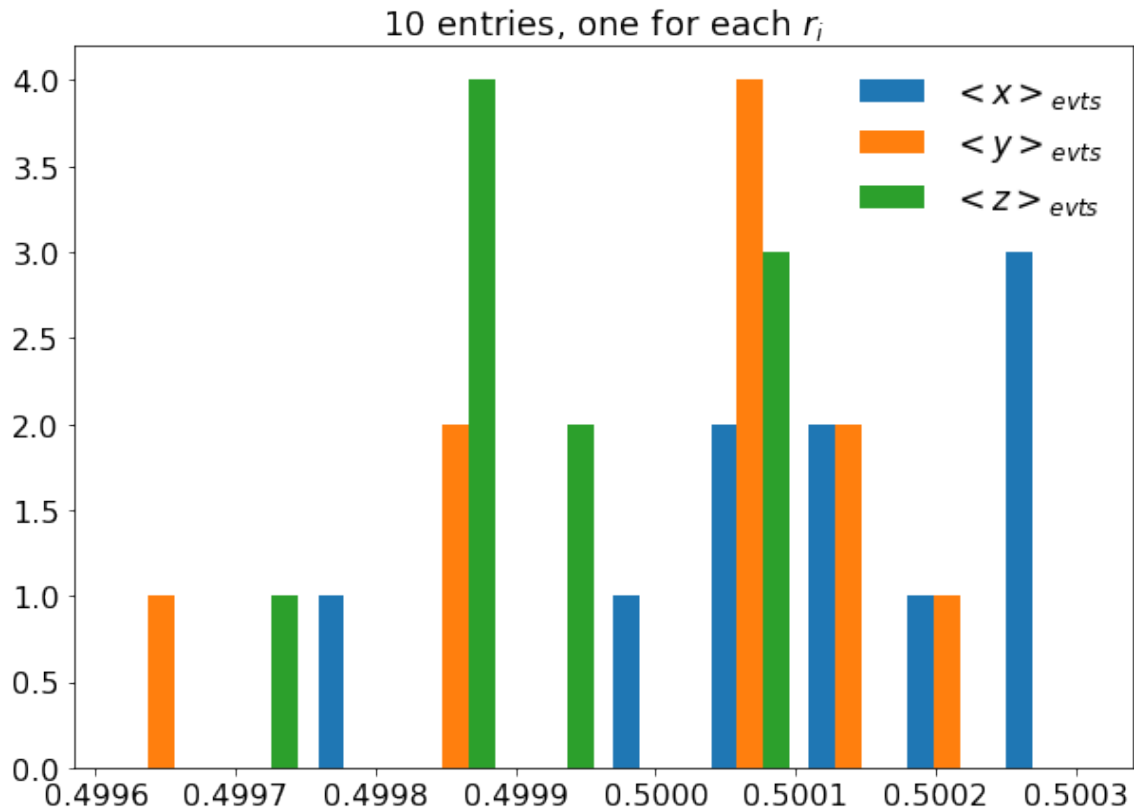


Figure 6: png

2.2.2 Mean over axis=1

This one will compute the average over the 10 vectors, for each of 1 million observations, reducing into a (1000000, 3) shape array, as seen below.

```
%timeit np.mean(r, axis=1)
m1 = np.mean(r, axis=1)
print(m1.shape)
ax = plt.hist(m1, label=['<x>_{i}', '<y>_{i}', '<z>_{i}'])
ax = plt.title('$10^6$ entries, one per event')
ax = plt.legend()
```

255 ms ± 8.14 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
(1000000, 3)

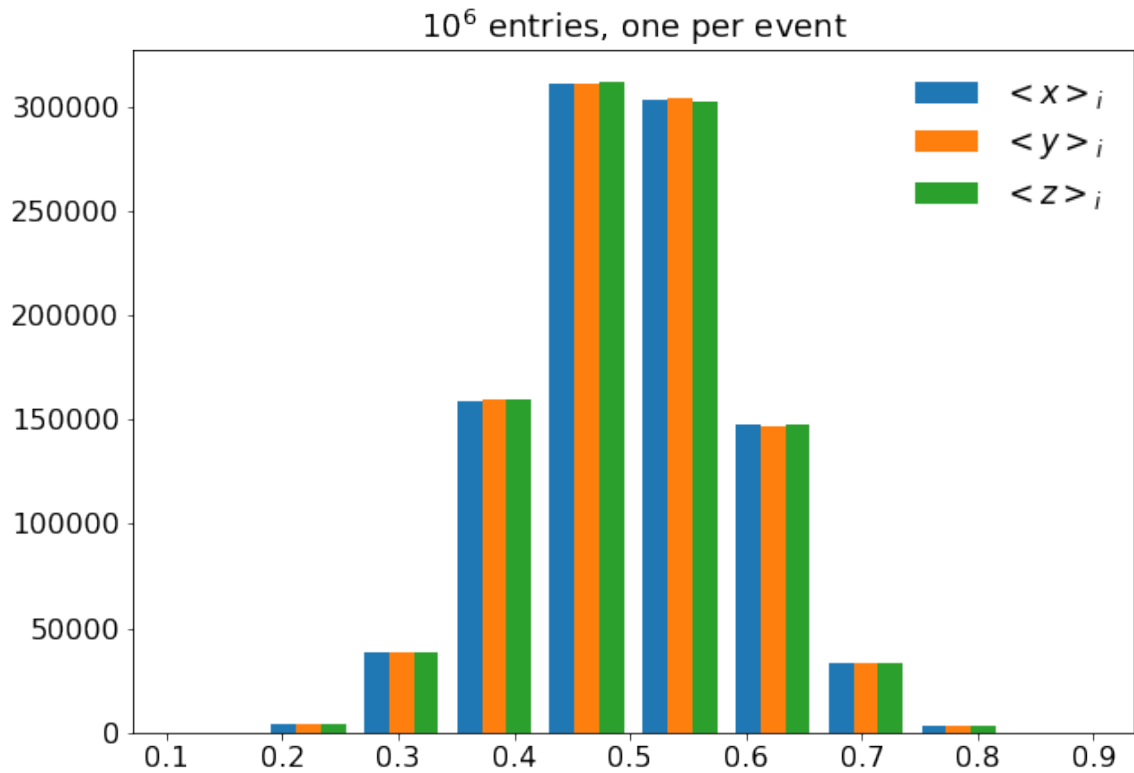


Figure 7: png

2.2.3 Mean over axis=2

This directly computes the average over the three coordinates for each vector of each event, in other words, the barycenter $(x + y + z)/3$.

```
%timeit np.mean(r, axis=2)
m2 = np.mean(r, axis=2)
print(m2.shape)
ax = plt.hist(m2, label=['$(x+y+z)/3|_{'+ '{'}.format(i)+'}'$' for i in
    ↪ range(1, 11)])
ax = plt.title('$10^6$ entries, one per event')
ax = plt.legend()
```

202 ms ± 2.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
(1000000, 10)

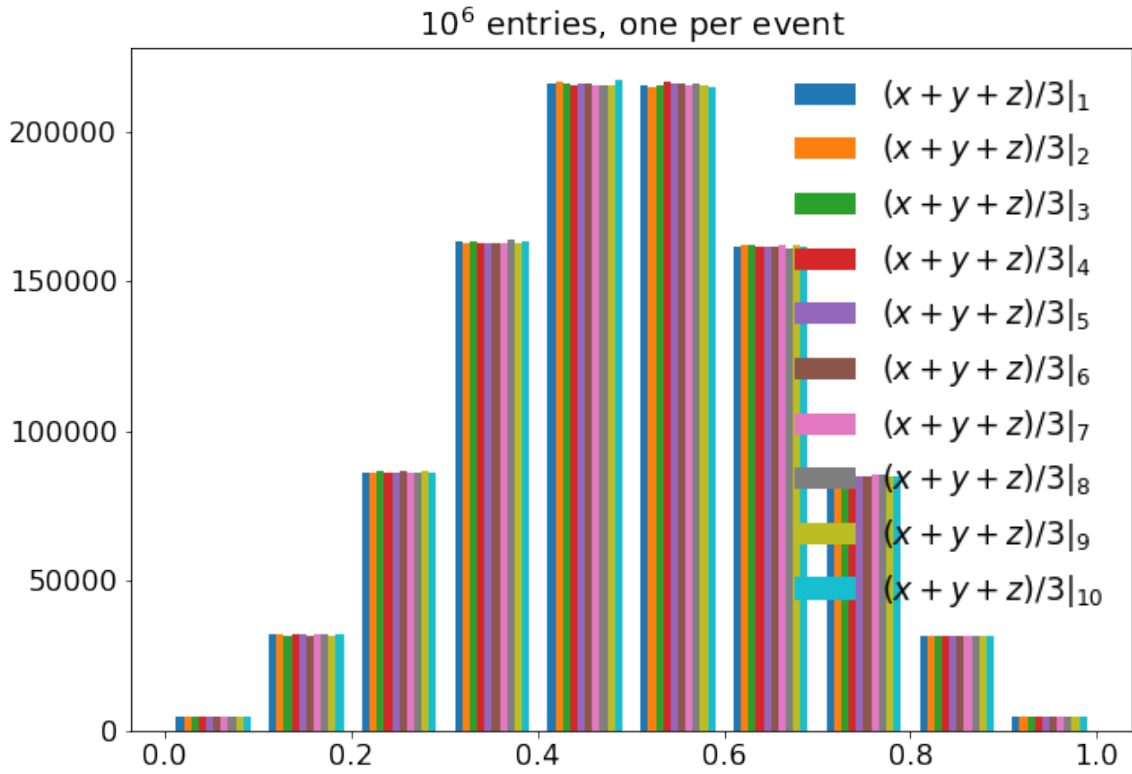


Figure 8: png

2.3 Distance computation

2.3.1 Distance to a reference r_0

```
# Reference definition
r0 = np.array([1, 2, 1])

# Compute all 10 differences for all the events
%timeit np.sum(((r-r0)**2)**0.5, axis=2)
d = np.sum(((r-r0)**2)**0.5, axis=2)
print(d.shape)
ax = plt.hist(d, label=['$d(r_{'+ '{'}.format(i)+'}',r_0)$' for i in range(1,
↪ 11)])
ax = plt.title('$10^6$ entries, one per event')
ax = plt.legend()
```

404 ms ± 9.02 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
(1000000, 10)

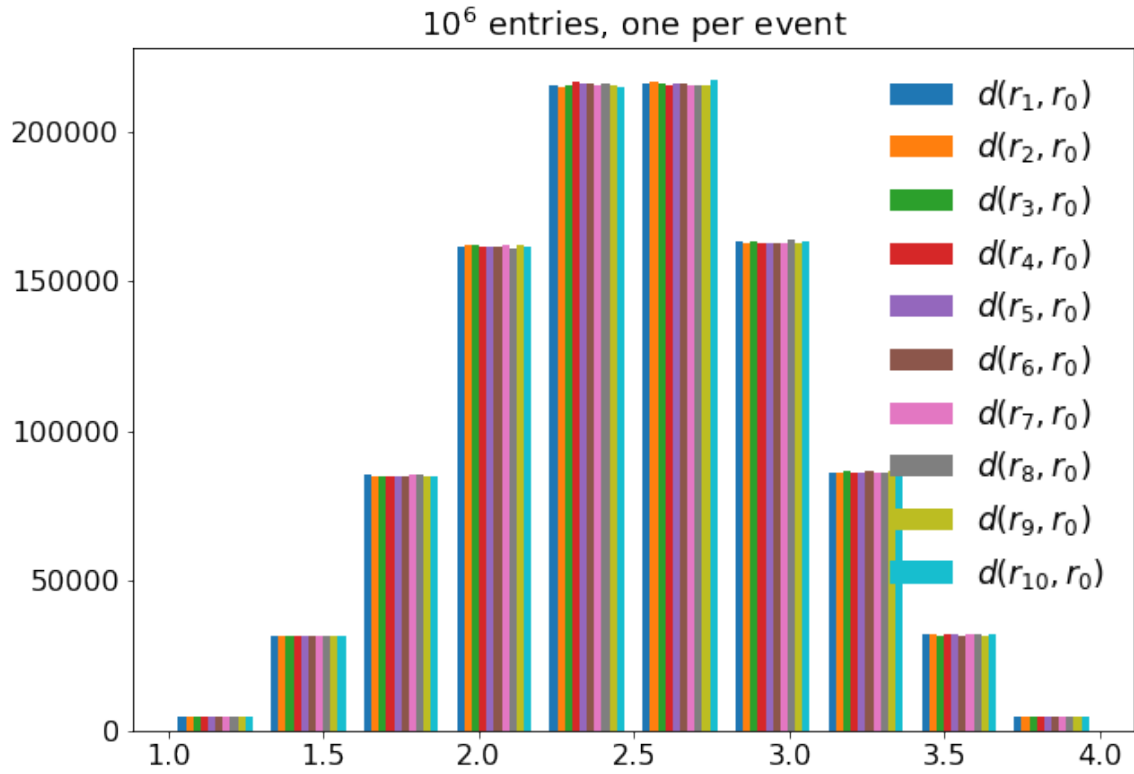


Figure 9: png

2.3.2 Distance between the two first vectors r_0 and r_1 for each event

```
# Condensed expression
%timeit np.sum(((r[:, 0, :] - r[:, 1, :])**2)**0.5, axis=1)

# More readable expression
r1, r2 = r[:, 0, :], r[:, 1, :]
d12 = np.sum(((r1 - r2)**2)**0.5, axis=1)
print(d12.shape)

# Plotting
ax = plt.hist(d12, label='$d(r_1, r_2)$', bins=100)
ax = plt.title('$10^6$ entries, one per event')
ax = plt.legend()
```

52.8 ms ± 3.01 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
(1000000,)

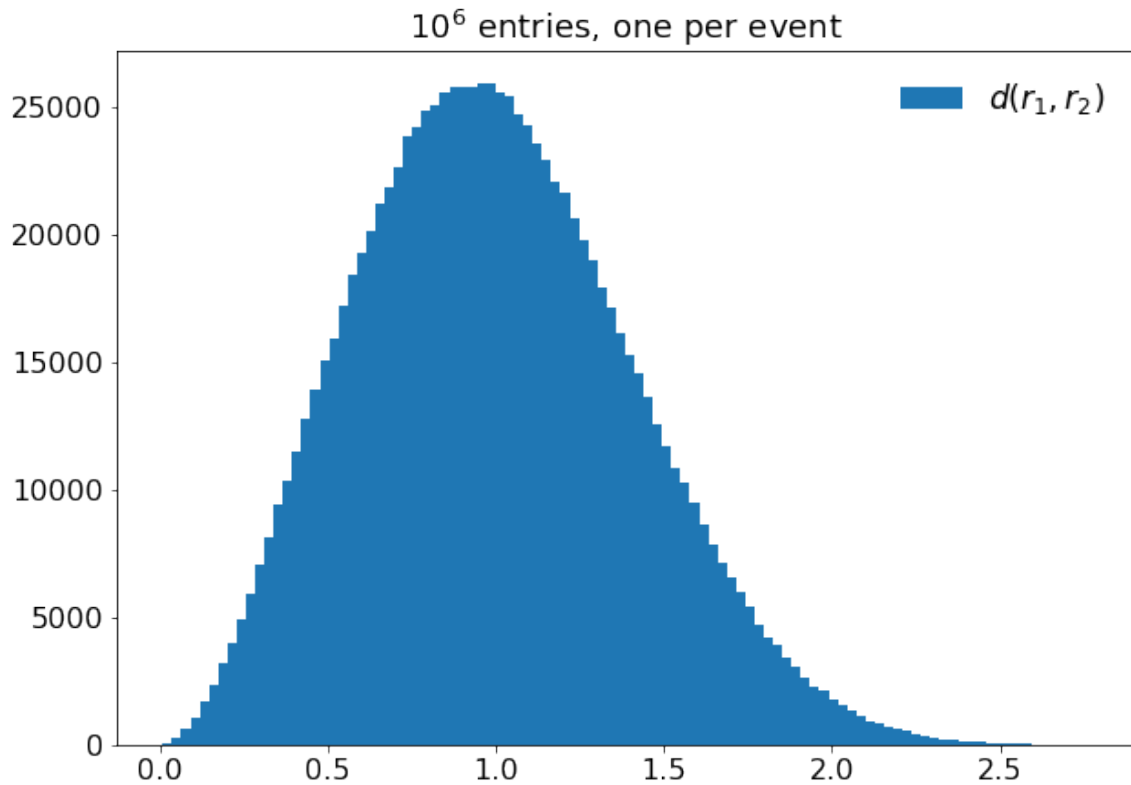


Figure 10: png

2.4 Pairing 3D vectors for each observation, without a loop

2.4.1 Finding all possible (r_i, r_j) pairs for all events

```
def combs_nd(a, n, axis=0, info=False):
    """
    Solution found on
    ↪ https://stackoverflow.com/questions/16003217/n-d-version-of-itertools-combinations-in-numpy

    The idea here is to simply work on indices to build the pairs
    since it doesn't really matter what are the nature of the objects ...
    """

    # 1. Initialisation of indices array along the axis we want to pair
    indices = np.arange(a.shape[axis])
    if info:
        print('initialisation -> indices={}'.format(indices))

    # 2. Datatype of index array ([int,int] for a pair)
```

```

dt = np.dtype([(' ', np.intp)]*n)
if info:
    print('datatype: {}'.format(dt))

# 3. Use itertools to compute combinations and overwrite indices
indices = np.fromiter(itertools.combinations(indices, n), dt) # [(0,1),
↳ (0,2), .. ]
if info:
    print('np.fromiter -> indices={}'.format(indices))
indices = indices.view(np.intp) # [0 1 0 2 ...]
if info:
    print('indices.view -> indices={}'.format(indices))
indices = indices.reshape(-1, n) # [[0 1], [0 2], ...]
if info:
    print('indices.reshape -> indices={}'.format(indices))

# 4. Take all elements in a defined by indices along a given axis
# the dimension of the array is changed because indices has (n,2) shape
return np.take(a, indices, axis=axis)

# Trying on the 2 first observations considering only the 5 first vectors ri:
↳ r[0:1, 0:5]
result = combs_nd(a=r[0:1,0:5], n=2, axis=1, info=True)

