Numpy for High Energy Physics

Romain Madar[1]

[1]Laboratoire de Physique de Clermont-Ferrand

January 4, 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.

Contents

1 Short introduction to numpy

Why numpy? Numpy stands for *numerical python* and is highly optimized (and then fast) for computations in python. Numpy is one of the core package on which many others are based on, such as scipy (for *scientific python*), matplotlib or pandas (described at the end of this chapter). A lot of other scientific tools are also based on numpy and that justifies to have - at least - a basic understanding of how it works. Very well, but one could also ask why using python?

Why python? Depending on your test and what you want to do, python can be a very good choice or not (of course this is largely a matter of taste and not everyone agrees with this statement). In any case, many tools are available in python, scanning a very broad spectrum of applications, from machine learning to web design or string processing.

1.1 The core object: arrays

The core of numpy is the called numpy array. These objects allow to efficiently perform computations over large dataset in a very consise way from the language point of view, and very fast from the processing time point of view. The price to pay is to give up explicit *for* loops. This lead to somehow a counter intuitive logic - at first.

1.1.1 Main differences with usual python lists

The first point is to differenciate numpy array from python list, since they don't behave in the same way. Let's define two python lists and the two equivalent numpy arrays.

```
import numpy as np
l1, l2 = [1, 2, 3], [3, 4, 5]
a1, a2 = np.array([1, 2, 3]), np.array([3, 4, 5])
print(l1, l2)
```

```
[1, 2, 3] [3, 4, 5]
```

First of all, all mathematical operations act element by element in a numpy array. For python list, the addition acts as a concatenation of the lists, and a multiplication by a scalar acts as a replication of the lists:

```
# obj1+obj2
print('python lists: {}'.format(l1+l2))
print('numpy arrays: {}'.format(a1+a2))
```

```
python lists: [1, 2, 3, 3, 4, 5]
numpy arrays: [4 6 8]
```

```
# obj*3
print('python list: {}'.format(l1*3))
print('numpy array: {}'.format(a1*3))
```

```
python list: [1, 2, 3, 1, 2, 3, 1, 2, 3]
numpy array: [3 6 9]
```

One other important difference is about the way to access element of an array, the so called slicing and indexing. Here the behaviour of python list and numpy arrays are closer expect that numpy array supports few more features, such as indexing by an array of integer (which doesn't work for python lists). Use cases of such indexing will be heavily illustrated in the next chapters.

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

```
python list: 2
numpy array: 2
```

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

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

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

```
python list: IMPOSSIBLE
numpy array: [1 3]
```

1.1.2 Main caracteristics of an array

The strenght of numpy array is to be multidimensional. This enables a description of a whole complex dataset into a single numpy array, on which one can do operations. In numpy, dimension are also called *axis*. For example, a set of 2 position in space $\vec{r}_i$ can be seen as 2D numpy array, with the first axis being the point $i = 1$ or $i = 2$, and the second axis being the coordinates (x, y, z). There are few attributes which describe multidimentional arrays:

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

```
points = np.array([[ 0,  1, 2],
                   [ 3,  4, 5]])

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

```
a.dtype = int64
a.shape = (2, 3)
a.size  = 6
a.ndim  = 2
```

1.2 The three key features of numpy

1.2.1 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(array):
    res = []
    for a in array:
        res.append(1./a)
    return np.array(res)
```

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

```
182 ms ± 23.3 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

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

```
14.5 ms ± 385 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

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

```
111 µs ± 4.56 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

The suppression of explicit *for* loops is probably the most unfamiliar aspect of numpy - according to me - and deserves a bit a of practice. At the end, lines of codes becomes relatively short but ones need to properly think how to implement a given computation in a *pythonic way*.

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         : [58 69 94]
a^2       : [3364 4761 8836]
a/(1-a^a): [8.74904014e-18 8.40800286e-18 9.40000000e+01]
cos(a)    : [0.11918014 0.99339038 0.96945937]
exp(a)    : [1.54553894e+25 9.25378173e+29 6.66317622e+40]
```