```

```

initialisation -> indices=[0 1 2 3 4]
datatype: [('f0', '<i8'), ('f1', '<i8')]
np.fromiter -> indices=[(0, 1) (0, 2) (0, 3) (0, 4) (1, 2) (1, 3) (1, 4) (2,
3) (2, 4) (3, 4)]
indices.view -> indices=[0 1 0 2 0 3 0 4 1 2 1 3 1 4 2 3 2 4 3 4]
indices.reshape -> indices=[[0 1]
[0 2]
[0 3]
[0 4]
[1 2]
[1 3]
[1 4]
[2 3]
[2 4]
[3 4]]

```

2.4.2 Computing (minimum) distances on these pairs

```
# Time and get the pair function
print('\nGetting all pairs')
%timeit combs_nd(r, 2, axis=1)
pairs = combs_nd(r, 2, axis=1)
print(pairs.shape)

# Time and Get the euclidian distance of all pair
print('\nGetting all euclidian distances')
dp = pairs[:, :, 0, :] - pairs[:, :, 1, :]
%timeit(np.sum(dp**2, axis=2))*0.5
diff_pairs = (np.sum(dp**2, axis=2))*0.5
print(dp.shape)
print(diff_pairs.shape)

# Time and get the minimum
print('\nGetting the minimum distances')
%timeit np.min(diff_pairs, axis=1)
closest_pair = np.min(diff_pairs, axis=1)
print(closest_pair.shape)
```

Getting all pairs

870 ms ± 35.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
(1000000, 45, 2, 3)

Getting all euclidian distances

1.09 s ± 40.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
(1000000, 45, 3)
(1000000, 45)

Getting the minimum distances

69.1 ms ± 1.48 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
(1000000,)

```
def compute_dr_min(a):
    pairs = combs_nd(a, 2, axis=1)

    # Get the axis of the pair index to build p1, p2 = a[...], a[...],1]
    d = pairs.ndim
    i1 = tuple([None if i != d-2 else 0 for i in range(0, d)])
    i2 = tuple([None if i != d-2 else 1 for i in range(0, d)])
```

```

return np.min(np.sum((pairs[i1]-pairs[i2])**2, axis=2)**0.5, axis=1)

# Time and get the minimum from r directly
print('\nGetting the minimum distances from r directly using one function')
%timeit compute_dr_min(r)

```

Getting the minimum distances from r directly using one function
 838 ms ± 9.78 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

```
ax = plt.hist(diff_pairs)
```

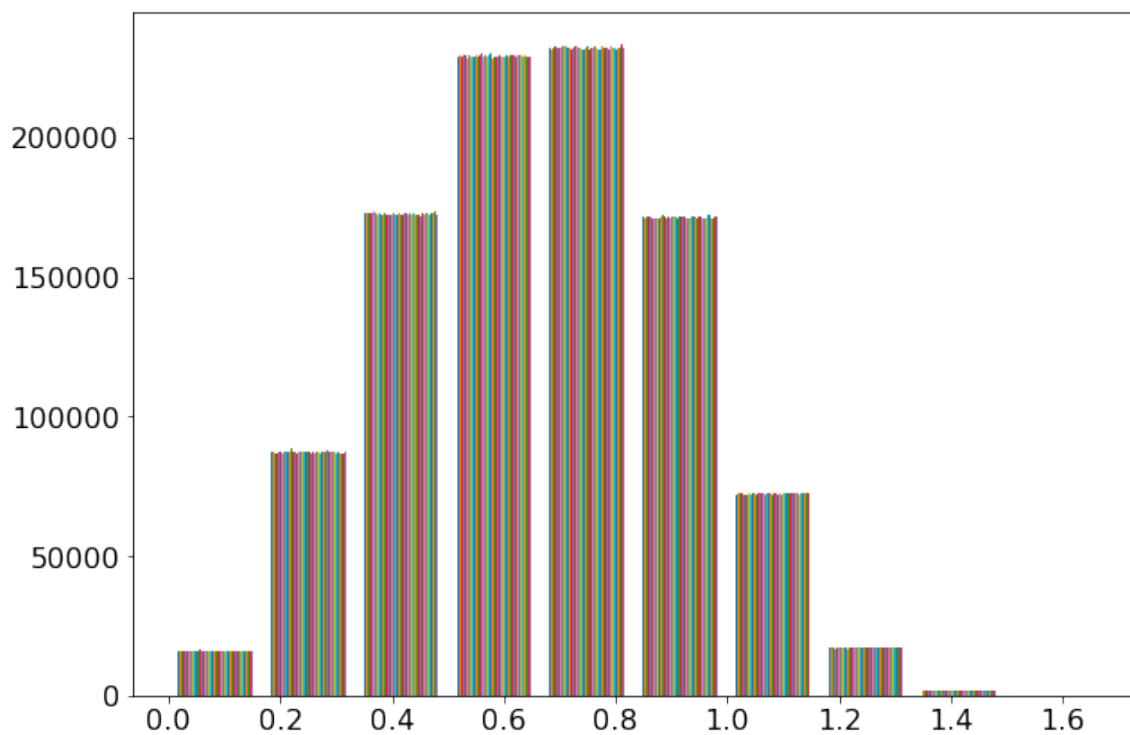


Figure 11: png

```
ax = plt.hist(diff_pairs.flatten(), bins=100)
```

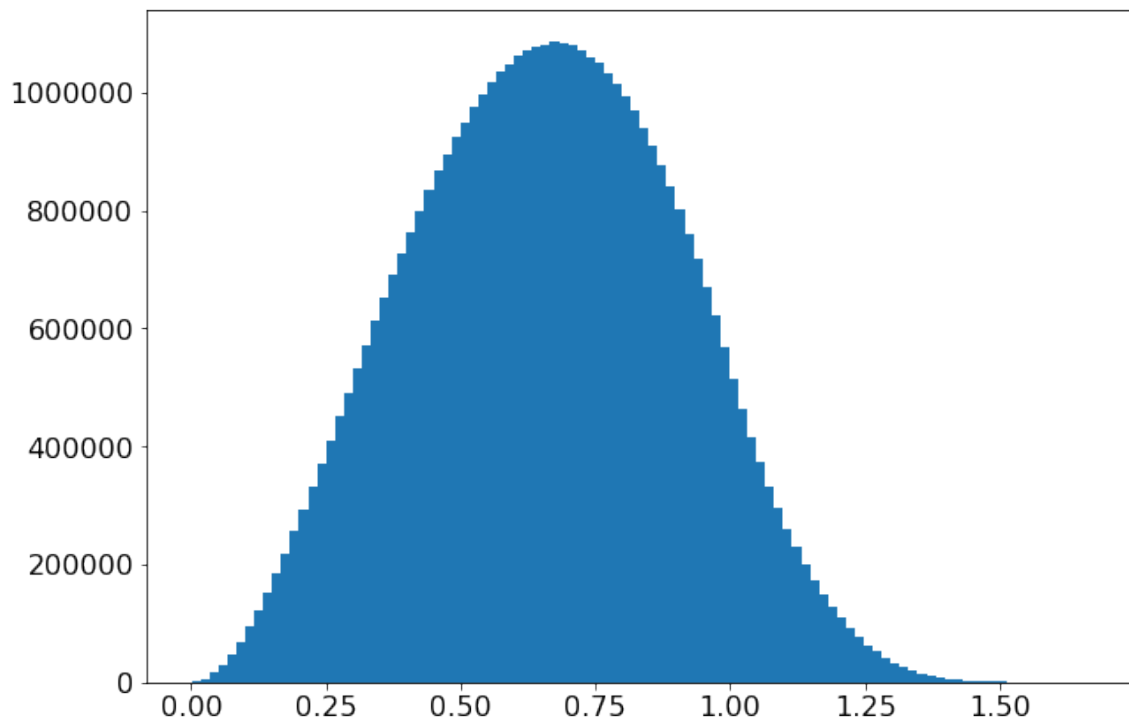


Figure 12: png

```
ax = plt.hist(closest_pair, bins=100)
```

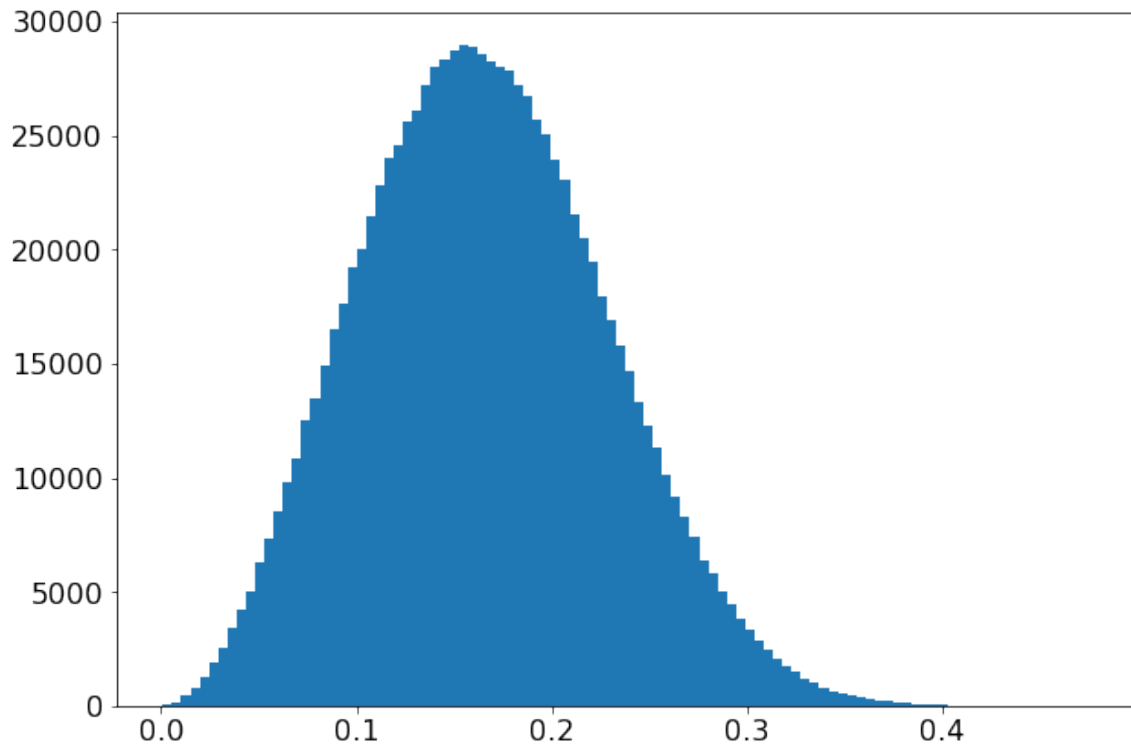



Figure 13: png

2.5 Selecting a subset of r_i based on (x, y, z) values, without loop

```
x, y, z = r[:, :, 0], r[:, :, 1], r[:, :, 2]  
ax = plt.hist(x**2+y**2+z**2, bins=10)
```

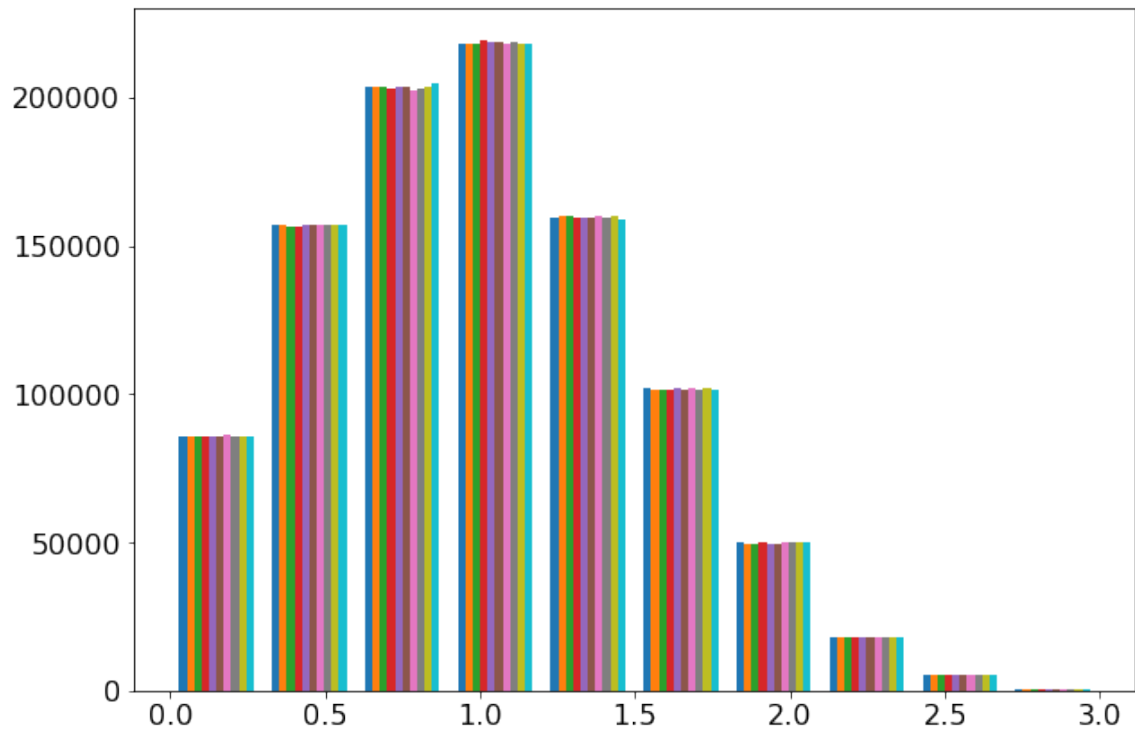


Figure 14: png

```
ax = plt.hist(np.sum(r**2, axis=2), bins=10)
```

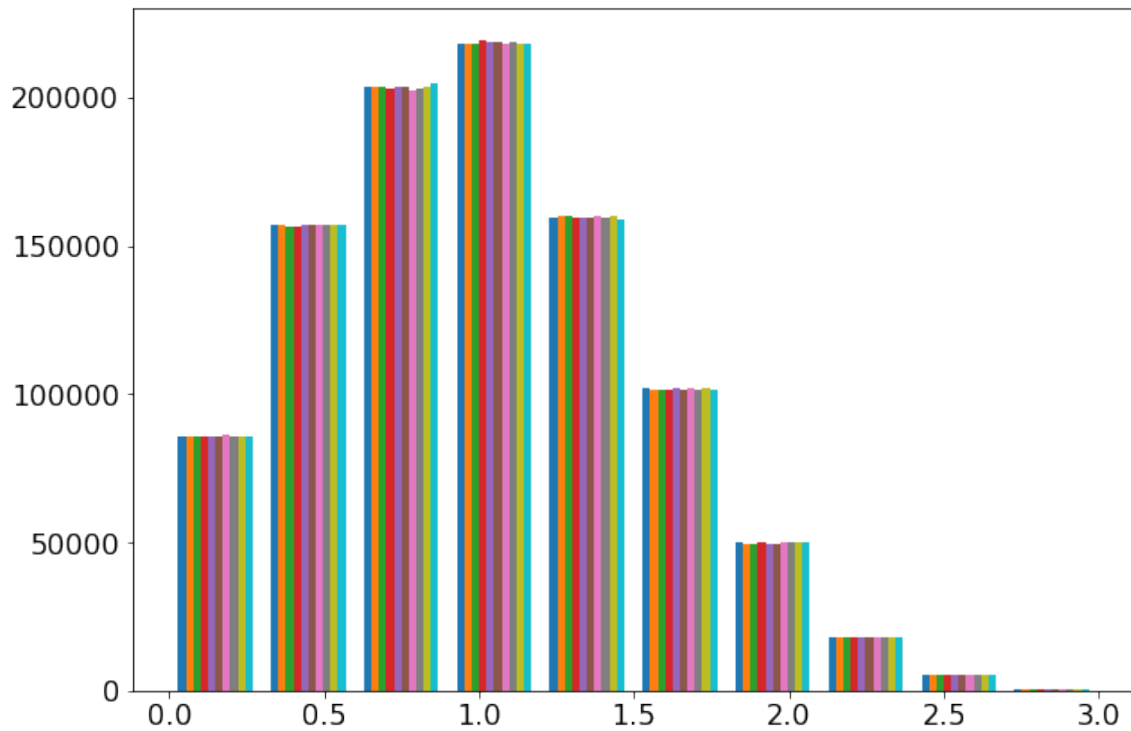


Figure 15: png

2.5.1 Counting number of points among the 10 with $x_i > y_i$ in each event

```
# define the selection
idx = x > y
print(idx.shape)

# Checkout the distribution of x,y,z for the selected points
ax = plt.hist(x[idx], bins=100, alpha=0.2)
ax = plt.hist(y[idx], bins=100, alpha=0.2)
ax = plt.hist(z[idx], bins=100, alpha=0.3)
```

```
(1000000, 10)
```

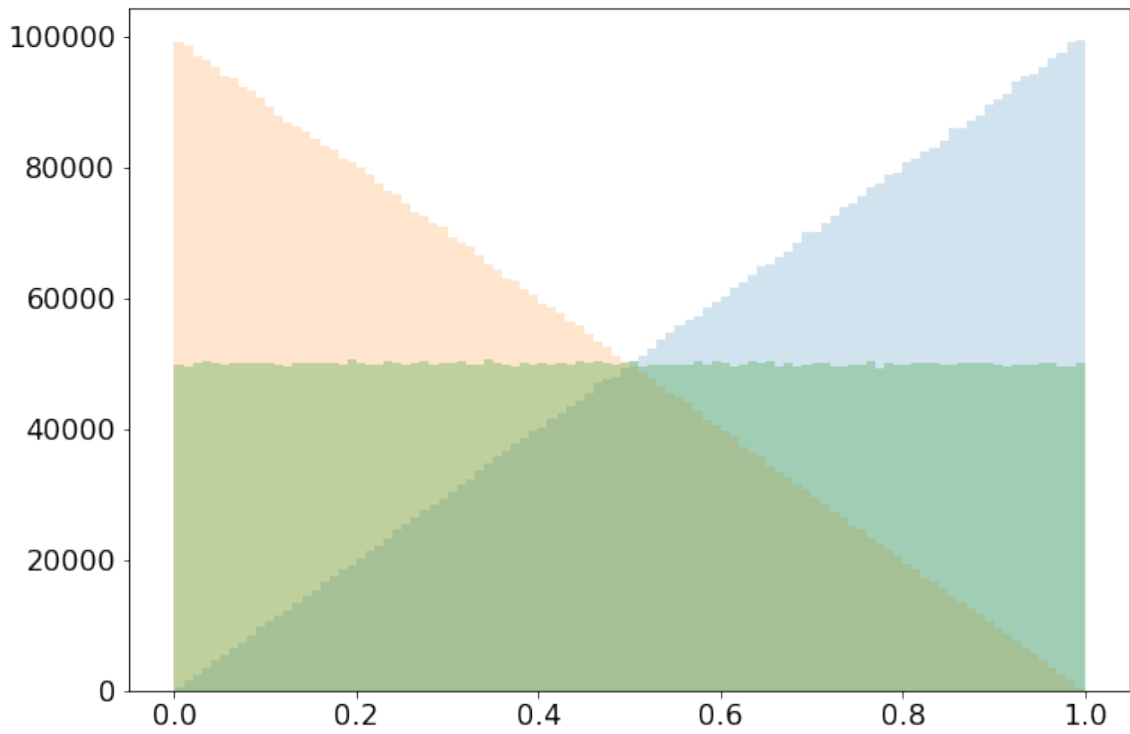


Figure 16: png

```
# Count the number of r per event satisfying x>y
c = np.count_nonzero(idx, axis=1)
print(c.shape)

# Plot the distribution of the count
ax = plt.hist(c, bins=20, alpha=0.5)
```

```
(1000000,)
```

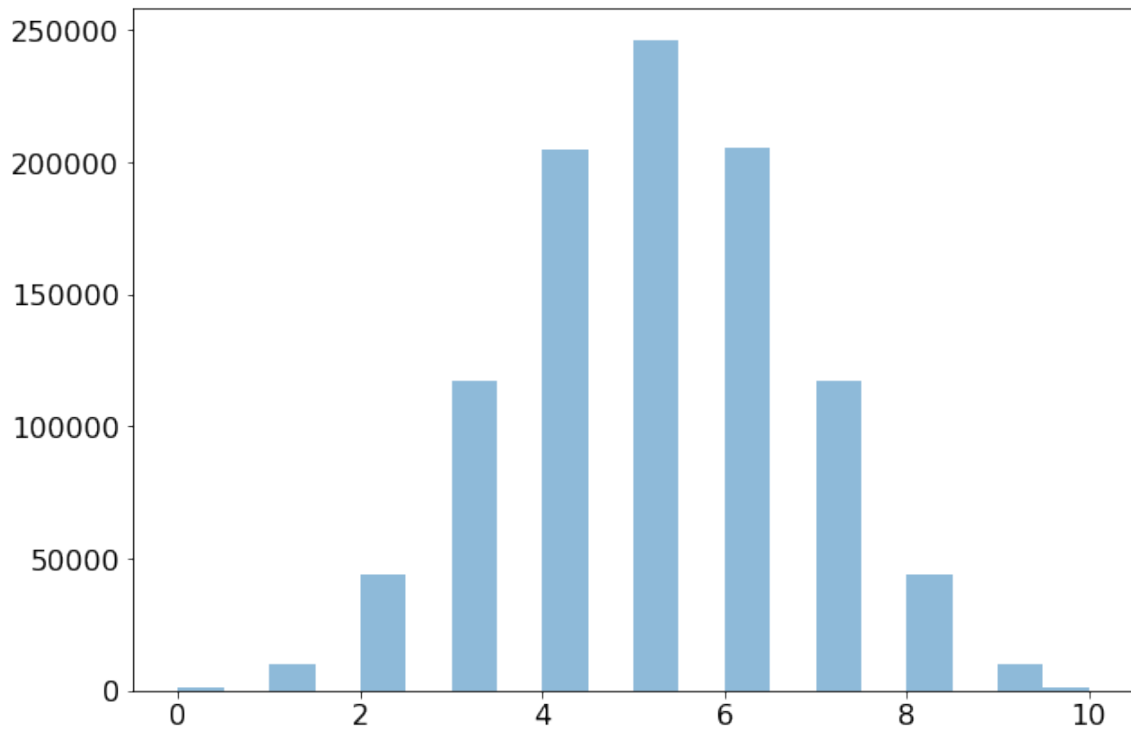


Figure 17: png

2.5.2 Plotting z for the two types of population ($x > y$ and $x < y$)

```
# Access xi and yi which satisfy xi>yi and xi<xi for each events (over 500
↳ events)
# using proper indexing
sx, sy, sz = x[0:500, ...], y[0:500, ...], z[0:500, ...]
sidx_gt, sidx_lt = sx > sy, sx < sy

# Taking the proper indexing will flatten the array
print('\n -> Full array shape    = {}'.format(sx.shape))
print(' -> Indexed array shape = {}'.format(sx[sidx_gt].shape))
print(' -> Counting *all* selected pairs:
↳ {}'.format(np.count_nonzero(sidx_gt)))

# Plotting x vs z for the two populations (marker size is 1/(z+0.001))
ax = plt.scatter(sx[sidx_gt], sy[sidx_gt], s=(sz[sidx_gt]+1e-3)**-1)
ax = plt.scatter(sx[sidx_lt], sy[sidx_lt], s=(sz[sidx_lt]+1e-3)**-1)
```

```
-> Full array shape    = (500, 10)
-> Indexed array shape = (2511,)
```

-> Counting **all** selected pairs: 2511

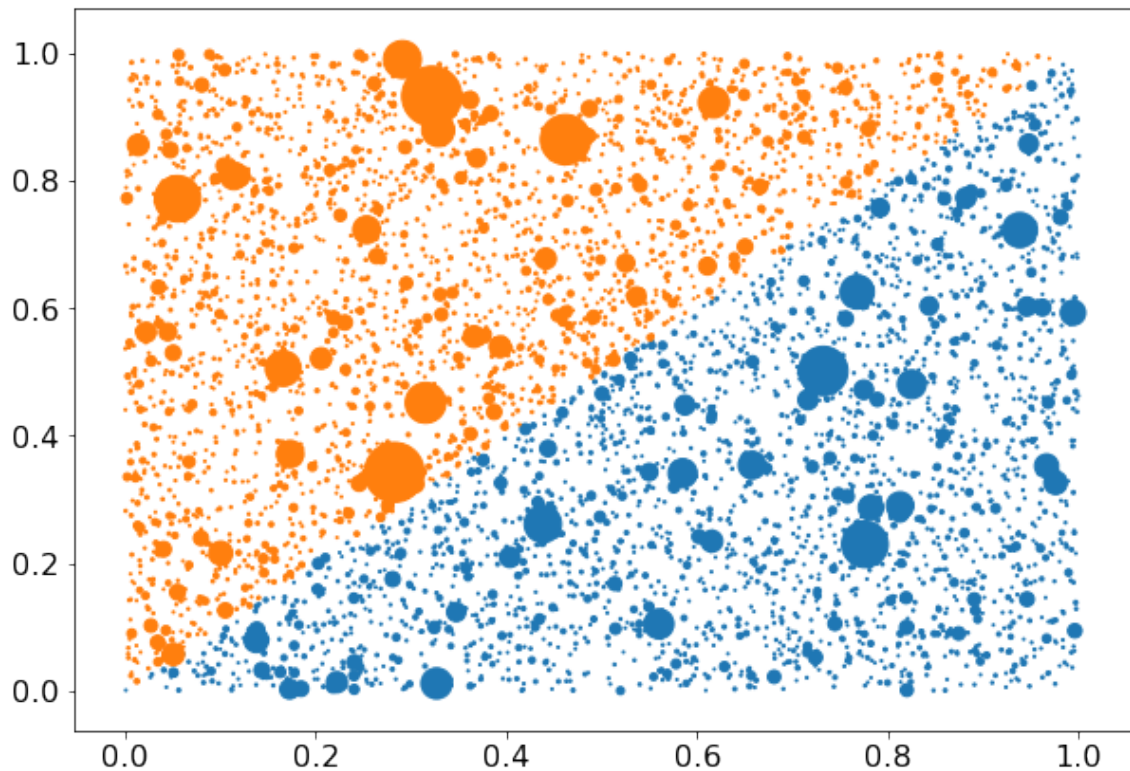


Figure 18: png

2.5.3 Compute $x_i + y_i + z_i$ sum over the collection of 10 r_i including only points that have $x_i > y_i$?

```
# First step: basic sum over 10 points, ie sum_{i=1..10}(xi+yi+zi) for each
↳ event.
ht1 = np.sum(x+y+z, axis=1)
print(ht1.shape)
ax = plt.hist(ht1, bins=100, alpha=0.4)

# Second step: doing the proper sum, ie only with points verifying x>y.
# 'x*selection' replace xi by 0 in [x0,...,x9]_evt where x<y for all
# events evt. They must have the same shape to be multiplied element
# by element.
selection = x>y
ht2 = np.sum((x+y+z)*selection, axis=1)
print(x.shape, selection.shape, ht2.shape)
ax = plt.hist(ht2[ht2 > 0], bins=100, alpha=0.4)
```

```
(1000000,)  
(1000000, 10) (1000000, 10) (1000000,)
```

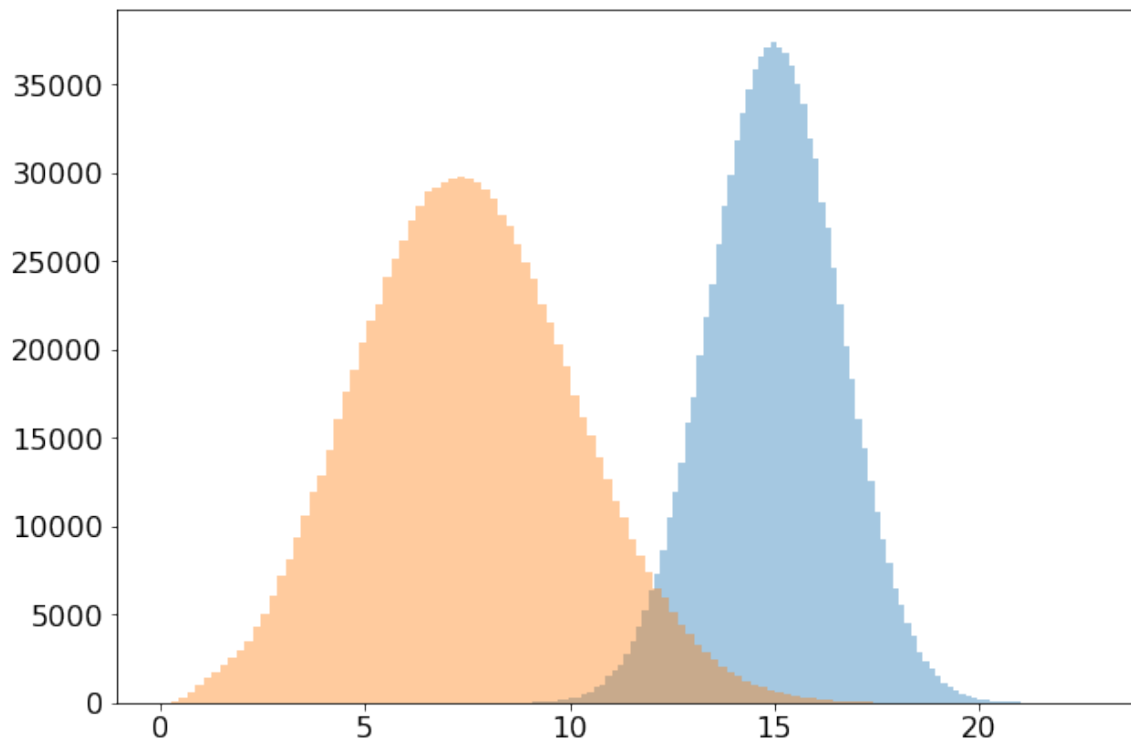


Figure 19: png

2.5.4 Pairing with a subset of r_i verifying $x_i > y_i$ only

```
# define index of selected points  
selection = x > y  
  
# add an empty dimension to make broadcasting possible  
selected_r = r*selection[:, :, None]  
  
# replace all 0 (False) by nan so that any combinaison with one of those  
# will be nan - and be filtered  
selected_r[selected_r == 0] = np.nan  
  
# Print the two first events  
print(selected_r[:2])
```

```
[[[ nan nan nan]
```

```
[0.42932845 0.32782561 0.27783779]
[0.56957279 0.0189856 0.25712615]
[      nan      nan      nan]
[      nan      nan      nan]
[      nan      nan      nan]
[      nan      nan      nan]
[      nan      nan      nan]
[      nan      nan      nan]
[0.34640387 0.29864917 0.30953939]]
```

```
[[      nan      nan      nan]
 [      nan      nan      nan]
 [      nan      nan      nan]
 [      nan      nan      nan]
[0.87616923 0.76191866 0.07106315]
[0.68232237 0.21571922 0.54843904]
 [      nan      nan      nan]
[0.7959044 0.33925827 0.74188441]
 [      nan      nan      nan]
[0.61883856 0.08677641 0.79843253]]]
```

```
pairs = combs_nd(selected_r, n=2, axis=1) # get all the possible pairs
print(pairs.shape)
```

```
(1000000, 45, 2, 3)
```

```
# get the first and second element of the pair
p1, p2 = pairs[:, :, 0, :], pairs[:, :, 1, :]

# compute the distance (summed over x,y,z, is axis=2)
dp = np.sum((p1-p2)**2, axis=2)**0.5

# set a default value of irrelevant pairs
dp[np.isnan(dp)] = 999
print(dp.shape)
```

```
(1000000, 45)
```

```
ax = plt.hist(dp.flatten(), bins=np.linspace(0, 2, 200), alpha=0.3,
    ↪ label='$|r_i-r_j|$ for all pairs')
ax = plt.hist(np.min(dp, axis=1), bins=np.linspace(0, 2, 200), alpha=0.3,
    ↪ label='min$(|r_i-r_j|)$')
ax = plt.legend()
```