All these ufunct can work for n-dimension arrays and can be used in a very flexible way depeding on the axis you are refering 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('mean = {}'.format(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 = {}'.format(distances))
```

```
mean = [47.8 57.2 33.4]
distances = [112.94689017  67.44627492  93.52539762 115.57681428
57.62811814]
```

1.2.2 Broadcasting

The *broadcasting* is a way to compute operation between arrays of having different sizes in a implicit (and consice) manner. One concrete example could be to translate three positions $\vec{r}_i = (x, y)_i$ by a vector $\vec{d}_0$ simply by adding `points+d0` where `points.shape=(3,2)` and `d0.shape=(2,)`. 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])
print('a+b = \n{}'.format(a+b))
```

```
a+b =
[6 7 8]
```

```
# operation between shape (3) and (1,2)
a = np.array([1, 2, 3])
b = np.array([
              [4],
              [5],
             ])
print('a+b = \n{}'.format(a+b))
```

```
a+b =
[[5 6 7]
 [6 7 8]]
```

```
# Translating 3 2D vectors by d0=(1,4)
points = np.random.normal(size=(3, 2))
d0 = np.array([1, 4])
print('points:\n {}\n'.format(points))
print('points+d0:\n {}'.format(points+d0))
```

```
points:
 [[ 1.38799588  1.14437047]
 [-0.00850104  0.32030856]
 [-0.10623281  2.69526833]]

points+d0:
 [[2.38799588 5.14437047]
 [0.99149896 4.32030856]
 [0.89376719 6.69526833]]
```

Not all shapes can be combined together and there are *broadcasting rules*, which are (quoting the numpy documentation):

> When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward. Two dimensions are compatible when
>
> 1. they are equal, or
> 2. one of them is 1

In a failing case, one can then add a new *empty axis* `np.newaxis` to an array to make their dimension equal and then the broadcasting possible. Here is a very simle example:

```
a = np.arange(10).reshape(2,5)
b = np.array([10,20])
```

```
try:
    res = a+b
    print('Possible for {} and {}:'.format(a.shape, b.shape))
    print('a+b = \n {}'.format(res))
except ValueError:
    print('Impossible for {} and {}'.format(a.shape, b.shape))
```

```
Impossible for (2, 5) and (2,)
```

```python
c = b[:, np.newaxis]
try:
    res = a+c
    print('Possible for {} and {}:'.format(a.shape, c.shape))
    print('a+c = \n {}'.format(res))
except ValueError:
    print('Broadcasting for {} and {}'.format(a.shape, c.shape))
```

```
Possible for (2, 5) and (2, 1):
a+c =
 [[10 11 12 13 14]
 [25 26 27 28 29]]
```

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

As mentioned eariler, *slicing and indexing* are ways to access elements or sub-arrays in a smart way. Python allows slicing with `Slice()` object but numpy allows to push the logic much further with what is called *fancy indexing*. Few examples are given below and for more details, please have a look to this documentation page.

Rule 1: the synthax is `a[i]` to access the ith element. It is also possible to go from the last element using negative indices: `a[-1]` is the last element.

```python
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 = [57 18 63 55 22 94 48 81 76 51]
a[2] = 63
a[-1] = 51
a[[1, 2, 5]] = [18 63 94]
```

Rule 2: numpy also support array of indices. If the index array is multi-dimensional, the returned array will have the same dimension as the indices array.

```
# Small n-dimensional indices array: 3 arrays of 2 elements
indices = np.arange(6).reshape(3,2)
print('indices =\n {}'.format(indices))
print('a[indices] =\n {}'.format(a[indices]))
```

```
indices =
 [[0 1]
 [2 3]
 [4 5]]
a[indices] =
 [[57 18]
 [63 55]
 [22 94]]
```

```
# Playing with n-dimensional indices array: 2 arrays of (10, 10) arrays
indices_big = np.random.randint(low=0, high=10, size=(2, 3, 2))
print('indices_big =\n {}'.format(indices_big))
print('a[indices_big] =\n {}'.format(a[indices_big]))
```

```
indices_big =
 [[[6 9]
  [8 4]
  [8 9]]

 [[4 9]
  [5 8]
  [0 2]]]
a[indices_big] =
 [[[48 51]
  [76 22]
  [76 51]]

 [[22 51]
  [94 76]
  [57 63]]]
```