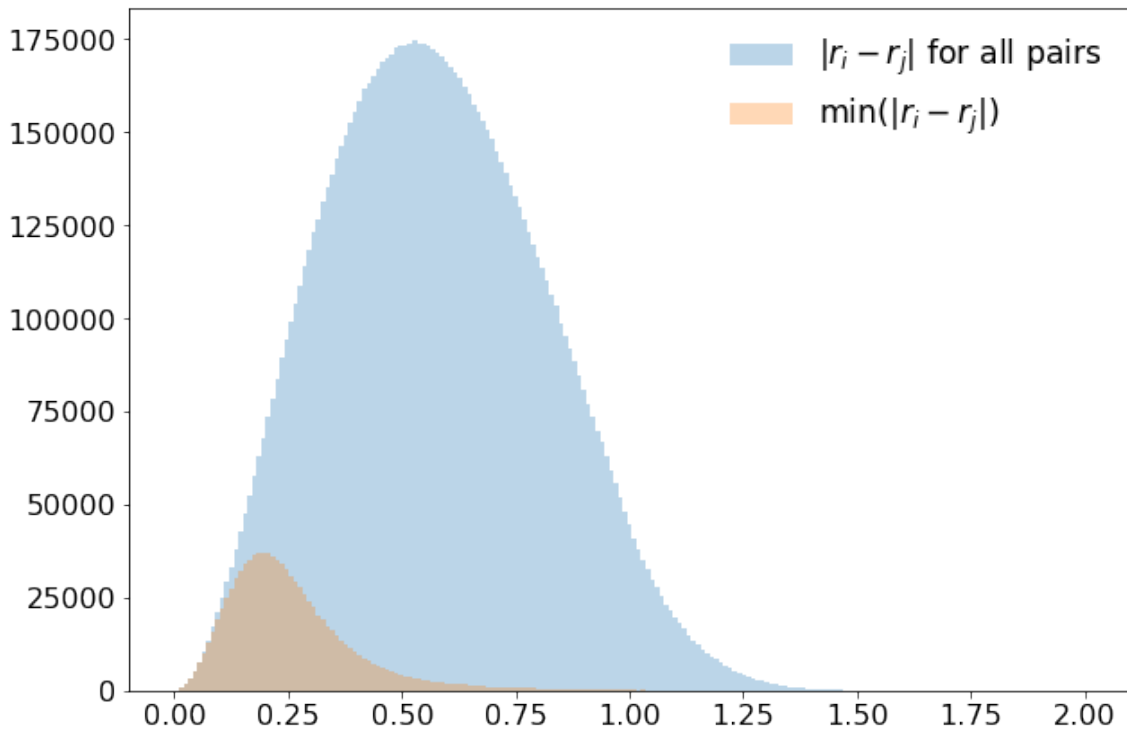



Figure 20: png

2.6 Play with two collections of vectors with different size $\{r_i\}_{10}$ and $\{q_i\}_6$

```
q = np.random.random_sample((1000000, 6, 3))
print(q[0:2])
```

```
[[[0.72846976 0.36453197 0.01145192]
  [0.35419366 0.12872047 0.88346129]
  [0.8832149  0.07994779 0.73534333]
  [0.69584834 0.53609698 0.52243828]
  [0.36197157 0.02261003 0.87612184]
  [0.34844683 0.83137148 0.77408195]]

 [[0.23096686 0.7873199  0.48843201]
  [0.47783539 0.79411894 0.60524871]
  [0.23530117 0.42049234 0.80970652]
  [0.67694867 0.48655929 0.62385231]
  [0.10215222 0.51858284 0.77035387]
  [0.81167546 0.91135836 0.90273964]]]
```

```
def all_pairs_nd(a, b, axis=0):

    # Sanity check
    good_shape = np.array_equal(
        np.delete(a.shape, axis), np.delete(b.shape, axis))
    if not good_shape:
        err = 'The shape along all dimensions but the one of axis={}'.format(
            axis)
        err += ' should be equal, while here:\n'
        err += '   -> shape of a is {} \n'.format(a.shape)
        err += '   -> shape of b is {} \n'.format(b.shape)
        raise NameError(err)

    # Individual indices
    a, b = np.asarray(a), np.asarray(b)
    ia, jb = np.arange(a.shape[axis]), np.arange(b.shape[axis])

    # Pairs of indices
    dt = np.dtype([('i', np.intp)]*2)
    if np.array_equal(a, b):
        ij = np.fromiter(itertools.combinations(ia, 2), dtype=dt)
    else:
        ij = np.fromiter(itertools.product(ia, jb), dtype=dt)
    ij = ij.view(np.intp).reshape(-1, 2)

    # Array of all pairs
    ipair, jpair = ij[:, 0], ij[:, 1]
    return np.stack([a.take(ipair, axis=axis), b.take(jpair, axis=axis)],
        ↪ axis=axis+1)
```

```
p = all_pairs_nd(r, q, axis=1)
print(p.shape)
```

```
(1000000, 60, 2, 3)
```

```
pairs = all_pairs_nd(r, r, axis=1)
print(pairs.shape)
```

```
(1000000, 45, 2, 3)
```

```
# Case where it will crash
p = all_pairs_nd(r, q, axis=2)
```

```
-----

NameError                                Traceback (most recent call last)

<ipython-input-29-b06c074af0d9> in <module>()
      1 # Case where it will crash
--> 2 p = all_pairs_nd(r, q, axis=2)

<ipython-input-26-3ff98aab2b79> in all_pairs_nd(a, b, axis)
     10         err += ' -> shape of a is {} \n'.format(a.shape)
     11         err += ' -> shape of b is {} \n'.format(b.shape)
--> 12         raise NameError(err)
     13
     14     # Individual indices
```

NameError: The shape along all dimensions but the one of axis=2 should be equal, while here:

```
-> shape of a is (1000000, 10, 3)
-> shape of b is (1000000, 6, 3)
```

2.7 Supplement: explanation of all_pairs_nd(a,b,axis)

```
axis = 1

a = np.array([
    [[1, 2], [3, 4]],
    [[5, 6], [7, 8]]
])

b = np.array([
    [[9, 10], [11, 12], [13, 14]],
    [[15, 16], [17, 18], [19, 20]]
])
```

a

```
array([[[1, 2],
        [3, 4]],

       [[5, 6],
        [7, 8]]])
```

b

```
array([[[ 9, 10],
        [11, 12],
        [13, 14]],

       [[15, 16],
        [17, 18],
        [19, 20]]])
```

```
# Get the indices of all pairs
ia, jb = np.arange(a.shape[axis]), np.arange(b.shape[axis])
dt = np.dtype([(' ', np.intp)]*2)
ij = np.fromiter(itertools.product(ia, jb), dtype=dt)
print(ij)
ij = ij.view(np.intp)
print(ij)
ij = ij.reshape(-1, 2)
print(ij)
```

```
[(0, 0) (0, 1) (0, 2) (1, 0) (1, 1) (1, 2)]
[0 0 0 1 0 2 1 0 1 1 1 2]
[[0 0]
 [0 1]
 [0 2]
 [1 0]
 [1 1]
 [1 2]]
```

```
ipair = ij[:, 0] # first element of ij is the index of a
pairs_a = np.take(a, ipair, axis=1)
pairs_a
```

```
array([[[1, 2],
        [1, 2],
        [1, 2],
        [3, 4],
        [3, 4],
        [3, 4]],

       [[5, 6],
        [5, 6],
        [5, 6],
        [7, 8],
        [7, 8],
        [7, 8]]])
```

```
jpair = ij[:, 1] # second element of ij is the index of b
pairs_b = np.take(b, jpair, axis=1)
pairs_b
```

```
array([[[ 9, 10],
        [11, 12],
        [13, 14],
        [ 9, 10],
        [11, 12],
        [13, 14]],

       [[15, 16],
        [17, 18],
        [19, 20],
        [15, 16],
        [17, 18],
        [19, 20]]])
```

```
pairs = np.stack([pairs_a, pairs_b], axis=2)
print(pairs.shape)
pairs
```

```
(2, 6, 2, 2)
```

```
array([[[[ 1,  2],
```

```
[ 9, 10]],  
  
[[ 1,  2],  
 [11, 12]],  
  
[[ 1,  2],  
 [13, 14]],  
  
[[ 3,  4],  
 [ 9, 10]],  
  
[[ 3,  4],  
 [11, 12]],  
  
[[ 3,  4],  
 [13, 14]]],  
  
[[[ 5,  6],  
   [15, 16]],  
  
 [[ 5,  6],  
   [17, 18]],  
  
 [[ 5,  6],  
   [19, 20]],  
  
 [[ 7,  8],  
   [15, 16]],  
  
 [[ 7,  8],  
   [17, 18]],  
  
 [[ 7,  8],  
   [19, 20]]]])
```

3 Analysis of typical collider data and numpy limitation

Caution: this notebook needs to have ROOT installed with python and root_numpy.

```
# Disable warnings  
import warnings  
warnings.filterwarnings('ignore')
```

```

# Usual library
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Time profiling
from timeit import default_timer
import cProfile

# Combinatorics tool
import itertools

# root_numpy (http://scikit-hep.org/root\_numpy)
from root_numpy import root2array

```

Welcome to JupyROOT 6.14/04

```

# Plot settings
import matplotlib as mpl
mpl.rcParams['legend.frameon'] = False
mpl.rcParams['legend.fontsize'] = 'xx-large'
mpl.rcParams['xtick.labelsize'] = 16
mpl.rcParams['ytick.labelsize'] = 16
mpl.rcParams['axes.titlesize'] = 18
mpl.rcParams['axes.labelsize'] = 18
mpl.rcParams['lines.linewidth'] = 2.5

```

3.1 Loading a ROOT TTree as a pandas DataFrame

Load a TTree and *view* it as a recarray (ie a *structured array* with fields accessible as attribute), then convert it into a pandas dataframe very easily. Our collider data are now in a pandas dataframe.

```

ar = root2array('collisions.root', 'event_tree').view(np.recarray)
df = pd.DataFrame(ar)
df.head()

```

```

<tr style="text-align: right;">
  <th></th>

```

```

<th>jet_pt</th>
<th>jet_eta</th>
<th>jet_phi</th>
<th>jet_mv2c10</th>
<th>jet_isbtagged_77</th>
<th>el_pt</th>
<th>el_eta</th>
<th>el_phi</th>
<th>mu_pt</th>
<th>mu_eta</th>
<th>mu_phi</th>
<th>mu</th>
</tr>

<tr>
  <th>0</th>
  <td>[169695.5, 122250.03]</td>
  <td>[-0.39637992, 1.6006567]</td>
  <td>[1.4989699, -1.4390093]</td>
  <td>[0.9628408, 0.99954563]</td>
  <td>[1, 1]</td>
  <td>[55366.094, 38978.633]</td>
  <td>[1.9268323, 0.13278118]</td>
  <td>[-3.0938299, -0.1095593]</td>
  <td>[]</td>
  <td>[]</td>
  <td>[]</td>
  <td>3.5</td>
</tr>

<tr>
  <th>1</th>
  <td>[92278.93, 70800.66, 69653.164, 27776.486]</td>
  <td>[0.40425044, 0.96515447, 0.5671447, 0.6175964]</td>
  <td>[-1.6717116, 1.0977219, -1.2310998, 0.40548608]</td>
  <td>[0.99883986, 0.999752, -0.897119, -0.8301639]</td>
  <td>[1, 1, 0, 0]</td>
  <td>[]</td>
  <td>[]</td>
  <td>[]</td>
  <td>[65079.883, 37495.855]</td>
  <td>[1.3351973, 0.3574057]</td>
  <td>[2.9533806, 3.1300983]</td>
  <td>26.5</td>
</tr>

<tr>

```



```

<th>2</th>
<td>[56349.285, 43751.82, 36588.938, 35095.082, 27...</td>
<td>[0.56248516, 2.4351118, -1.7529668, 1.2876523,...</td>
<td>[-0.38178313, 1.3481613, -2.3859453, -2.722344...</td>
<td>[0.9999337, -0.7246226, -0.7773614, -0.9201909...</td>
<td>[1, 0, 0, 0, 0]</td>
<td>[76494.64]</td>
<td>[-0.31438547]</td>
<td>[-1.9961401]</td>
<td>[49808.887]</td>
<td>[-0.17943819]</td>
<td>[-3.1100628]</td>
<td>27.5</td>
</tr>
<tr>
<th>3</th>
<td>[59820.547, 41592.062]</td>
<td>[-2.302116, -2.218402]</td>
<td>[1.9511205, 0.5742027]</td>
<td>[-0.82315505, 0.9919272]</td>
<td>[0, 1]</td>
<td>[39917.418]</td>
<td>[-1.3356979]</td>
<td>[2.6993954]</td>
<td>[103460.734]</td>
<td>[-2.07254]</td>
<td>[-1.3708612]</td>
<td>9.5</td>
</tr>
<tr>
<th>4</th>
<td>[196711.52, 123898.07, 87307.625, 82197.49, 41...</td>
<td>[-2.2168732, -0.5487006, -1.6435306, -1.037495...</td>
<td>[-0.24934195, 0.61117375, 1.870176, -2.149553,...</td>
<td>[-0.97217584, -0.9577969, -0.88246375, 0.99995...</td>
<td>[0, 0, 0, 1, 0, 0]</td>
<td>[197385.73]</td>
<td>[-0.749733]</td>
<td>[-2.845913]</td>
<td>[34190.586]</td>
<td>[-1.418337]</td>
<td>[-1.6127276]</td>
<td>12.5</td>
</tr>

```

```
print('Number of events: {:.0f}'.format(len(df)))
```

```
Number of events: 250000
```

3.2 Variable-size arrays and “squared” arrays

Pandas is very nice and powerful for flat numbers (*i.e.* no arrays), while in collider physics we have various collections of physics objects (of various size) for each events. This means two things: 1. it’s very common to have arrays per event and not only numbers 2. the size of the array will change from an event to another (those are called *jagged arrays*).

Doing pure python is not a problem with jagged arrays but it’s impossible to benefit from numpy vectorization since this requires well defined shape. In practice, the numpy array obtained by `df.values` is a 1D-array of arrays, and not a n-dimensional array:

```
array_jet_pt = df['jet_pt'].values
print('shape: {}'.format(array_jet_pt.shape))
```

```
shape: (250000,)
```

```
# Comprehensive loop for Njets
%timeit Njets=[len(j) for j in array_jet_pt]
```

```
10 loops, best of 3: 53.7 ms per loop
```

3.2.1 Squaring arrays

In order to work around this issue, one can “square jagged arrays” by setting the variable size to the maximum number of objects among all events, and fill empty values with a dummy value (to be carefully chosen depending on your computation). This is exactly what the function `square_jagged_2Darray(a, val=value, nobj=Nmax)` does, as illustrated below. The cell below prints the jet p_T array for the three first event, for different formatting of the array. The construction of this function is detailed (and timed) after.

```
# Utils function to manipulate jagged arrays
import np_utils as npu
```

```
# Main function
help(npu.square_jagged_2Darray)
```

Help on function square_jagged_2Darray in module np_utils:

```
square_jagged_2Darray(a, **kwargs)
```

Give the same dimension to all rows of a jagged 2D array.