Rule 3: There is a smart way to access sub-arrays with the synthax `a[min:max:step]`. In that way, it's for example very easy to take one element over two (`step=2`), or reverse the order of an 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 shape (5, 2, 3) - that can considered as 5 observations of 2 positions in space.

```
# 1D array
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              = [13 40 81 65 11 85 20 86 42 21]
from 0 to 1: a[:2]        = [13 40]
from 4 to end: a[4:]      = [11 85 20 86 42 21]
reverse order: a[::-1]    = [21 42 86 20 85 11 65 81 40 13]
all even elements: a[::2] = [13 81 11 20 42]
```

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

```
a =
[[[60 38 98]
  [ 3 81 26]]

 [[82 58 80]
  [98 44 63]]

 [[39 48 51]
  [90  4 63]]

 [[73 44 96]
  [10 51 44]]

 [[52 58 58]
  [75 35 58]]]
```

Let's say, one wants to take only the (x, y) coordinates for the first vector for all 5 observations. This is how each axis will be sliced: - first axis (=5 observations): `:`, *i.e.* takes all - second axis (=2 vectors): 1 *i.e.* only the 2nd element - third axis (=3 coordinates): `0:2` *i.e.* from 0 to $2-1=1$, so only (x, y)

```
# Taking only the x,y values of the first vector for all observation:
print('a[:, 0, 0:2] =\n {}'.format(a[:, 0, 0:2]))
```

```
a[:, 0, 0:2] =
 [[60 38]
 [82 58]
 [39 48]
 [73 44]
 [52 58]]
```

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

```
a[:, ::-1, :] =
[[[ 3 81 26]
  [60 38 98]]

 [[98 44 63]
  [82 58 80]]

 [[90  4 63]
  [39 48 51]]

 [[10 51 44]
  [73 44 96]]

 [[75 35 58]
  [52 58 58]]]
```

Rule 4: The last part of of indexing is about *masking* array or in a more common language, *selecting* sub-arrays/elements. This allows to get only elements satisfying a given criteria, exploiting the indexing rules described above. Indeed, a boolean operation applied to an array such as a>0 will directly return an array of boolean values True or False depending if the corresponding element satisfies the condition or not.

```
a = np.random.randint(low=-100, high=100, size=(5, 3))
mask = a>0
print('a = \n{}'.format(a))
print('\nmask = \n {}'.format(mask))
```

```
a =
[[ 59  83 -83]
 [ 42  16 -62]
 [-36  63 -72]
 [-21 -63  11]
 [ 19 -31  46]]

mask =
 [[ True  True False]
 [ True  True False]
 [False  True False]
 [False False  True]
 [ True False  True]]
```

```
print('\na[mask] = \n {}'.format(a[mask])) # always return 1D array
print('\na*mask = \n {}'.format(a*mask)) # preserves the dimension (False=0)
print('\na[~mask] = \n {}'.format(a[~mask])) # ~mask is the negation of mask
print('\na*~mask  = \n {}'.format(a*~mask)) # working for a product too.
```

```
a[mask] =
 [59 83 42 16 63 11 19 46]

a*mask =
 [[59 83  0]
 [42 16  0]
 [ 0 63  0]
 [ 0  0 11]
 [19  0 46]]

a[~mask] =
 [-83 -62 -36 -72 -21 -63 -31]

a*~mask  =
 [[  0   0 -83]
 [  0   0 -62]
 [-36   0 -72]
 [-21 -63   0]
 [  0 -31   0]]
```

Note the case of boolean arrays as indices has then a special treatment in numpy (since the result is always a 1D array). There is actually a dedicated numpy object called *masked array* (cf. documentation) which allows to keep the whole array but without considering some elements in the computation (*e.g.* CCD camera with dead pixel). Note however that when a boolean array is used in an mathematical operation (such as `a*mask`) then `False` is treated as 0 and `True` as 1:

```
print('a+mask = \n{}'.format(a+mask))
```

```
a+mask =
[[ 60  84 -83]
 [ 43  17 -62]
 [-36  64 -72]
 [-21 -63  12]
 [ 20 -31  47]]
```