This function equalizes the the size of every row (obj collection) using a default value 'val' (nan if nothing specifed) using either the maximum size of object collection among all column (events) or using a maximum size 'size'. The goal of this function is to fully use numpy vectorization which works only on fixed size arrays.

Parameters

a: array of arrays with different sizes this is the jagged 2D array to be squared

keyword arguments

dtype: string

data type of the variable-size array. If not specified, it is 'float32'. None means dt=data.dt.

nobj: int

max size of the array.shape[1]. if not specified (or None), this size is the maximum size of all rows.

val: float32

default value used to fill empty elements in order to get the proper size. If not specified (or None), val is np.nan.

Returns

out: np.ndarray

with a dimension (ncol,nobj).

Examples

```
>> import numpy as np
>> a=np.array([
    [1,2,3,4,5],
    [6,7],
    [8],
    [9,10,11,12,13]
```

```

])
>>
>> square_jagged_2Darray(a)
array([[ 1.,  2.,  3.,  4.,  5.],
       [ 6.,  7., nan, nan, nan],
       [ 8., nan, nan, nan, nan],
       [ 9., 10., 11., 12., 13.]], dtype=float32)
>>
>> square_jagged_2Darray(a,nobj=2,val=-999)
>> array([[ 1.,  2.],
       [ 6.,  7.],
       [ 8., -999.],
       [ 9., 10.]], dtype=float32)

```

```

# Raw numbers (ie before squaring)
print('\n\nBefore squaring:')
print('=====')
jet_pt_df = df['jet_pt'].values
print('shape: {}'.format(jet_pt_df.shape))
for pt in jet_pt_df[0:3]:
    print(len(pt), pt)

# After squaring
print('\n\nAfter squaring:')
print('=====')
jet_pt_np = npu.square_jagged_2Darray(jet_pt_df, val=-999)
print('shape: {}'.format(jet_pt_np.shape))
for pt in jet_pt_np[0:3]:
    print(len(pt), pt)

# After squaring with Nmax=3
print('\n\nAfter squaring with Nmax=3:')
print('=====')
jet_pt_np = npu.square_jagged_2Darray(jet_pt_df, val=-999, nobj=3)
print('shape: {}'.format(jet_pt_np.shape))
for pt in jet_pt_np[0:3]:
    print(len(pt), pt)

```

Before squaring:

=====

```

shape: (250000,)
(2, array([169695.5 , 122250.03], dtype=float32))
(4, array([92278.93 , 70800.66 , 69653.164, 27776.486], dtype=float32))
(5, array([56349.285, 43751.82 , 36588.938, 35095.082, 27441.059],

```

```
dtype=float32))
```

After squaring:

```
=====
```

```
shape: (250000, 11)
```

```
(11, array([169695.5 , 122250.03,  -999.   ,  -999.   ,  -999.   ,  -999.   ,
          -999.   ,  -999.   ,  -999.   ,  -999.   ,  -999.   ],
          dtype=float32))
```

```
(11, array([92278.93 , 70800.66 , 69653.164, 27776.486,  -999.   ,  -999.   ,
          -999.   ,  -999.   ,  -999.   ,  -999.   ,  -999.   ],
          dtype=float32))
```

```
(11, array([56349.285, 43751.82 , 36588.938, 35095.082, 27441.059,  -999.   ,
          -999.   ,  -999.   ,  -999.   ,  -999.   ,  -999.   ],
          dtype=float32))
```

After squaring with Nmax=3:

```
=====
```

```
shape: (250000, 3)
```

```
(3, array([169695.5 , 122250.03,  -999.   ], dtype=float32))
```

```
(3, array([92278.93 , 70800.66 , 69653.164], dtype=float32))
```

```
(3, array([56349.285, 43751.82 , 36588.938], dtype=float32))
```

3.2.2 Typical timing

```
# Getting the array directly
%timeit df['jet_pt'].values

# Squaring the array
%timeit npu.square_jagged_2Darray(jet_pt_df, val=-999)

# # Squaring the array with max 3 objects
%timeit npu.square_jagged_2Darray(jet_pt_df, val=-999, nobj=3)
```

The slowest run took 15.08 times longer than the fastest. This could mean that an intermediate result is being cached.

100000 loops, best of 3: 2.72 μ s per loop

1 loop, best of 3: 211 ms per loop

1 loop, best of 3: 206 ms per loop

3.2.3 Detail of square_jagged_2Darray() function

As it was probably noted, the square_jagged_2Darray() is longer than directly taking the numpy array. This is mostly due to two steps: scanning to find the max of object numbers, and the concatenation of all individual arrays. At the end, loading the squared numpy array takes 0.2 seconds for 250 kEvents. The timing and the details of operation is shown below:

```
# 0. Getting the 1D array of arrays
jet_pt_df = df['jet_pt'].values

# 1. Getting all the sub-array length
%timeit lens = np.array([len(i) for i in jet_pt_df])
lens = np.array([len(i) for i in jet_pt_df])
print('lens:\n {}'.format(lens[:3]))

# 2. Create a mask to know which value should be filled
%timeit mask = np.arange(lens.max()) < lens[:, None]
mask = np.arange(lens.max()) < lens[:, None]
print('mask:\n {}'.format(mask[:3]))

# 3. Initialize the final squared array
%timeit out = np.zeros(mask.shape, dtype='float32')
out = np.zeros(mask.shape, dtype='float32')
print('out:\n {}'.format(out[:3]))

# 4. Fill the default values where needed
%timeit out.fill(999)
out.fill(999)
print('out:\n {}'.format(out[0:3]))

# 5. Fill the 1D array (out[mask]) of all jet pT with the values using
↳ concatenate
%timeit out[mask] = np.concatenate(jet_pt_df)
jet_pt_1d = np.concatenate(jet_pt_df)
out[mask] = jet_pt_1d
print(('out:\n {}'.format(out[0:3])))
```

10 loops, best of 3: 62.8 ms per loop

lens:

[2 4 5]

100 loops, best of 3: 4.13 ms per loop

mask:

```
[[ True  True False False False False False False False False]
 [ True  True  True  True False False False False False False]]
```

```
[ True  True  True  True  True False False False False False False]]
1000 loops, best of 3: 640 µs per loop
out:
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
1000 loops, best of 3: 750 µs per loop
out:
[[999. 999. 999. 999. 999. 999. 999. 999. 999. 999. 999.]
 [999. 999. 999. 999. 999. 999. 999. 999. 999. 999. 999.]
 [999. 999. 999. 999. 999. 999. 999. 999. 999. 999. 999.]]
10 loops, best of 3: 107 ms per loop
out:
[[169695.5  122250.03    999.      999.      999.      999.
   999.      999.      999.      999.      999. ]
 [ 92278.93  70800.66  69653.164  27776.486   999.      999.
   999.      999.      999.      999.      999. ]
 [ 56349.285  43751.82  36588.938  35095.082  27441.059   999.
   999.      999.      999.      999.      999. ]]
```

Another function called `df2array()` allows to load several columns (with the same maximum size) into a given nd array. This is needed if one wants to make computations based on all those columns. The best example is the dR variable which involves both η and ϕ . These two variables can be grouped in a big numpy array of dimension $(N_{\text{evts}}, N_{\text{jets}}, 2)$, where 2 corresponds to the number of variables. This function internally call the `np.stack()` method (on top of some checks):

```
jets_kin = npu.df2array(df, ['jet_pt', 'jet_eta', 'jet_phi'])
```

is equivalent to

```
jets_pt  = npu.square_jagged_2Darray(df['jet_pt'].values)
jets_eta = npu.square_jagged_2Darray(df['jet_eta'].values)
jets_phi = npu.square_jagged_2Darray(df['jet_phi'].values)
jets_kin = np.concatenate([jets_pt, jets_eta, jets_phi], axis=2)
```

```
jets_kin = npu.df2array(df[0:1000], ['jet_pt', 'jet_eta', 'jet_phi'])
print(jets_kin.shape)

jets_btg = npu.df2array(df[0:1000], ['jet_mv2c10', 'jet_isbtaged_77'])
print(jets_btg.shape)
```

```
jets = npu.df2array(df[0:1000], ['jet_pt', 'jet_eta',  
                                'jet_phi', 'jet_mv2c10',  
                                ↪ 'jet_isbtagged_77'])  
print(jets.shape)
```

```
(1000, 8, 3)
```

```
(1000, 8, 2)
```

```
(1000, 8, 5)
```

```
# Load jet array for all events  
jets = npu.df2array(df, ['jet_pt', 'jet_eta', 'jet_phi',  
                        'jet_mv2c10', 'jet_isbtagged_77'])
```

3.3 Producing some non-trivial plots using numpy arrays

Everything which is based on flat number can be directly done pandas columns directly, *e.g.* the following code will be similarly efficient as with a `TTree->Draw()` command.

```
plt.figure(figsize=(10,7))  
ax=plt.hist(df['mu'])
```

But the more tricky part is what to do with python to make some more complex computations **without doing an explicit event loop**? The next sub-sections give some examples.

```
plt.figure(figsize=(10, 7))  
ax = plt.hist(df['mu'])
```

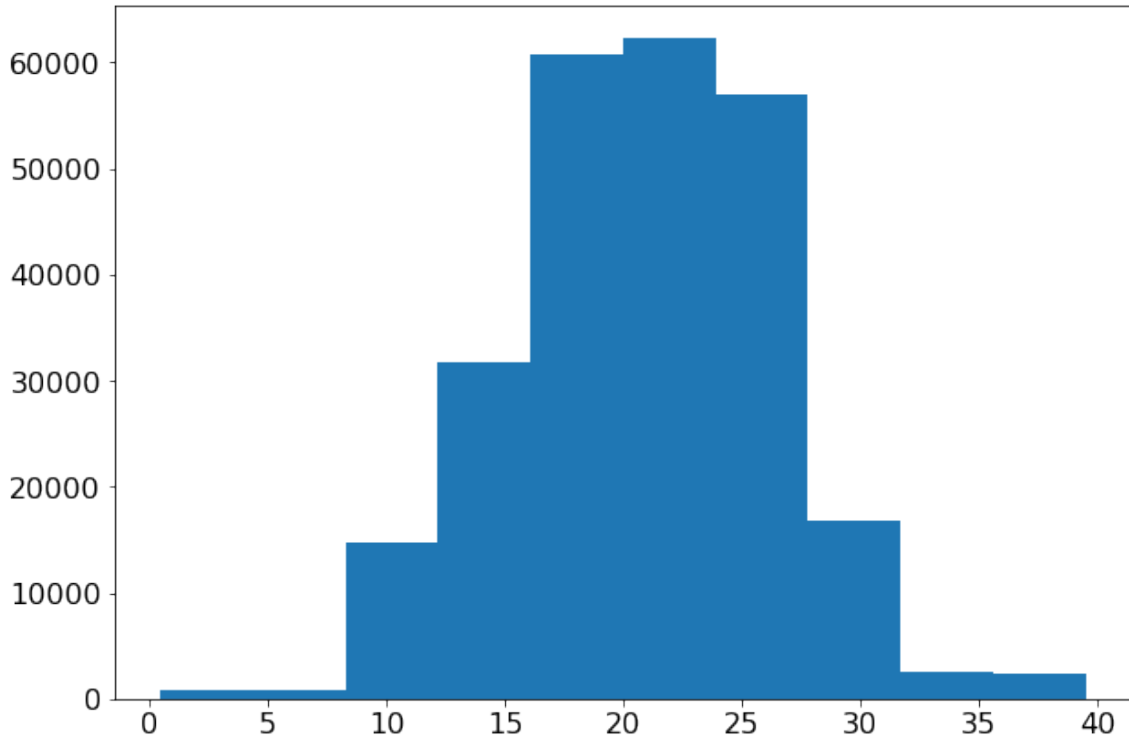



Figure 21: png

3.3.1 Jet multiplicity for different p_T thresholds

Looking at the jet multiplicity depending on the p_T threshold: `+ jets[...:0]` means that all dimension but the last one is inclusive (here it means all events and all jets for each event), while the 0 means first variable (i.e. the p_T since it comes first in the command `df2array(df, ['jet_pt', 'jet_eta', 'jet_phi', 'jet_mv2c10', 'jet_isbtagged_77'])`); `+ jets[...:0]>pt` is a 2D array of shape (Nevts,Njets) with True and False depending on whether the element is above p_T or not; `+ np.count_nonzero(jets[...:0]>pt, axis=1)` is a 1D array of shape (Nevts) which counts the number of True along the Njets axis (so per event).

```
plt.figure(figsize=(10, 7))
for pt in np.linspace(25, 100, 4)*1000:
    ax = plt.hist(np.count_nonzero(jets[...:0] > pt, axis=1),
                  label='$p_T>{:0f}$ GeV'.format(pt/1000.),
                  alpha=0.8, histtype='step', linewidth=3,
                  bins=np.linspace(0, 15, 15), log=True)
ax = plt.legend()
ax = plt.xlabel('$N_{jets}(p_T>X)$')
ax = plt.ylabel('Event count')
```

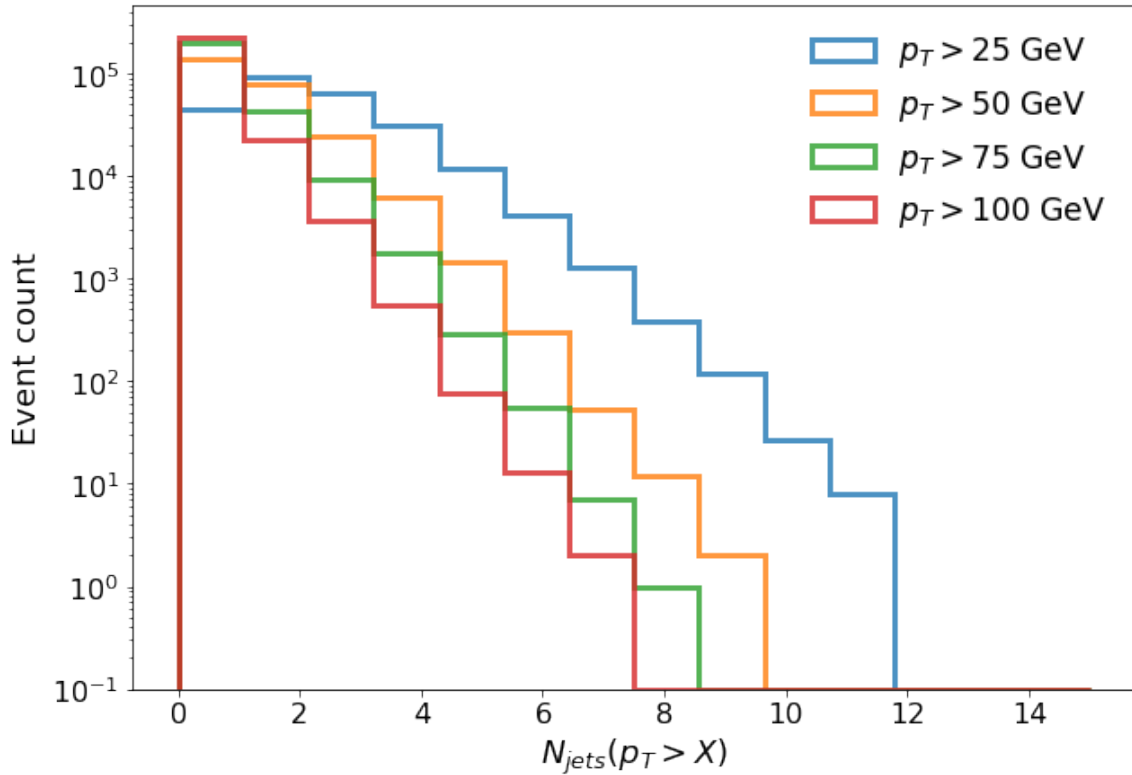


Figure 22: png

3.3.2 Jet p_T distribution for every jets in the event

This is also very easy to look at the p_T distributions of the leading, sub-leading, ... jets. For this, one first needs to replace all nan (not a number) by a appropriate default value (0 for instance), otherwise the plotting step will crash (cannot plot nan). Then a loop over all the jets is performed (the number of jets is the size of the dimension 2, *i.e.* `shape[1]`).

```
jets_pt_plots = npu.replace_nan(jets[..., 0], value=0)

fig = plt.figure(figsize=(10, 7))
Njets = jets_pt_plots.shape[1]
for i in np.arange(Njets):
    ax = plt.hist(jets_pt_plots[:, i]/1000., alpha=0.3, linewidth=3,
                  bins=np.linspace(25, 2000, 100), log=True, label='jet
    ↪ {}'.format(i+1))
ax = plt.legend()
ax = plt.xlabel('$p^{jets}_T$ [GeV]')
ax = plt.ylabel('Unweighted events')
```

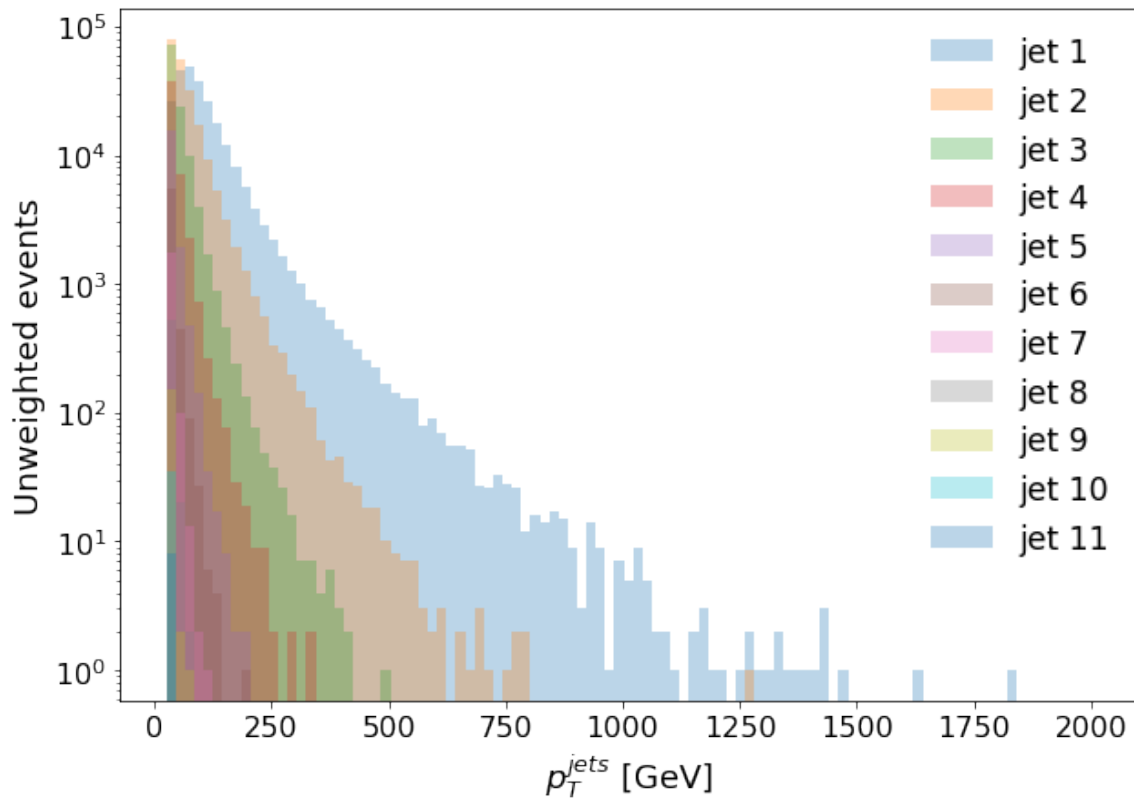


Figure 23: png

3.3.3 *Apparte*: difference between `a*(a>x)` and `a[a>x]`

First of all `a>x` is an array filled with True or False depending on whether the condition is true or false (in numpy, it is called a *mask*). What do the two different commands do: + `a[a>x]` return all elements of `a` which pass the condition. In practice, it removes the other elements from the array. **This is always a 1D array.** + `a*(a>x)` return the product of `a` and `a>x` converted into a `int` (so 0 or 1). In practice, it replaces the values not passing the condition by `False` or 0. + if `a` is multi-dimensional, `a[a>x]` will be a flat (1D) array. This is unavoidable since the output would be a jagged array. Indeed, for a 2D array, the number of elements per line might depend on the line.

This is illustrated with examples below for both 1D and 2D arrays.

```
# 1D arrays
a = np.arange(12)
print('a          = {}'.format(a))
print('a>4        = {}'.format(a > 4))
print('a*(a>4)    = {}'.format(a*(a > 4)))
print('a[a>4]     = {}'.format(a[a > 4]))
```

```

a      = [ 0  1  2  3  4  5  6  7  8  9 10 11]
a>4    = [False False False False False  True  True  True  True  True  True]
a*(a>4) = [ 0  0  0  0  0  5  6  7  8  9 10 11]
a[a>4]  = [ 5  6  7  8  9 10 11]

```

```

# 2D arrays
a = np.arange(12).reshape(6, 2)
print('a      = {}'.format(a))
print('a>4    = {}'.format(a > 4))
print('a*(a>4) = {}'.format(a*(a > 4)))
print('a[a>4] = {}'.format(a[a > 4]))

```

```

a      = [[ 0  1]
 [ 2  3]
 [ 4  5]
 [ 6  7]
 [ 8  9]
 [10 11]]
a>4    = [[False False]
 [False False]
 [False  True]
 [ True  True]
 [ True  True]
 [ True  True]]
a*(a>4) = [[ 0  0]
 [ 0  0]
 [ 0  5]
 [ 6  7]
 [ 8  9]
 [10 11]]
a[a>4]  = [ 5  6  7  8  9 10 11]

```

3.3.4 H_T distribution in different configurations

One can also recompute observables using only objects passing certain selections (this is not so easy to do with TTree->Draw() commands). Let's take the example of H_T defined as the scalar sum of p_T over the jets (probing the “hardness” of the collision): + Usual case: jet_pt_ht is the p_T array with a shape (Nevt,Njets), so sum over axis=1 will give the H_T array with shape (Nevts). HTjets[HTjets>0] means removing events with $H_T = 0$ (if not jets at all for example); + Compute H_T only with central jets: jet_pt_ht*(np.abs(jet_eta)<1.0) is an array containing only p_T of jets with $|\eta| < 1.0$, then the logic remains the same; + Compute H_T

only with b-tagged jets: `jet_pt_ht*(jet_btagw>0.67)` is an array containing only p_T of jets with $w_b > 0.67$.

```
jet_pt_ht = npu.replace_nan(jets[..., 0], value=0)/1000.
jet_eta = jets[..., 1]
jet_btagw = jets[..., 3]
```

```
fig = plt.figure(figsize=(10, 7))

# Compute usual HT jets
HTjets = np.sum(jet_pt_ht, axis=1)
ax = plt.hist(HTjets[HTjets > 0], alpha=0.5, bins=np.linspace(
    0, 2000, 100), label='$|eta|<2.5$', log=True)

# Compute HT only with central jets
central_jet_pt_ht = jet_pt_ht*(np.abs(jet_eta) < 1.0)
HTjets_central = np.sum(central_jet_pt_ht, axis=1)
ax = plt.hist(HTjets_central[HTjets_central > 0], alpha=0.5,
    ↪ bins=np.linspace(
        0, 2000, 100), label='$|eta|<1$', log=True)

# Compute HT only with b-jets
bjets_pt_ht = jet_pt_ht*(jet_btagw > 0.67)
HTbjets = np.sum(bjets_pt_ht, axis=1)
ax = plt.hist(HTbjets[HTbjets > 0], alpha=0.5, bins=np.linspace(
    0, 2000, 100), label='b-jets', log=True)

ax = plt.title('All jets vs central jets $|\\eta|<1$')
ax = plt.xlabel('$H_T$ [GeV]')
ax = plt.ylabel('Unweighted events')
ax = plt.legend(title='$t\\bar{t}$')
```

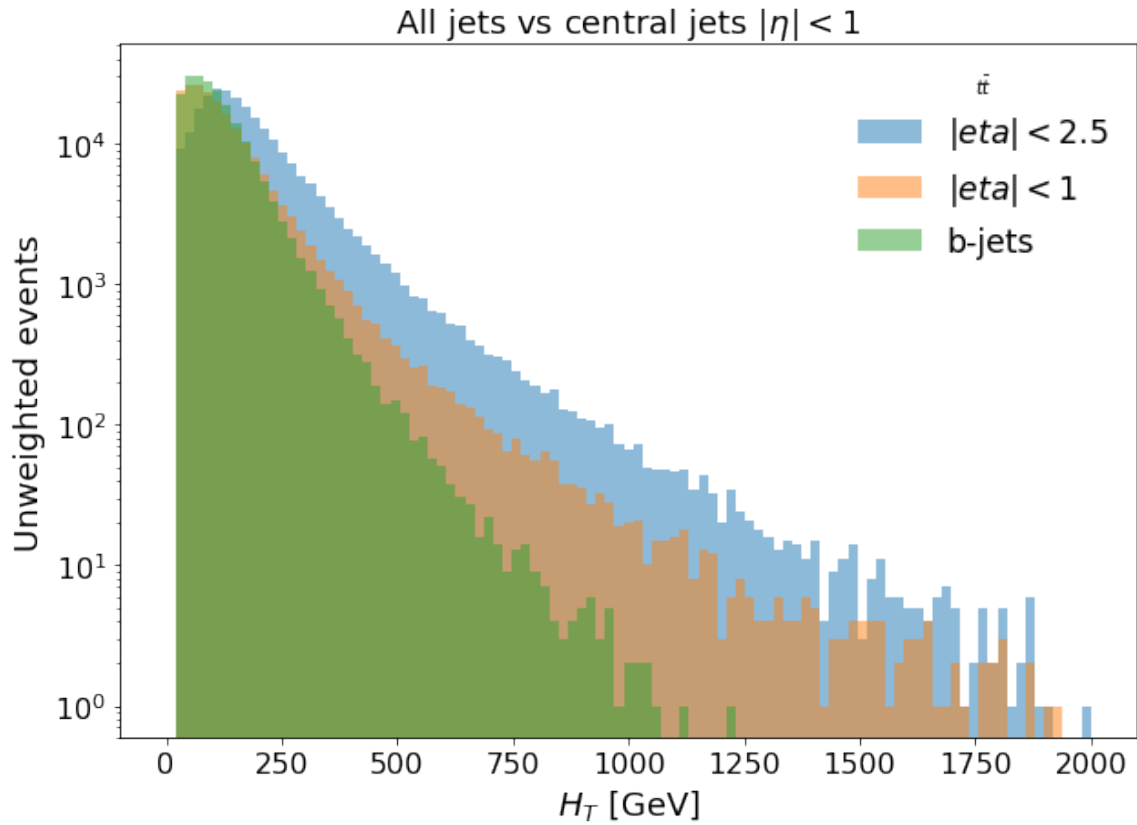


Figure 24: png

3.4 Perform event-by-event computations without explicit loop

There are many obvious use cases of doing these typical calculations: + identify the jet which is the closest of a given lepton (minimum ΔR computation) + compute invariant mass between all possible electrons and find the combination corresponding to a Z decay + find the jet pair which best match a hadronic W decay

In principle, the same methodology could be applied to combination having more than 2 objects (rough decay reconstruction). But this can be quite long to compute - depending on the number of events - because we have to deal with large number of objects (the max one, in order to get fixed-size array). One option though, is to limit the number of object participating to the combination, by taking for example the 5th first leading p_T jets. In our current example, this would reduce the number of jets from 13 to 5 (in term of $N(N-1)/2$ combinations: 78 to 10).

3.4.1 Getting all possible pairs of jets

```
jet_pairs = npu.all_pairs_nd(jets)
print(jet_pairs.shape)
```

```
(250000, 55, 2, 5)
```

3.4.2 How to select only events with at least two objects?

In the case of making pairs of the two same objects, one needs to make sure there are at least two! Let's take the example of jets:

1. we need to compute the number of jets, *i.e.* the number of not nan per event (since empty elements are set to nan), which can be done for any variable (here p_T): python
`nj=npu.count_nonnan(jets[...],0),axis=1)`
2. Select all jets and all variables for events with `nj>1`: python `jets_atl2 = jets[nj>1,...]`

```
nj=npu.count_nonnan(jets[...],0),axis=1)
print('There are {} events without any jets'.format(np.count_nonzero(nj==0)))
is_0j = nj==0
print(is_0j.shape)
jets_atl2 = jets[~is_0j]
print(jets_atl2.shape)
```

```
There are 4803 events without any jets
(250000,)
(245197, 11, 5)
```

3.4.3 Compute pair-related observables

Once the pairs are formed, we can do any computation with it. For convenience, you can make two variables being the first jet `j1` and the second jet `j2` of the pair. Those will be array of shape `(Nevt,Npair,Nvar)`:

```
j1, j2 = jet_pairs[:, :, 0, :], jet_pairs[:, :, 1, :]
print(j1.shape, j2.shape)
```

```
((250000, 55, 5), (250000, 55, 5))
```

3.4.3.1 Minimum $\Delta R(j, j)$

We can then take the sum, the difference, the invariant mass or anything else based on j_1 and j_2 . Below, we form the array of $(\Delta\eta, \Delta\phi)$ for each pair, having a shape (Nevt,Npair,2):

```
# keep only eta,phi to compute dR=sqrt(deta^2+dphi^2)
dj_etaphi = j1[..., 1:3] - j2[..., 1:3]

# remove nan by a relevant default values (outside plots)
dj_etaphi = npu.replace_nan(dj_etaphi, value=999)

# print the 5th first pair of the 3rd event
print(dj_etaphi.shape, dj_etaphi[2, 0:5])
```

```
((250000, 55, 2), array([[ -1.8726265e+00, -1.7299445e+00],
 [ 2.3154519e+00,  2.0041623e+00],
 [-7.2516710e-01,  2.3405614e+00],
 [-1.8223300e+00,  2.6417046e+00],
 [ 9.9900000e+02,  9.9900000e+02]], dtype=float32))
```

```
dR = np.sum(dj_etaphi**2, axis=2)**0.5
print(dR.shape, dR[0, 0:5])

dR = npu.replace_val(dR, (2**0.5)*999., 999)
print(dR.shape, dR[0, 0:5])
```

```
((250000, 55), array([ 3.5524466, 1412.7993, 1412.7993, 1412.7993,
 1412.7993], dtype=float32))
((250000, 55), array([ 3.5524466, 999., 999., 999.,
 999.],
 dtype=float32))
```

```
fig = plt.figure(figsize=(10, 7))
ax = plt.hist(dR.flatten(), bins=np.linspace(
    0, 8, 100), alpha=0.5, label='All jet pairs')
ax = plt.hist(np.min(dR, axis=1), bins=np.linspace(
    0, 8, 100), alpha=0.5, label='Minimal  $\Delta R(j, j)$ ')
ax = plt.xlabel('$\Delta R(j, j)$')
ax = plt.ylabel('Unweighted events')
ax = plt.legend()
```

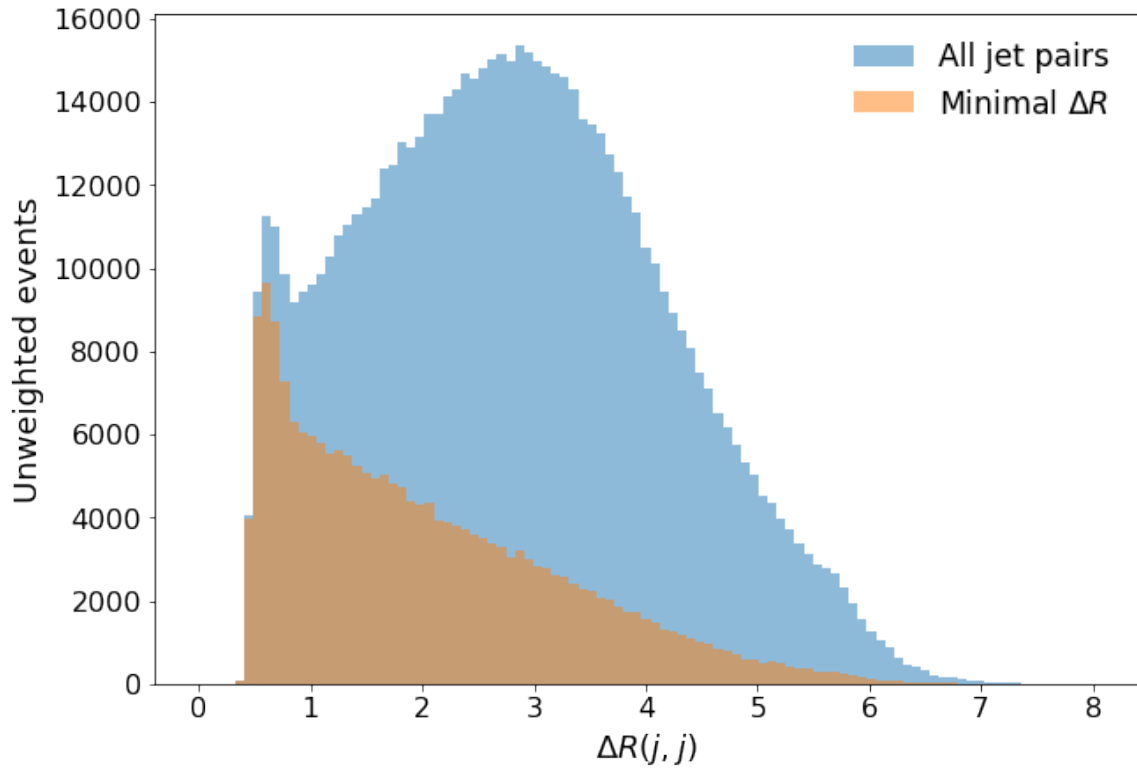



Figure 25: png

3.4.3.2 Minimum $\Delta R(j, e)$

```
jet_direction = jets[:, :, 1:3]
ele_direction = npu.df2array(df, ['el_eta', 'el_phi'])
```

```
jet_ele_pairs_direction = npu.all_pairs_nd(jet_direction, ele_direction)
```

```
dej = jet_ele_pairs_direction[:, :, 0, :] - jet_ele_pairs_direction[:, :, 1, :]
dRej = npu.replace_nan(np.sum(dej**2, axis=2)**0.5, value=999)
dRmin = np.min(dRej, axis=1)
```

```
fig = plt.figure(figsize=(10, 7))
style = {
    'bins': np.linspace(0, 8, 100),
    'alpha': 0.5,
    'density': True,
    'log': True,
}
```

```

ax = plt.hist(dRej.flatten(), label='All jet-electron pairs', **style)
ax = plt.hist(dRmin, label='Minimal $\Delta R(j,e)$', **style)
ax = plt.xlabel('$\Delta R(j,e)$')
ax = plt.ylabel('Unweighted events')
ax = plt.legend()

```

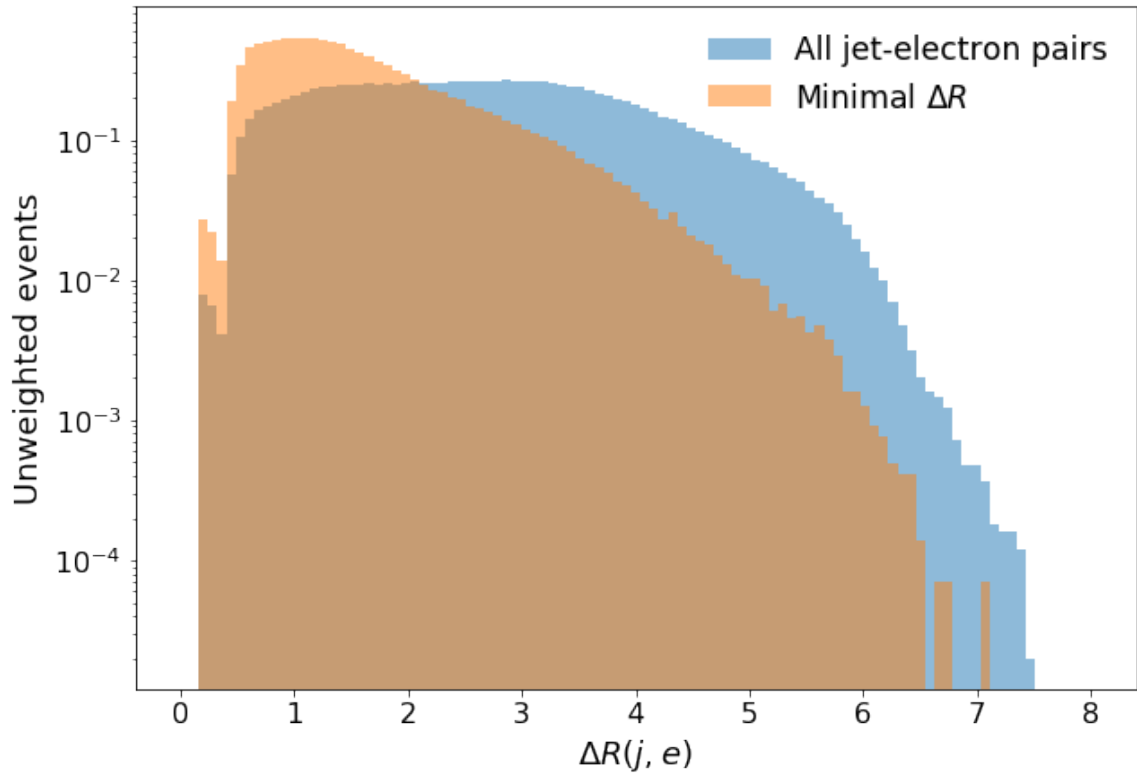


Figure 26: png

3.4.4 Di-jet invariant masses

Let's take the example of the invariant mass between j_1 and j_2 :

$$m^2 = p_{T1}^2 p_{T2}^2 (\cosh(\eta_1 - \eta_2) - \cos(\phi_1 - \phi_2))$$

```

deta, dphi = dj_etaphi[..., 0], dj_etaphi[..., 1]
pt1, pt2 = j1[..., 0], j2[..., 0]
print(pt1.shape, deta.shape)

```

```
((250000, 55), (250000, 55))
```

```
m = np.sqrt(pt1*pt2 * (np.cosh(deta)-np.cos(dphi))) / 1000.  
m = npu.replace_nan(m, 1e10)  
print(m.shape)
```

(250000, 55)

```
fig = plt.figure(figsize=(10, 7))  
  
style = {  
    'bins': np.linspace(0, 500, 100),  
    'alpha': 0.8,  
    'density': True,  
    'log': False,  
    'histtype': 'step',  
    'linewidth': 3.0  
}  
  
ax = plt.hist(m.flatten(), label='All pairs', **style)  
ax = plt.hist(np.min(m, axis=1), label='Min  $M(j,j)$ ', **style)  
ax = plt.hist(np.max(npu.replace_val(m, 1e10, -1e10), axis=1),  
              label='Max  $M(j,j)$ ', **style)  
ax = plt.xlabel(' $M(j,j)$  [GeV]')  
ax = plt.ylabel('Unweighted events')  
ax = plt.legend()
```

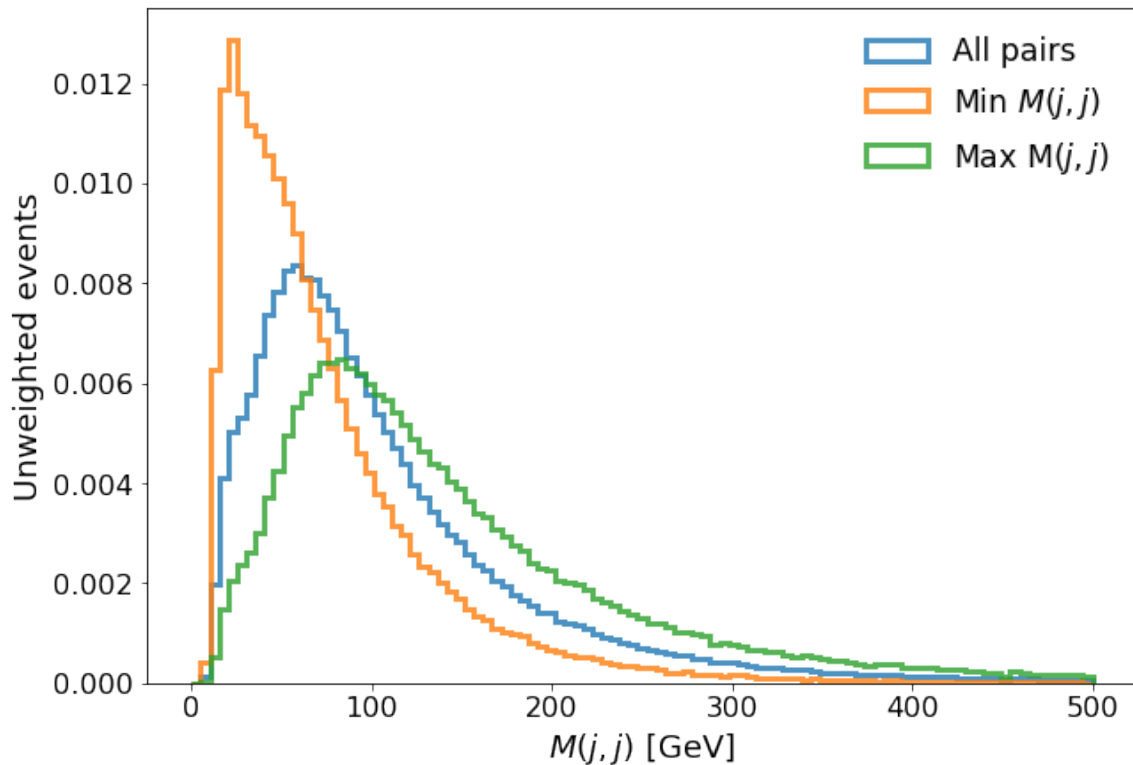


Figure 27: png

3.5 Build up a system with several collections of objects (e.g. electrons and jets)

In particle physics, we often want to group objects together and compute observables related to the global system. For example, grouping all leptons together can be useful to reconstruct W transverse mass regardless of the lepton flavour of W -decay products. Another example could be to group together objects with a similar signature in the detector (e.g. deposit into the electromagnetic calorimeter). Typically: $\text{lep}=\{\text{el}+\mu\}$ or $\text{EMobj}=\{\text{jets}+\text{ele}\}$.

3.5.1 Preamble: implementing default values like `df2array(df, ['var1', 'var2', '999'])`

This would be useful to work around the constraint of having the same number of variables per object. For example, if one wants to make all possible pairs of electrons and jets or simply group the collection together, we need to have the same dimension along the variable axis (i.e. `axis=3`). Of course, variables for jets might not exist for electrons (or the opposite). Concretely, the following code

```
jets      =
↳ df2array(df, ['jet_pt', 'jet_eta', 'jet_phi', 'jet_mv2c10', 'jet_isbtagged77'])
electrons = df2array(df, ['el_pt', 'el_eta', 'el_phi'])
```

```
ele_jets = all_pairs_nd(jets,electrons)
```

will not work and will return something like

NameError: The shape along all dimensions but the one of axis=1 should be equal, while here:

-> shape of a is (1000, 8, 5)

-> shape of b is (1000, 3, 3)

The adopted possibility is to be able to set a default value just to have the proper number of variable for both object **and** remember that this is a dummy value, like

```
jets =
↳ df2array(df,['jet_pt','jet_eta','jet_phi','jet_mv2c10','jet_isbtagged_77'])
electrons = df2array(df,['el_pt','el_eta','el_phi', '-999',
↳ 'nan'])
ele_jets = all_pairs_nd(jets,electrons)
```

Since jets currently contains 5 variables, one needs to build up a collection of electrons with 5 variables. But the btagg weight is not defined for electron, so we put a dummy value (otherwise the stacking cannot work).

```
print(jets.shape)
```

```
(250000, 11, 5)
```

```
eles = npu.df2array(df, ['el_pt', 'el_eta', 'el_phi', 'nan', 'nan'])
```

```
jets_eles = npu.stack_collections([jets, eles])
print(jets.shape, eles.shape, jets_eles.shape)
```

```
((250000, 11, 5), (250000, 3, 5), (250000, 14, 5))
```

```
jet_el_pt = npu.replace_nan(jets_eles[:, :, 0])
jet_el_HT = np.sum(jet_el_pt/1000., axis=1)
print(jet_el_HT.shape)
```

```
(250000,)
```

```
fig = plt.figure(figsize=(10, 7))
plt.hist(jet_el_HT[jet_el_HT > 0], bins=np.linspace(0, 1000, 100), alpha=0.5)
ax = plt.xlabel('$\sum_{e,j} \; p_T$ [GeV]')
ax = plt.ylabel('Unweighted events')
```

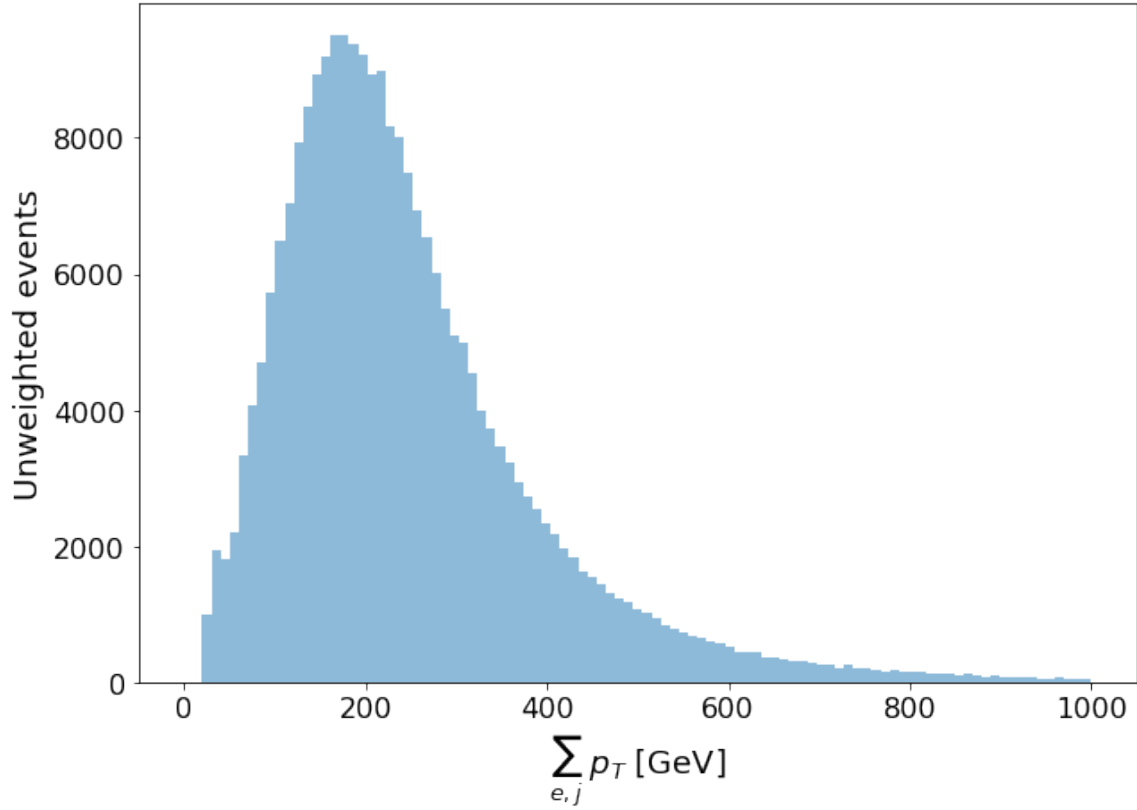


Figure 28: png

3.5.2 Select an object based on an event-level criteria (distance, invariant mass, etc ...)

The goal of this section is to look at say the isolation of the leptons which form a pair having $M(e, e) \sim M(Z)$.

3.5.2.1 E.g. 1: compare the b-tagging weight of the jet closest to an electron and the others

We reform all the pair here, but not only with the direction but all needed variables:

```
jets_elec_pairs = npu.all_pairs_nd(jets, eles)
print(jets.shape, eles.shape, jets_elec_pairs.shape)
```

```
((250000, 11, 5), (250000, 3, 5), (250000, 33, 2, 5))
```

Then we need to isolate an array of shape (Nevt,Npair) containing the btagg weight (3rd variable) of the first element (*i.e.* the jet) for any pair: `btagw=jets_ele[:, :, 0, 3]`

```
jet_btag_w = npu.replace_nan(jets_elec_pairs[:, :, 0, 3], value=999)
```

Reminder of $dR(e, j)$ and $\text{mind}R(e, j)$ distribution (already computed from before):

```
fig = plt.figure(figsize=(20, 7))
plt.subplot(121)
ax = plt.hist(dRmin, bins=np.linspace(0, 8, 100), alpha=0.5,
              density=True, log=True, label='Smallest $dR$')
ax = plt.hist(dRej.flatten(), bins=np.linspace(0, 8, 100),
              alpha=0.5, density=True, log=True, label='All pairs')
ax = plt.xlabel('$dR(j,e)$')
ax = plt.ylabel('PDF')
ax = plt.legend()

plt.subplot(122)
ax = plt.hist(dRmin, bins=np.linspace(0, 0.5, 50), alpha=0.5, density=True)
ax = plt.xlabel('min $dR(j,e)$')
ax = plt.ylabel('PDF')
```

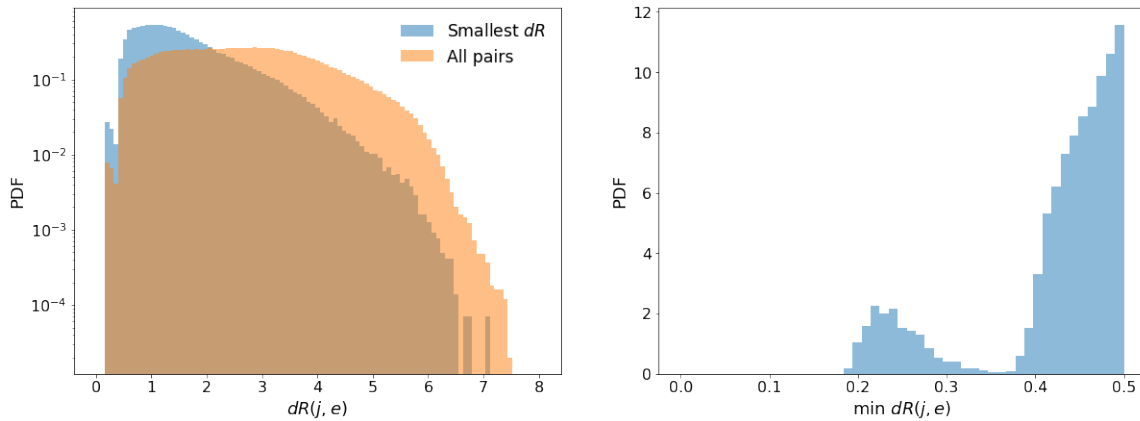


Figure 29: png

Getting now the index of the pair having the minimal dR using the command `np.argmax(dRej,axis=1)` which return a 1D array of shape (Nevt) containing the wanted index for each event. Then one can use the functions `get_indexed_value()` and `get_all_but_indexed_value()` to get either the btag weight of the minimal dR or all the others.

```
idRmin = np.argmin(dRej, axis=1)
jet_btag_w_dRmin = npu.get_indexed_value(jet_btag_w, idRmin)
jet_btag_w_other = npu.get_all_but_indexed_value(jet_btag_w, idRmin)
```

```
fig = plt.figure(figsize=(20, 7))
plt.subplot(121)
ax = plt.hist(jet_btag_w_dRmin, bins=np.linspace(-1, 1, 50),
              alpha=0.5, density=True, log=True, label='Closest jet')
ax = plt.hist(jet_btag_w_other.flatten(), bins=np.linspace(-1, 1, 50),
              alpha=0.5, density=True, log=True, label='Not the closest
→ jets')
ax = plt.xlabel('$w_{b-tagging}$ of the jet')
ax = plt.ylabel('PDF')
ax = plt.legend()
```

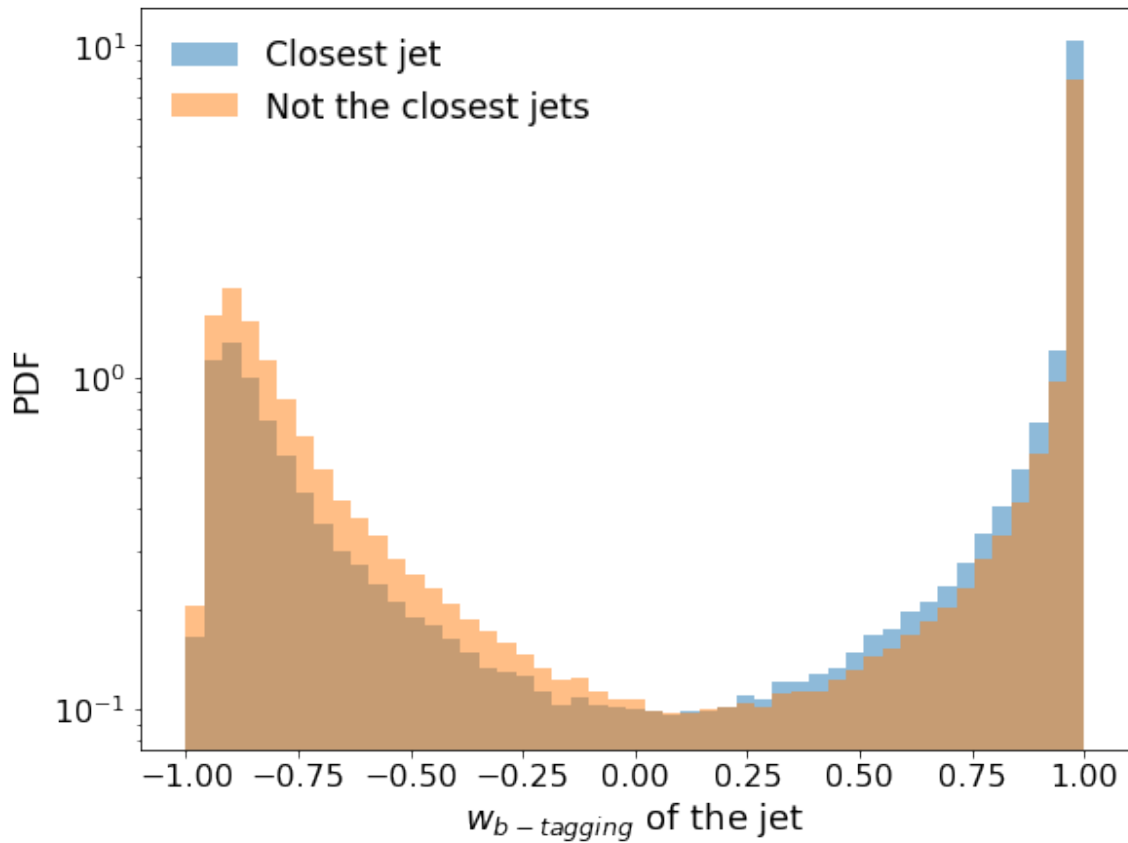


Figure 30: png


```

fig = plt.figure(figsize=(17, 10))
style = {
    'bins': np.linspace(-1, 1, 50),
    'alpha': 0.8,
    'density': True,
    'log': True,
    'histtype': 'step',
    'linewidth': 3.0
}
for i, cut in enumerate([0.35, 0.5, 1.0, 3.0]):
    plt.subplot(2, 2, i+1)
    dRgt_btag = jet_btag_w_dRmin*(dRmin>cut)
    dRgt_btag[dRgt_btag == 0] = 999
    dRlt_btag = jet_btag_w_dRmin*(dRmin<cut)
    dRlt_btag[dRlt_btag == 0] = 999
    ax = plt.hist(dRgt_btag, label='$dR_{min}>'+ '{:.2f}$'.format(cut),
        ↪ **style)
    ax = plt.hist(dRlt_btag, label='$dR_{min}<'+ '{:.2f}$'.format(cut),
        ↪ **style)
    ax = plt.xlabel('$w_{b-tagging}$ of the closest jet')
    ax = plt.ylabel('PDF')
    ax = plt.legend()
plt.tight_layout()

```

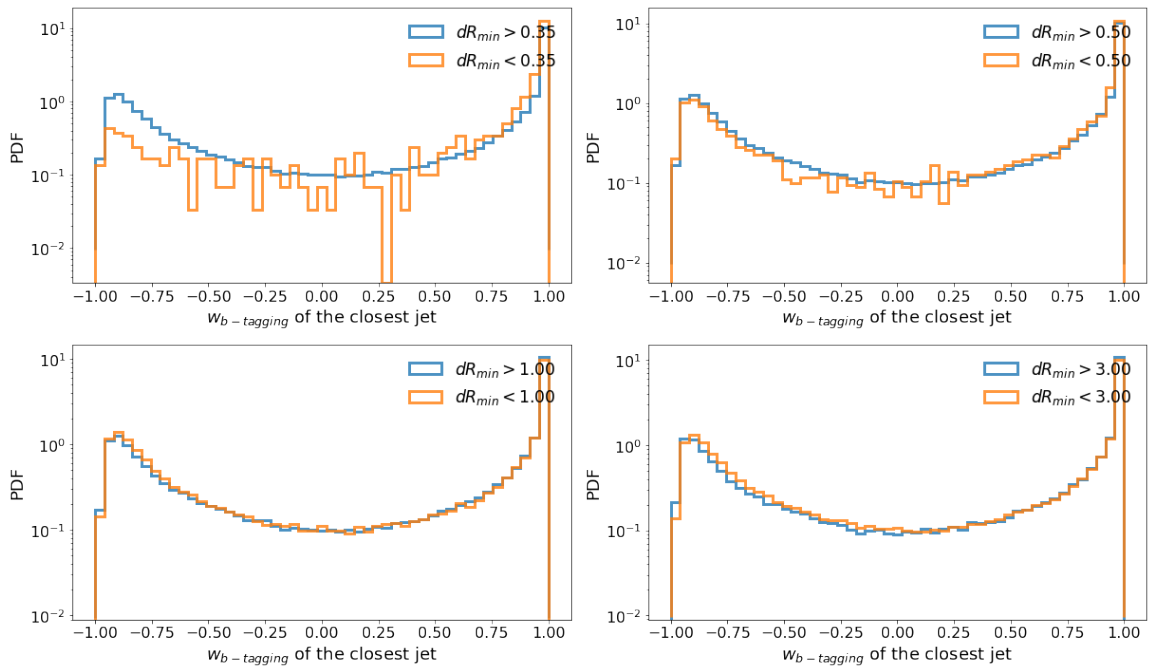


Figure 31: png

3.5.2.2 E.g. 2: p_T distribution for jets forming the highest $M(j, j)$

```
j1, j2 = jet_pairs[:, :, 0, :], jet_pairs[:, :, 1, :]
deta, dphi = j1[:, 1]-j2[:, 1], j1[:, 2]-j2[:, 2]
pt1, pt2 = j1[:, 0], j2[:, 0]
mjj = npu.replace_nan(
    np.sqrt(pt1*pt2 * (np.cosh(deta)-np.cos(dphi)))/1000., -1e10)
```

```
i_mjj_max = np.argmax(mjj, axis=1)
pt_mjj_max = np.concatenate([
    npu.get_indexed_value(pt1, i_mjj_max)/1000,
    npu.get_indexed_value(pt2, i_mjj_max)/1000
])
```

```
fig = plt.figure(figsize=(10, 7))
style = {
    'bins': np.linspace(0, 2000, 100),
    'alpha': 0.8,
    'log': True,
}
ax = plt.hist(npu.replace_nan(pt_mjj_max, -999), **style)
ax = plt.xlabel('Jet $p_T$ [GeV] for the two jets having max $M(j,j)$')
ax = plt.ylabel('Unweighted events')
```

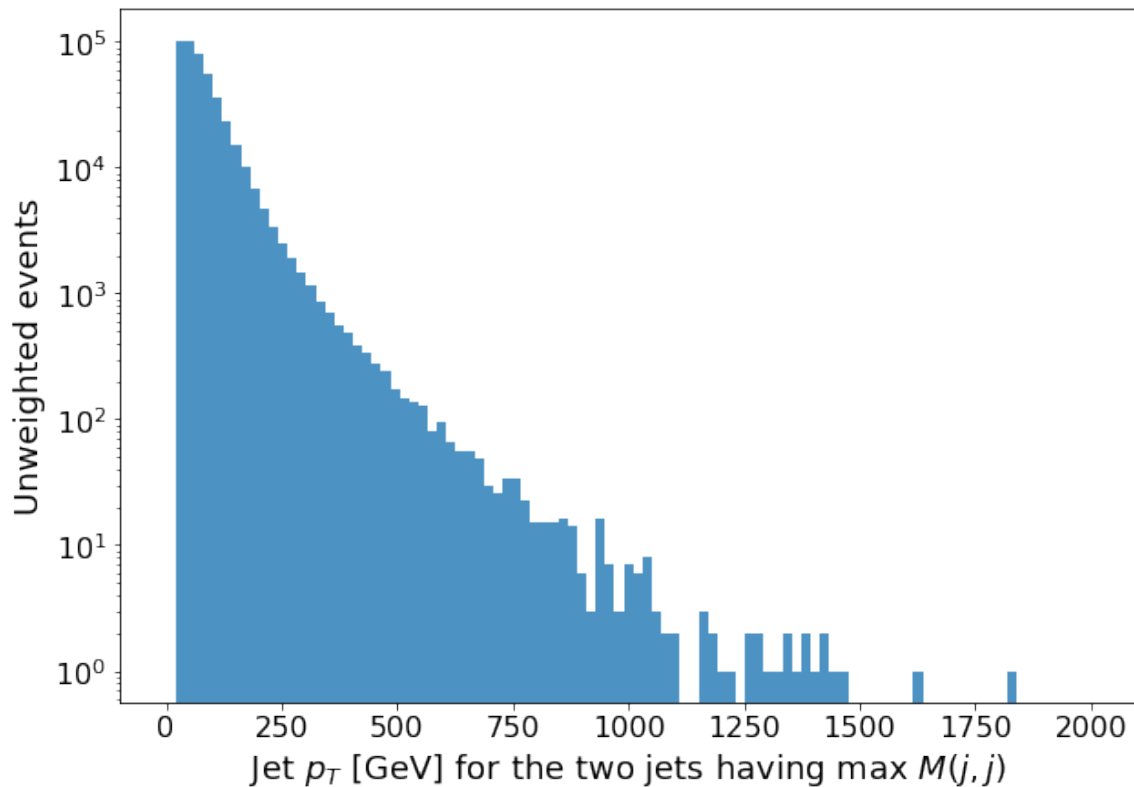


Figure 32: png

3.6 IO between pandas/numpy and ROOT

3.6.1 Save some variables back into a ROOT file

This is possible that one wants to save and share the obtained variables in a ROOT file. This is possible to do using a *structured array*, numpy object accepting both field different shape for each “column”. This goes through a data type (which in for instance the event model) and then each column can be filled. This is in principle a simplified pandas dataframe (simplified but interfaced with TTree via `root_numpy`). **Note that every `np.nan` will manifest in `tree->Draw()` as 0.0**

```
from root_numpy import array2root

# Define the event model with ('name', 'type', 'shape') for each column
event_model = np.dtype([
    ('n_jets', 'i4'),
    ('jet_pt', 'f8', (jets.shape[1],)),
    ('jet_eta', 'f8', (jets.shape[1],)),
    ('jet_phi', 'f8', (jets.shape[1],)),
```

```

    ('ht', 'f8'),
    ('ht_cent', 'f8'),
])

# Create the giant structured array
events = np.zeros(jets.shape[0], dtype=event_model)
events['n_jets'] = npu.count_nonnan(jets[..., 0], axis=1)
events['jet_pt'] = jets[..., 0]
events['jet_eta'] = jets[..., 1]
events['jet_phi'] = jets[..., 2]
events['ht'] = HTjets
events['ht_cent'] = HTjets_central

# Convert it into a TTree stored in a ROOT file
array2root(events, 'ttbar_jets.root', 'tree_jets', mode='recreate')

```

3.6.2 Read back the created file with root_numpy WIP

```

ar_new = root2array('ttbar_jets.root', 'tree_jets',
                    branches=['jet_pt']).view(np.recarray)

# The following line crashes with
# --> Exception: Data must be 1-dimensional
# df_new = pd.DataFrame(ar_new, index=np.arange(len(ar_new)))

```

3.7 Other existing tools

Many python tools for HEP can be found on <http://scikit-hep.org>. In particular, we can see various python tools to deal with ROOT files, as well as a special tool under development to manage jagged array as numpy array without the “squaring” I used. This tool is called [awkward-array](#).