This boolean arrays are also very useful to *replace a category of elements* with a given value in a very easy, consise and readable way:

```
a = np.random.randint(low=-100, high=100, size=(5, 3))
print('Before: a=\n{}'.format(a))

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

```
Before: a=
[[-71 -58  70]
 [ 52 -62 -94]
 [ 15  85 -53]
 [-47 -19 -51]
 [-47  -5 -67]]

After: a=
[[5041 3364   70]
 [  52 3844 8836]
 [  15   85 2809]
 [2209  361 2601]
 [2209   25 4489]]
```

1.3 Two powerfull tools: matplotlib and pandas

1.3.1 Plotting with matplotlib

matplolib is an extremely rich librairy for data visualization and there is no way to cover all its features in this note. The goal of this section is just to give short and practical examples to plot data. Much more details can be obtained on the webpage. The following shows how to quickly make *histograms*, *graph*, *2D and 3D scatter plots*.

The main object of matplotlib is `matplotlib.pyplot` imported as `plt` here (and usually). The most common functions are then called on this objects, and often takes numpy arrays in argument (possibly with more than one dimension) and a lot of `kwargs` to define the plotting style.

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

1.3.1.1 Example of 1D plots and histograms

To play with data, we generate 2 samples of 1000 values distributed according to a normal probability density function with $\mu = -1$ and $\mu = 1$ respectively, and $\sigma = 0.5$. These data are stored in a numpy array `x` of shape `x.shape=(1000, 2)`. We then simply compute and store the sinus of all these values into a same shape array `y`:

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

The next step is to plot these data in two ways: first we want `y` v.s. `x`, second we want the histogram of the `x` values. We need to first create a figure, then create two *subplots* (specifying the number of line, column, and subplot index). Note that matplotlib take always the first dimension to define the numbers to plot, while higher dimensions are considered as other plots - automatically overlaid.

```
plt.figure(figsize=(24, 10))
plt.subplot(121) # 121 means 1 line, 2 column, 1st plot
ax = plt.plot(x, y, marker='o', markersize=5, linewidth=0.0)
plt.subplot(122) # 122 means 1 line, 2 column, 2nd plot
ax = plt.hist(x, bins=20)
```

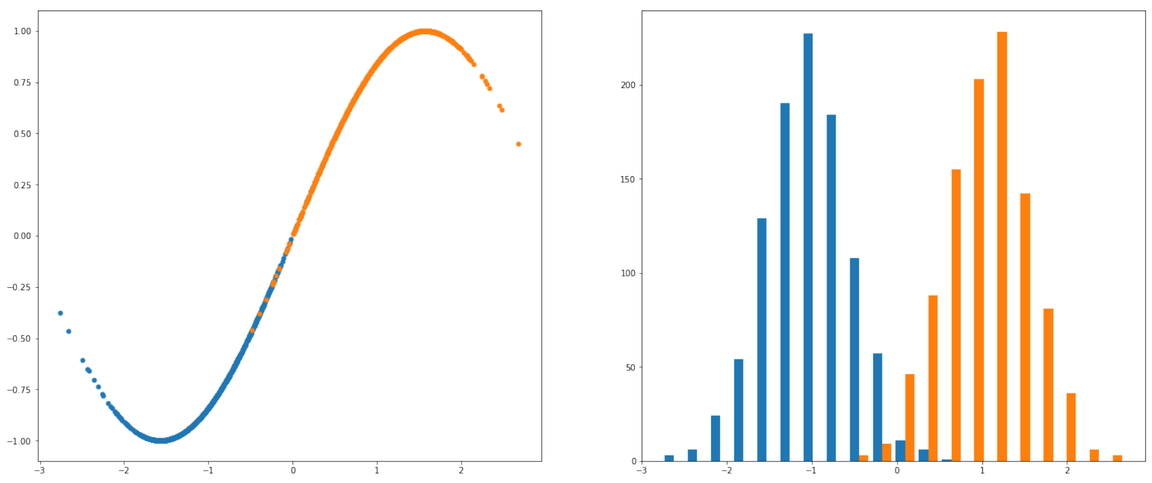

Figure 1

1.3.1.2 Example of 2D scatter plot

A scatter plot allows to draw marker in a 2D space and a thrid information is encoded into the marker size. In order to play, we generated two set of 5000 numbers distributed according to uncorrelated gaussians of $(\mu_0 = \mu_1 = 0)$ and $(\sigma_1, \sigma_2) = (0.5, 0.8)$ in a numpy array `points` of shape `points.shape=(5000,2)`:

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

These two set of numbers are interepreted as (x, y) position:

```
x, y = points[:, 0], points[:, 1]
```

We can then plot the 5000 points in the 2D plan, and here we specify the marker size at $100 \times \sin^2(x)$ using the argument `s` of `the plt.scatter()` function (note that the array `x`, `y` and `s` must have the same shape):

```
fig = plt.figure(figsize=(10,6))
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)
```

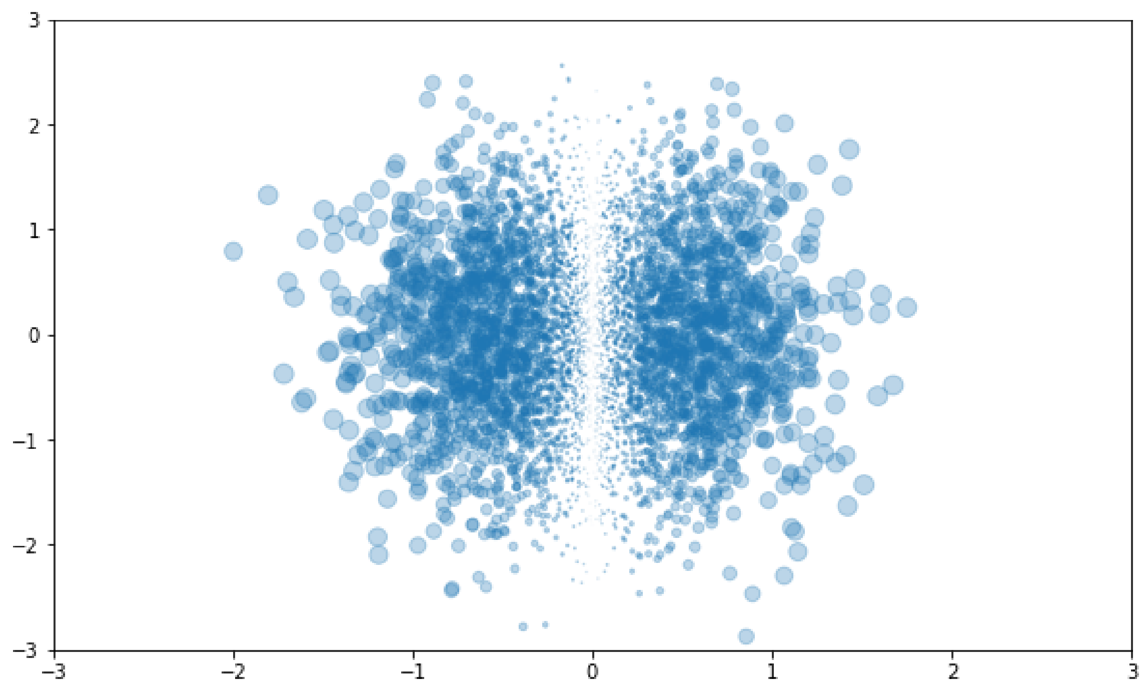

Figure 2

1.3.1.3 Example of 3D plots

For 3D plots, one can generate 1000 positions in space, and operate a translation by a vector $\vec{r}_0$ using broadcasting:

```
data = np.random.normal(size=(1000, 3))
r0 = np.array([1, 4, 2])
data_trans = data + r0
```

It is then easy to get back the spatial initial (*i.e.* before translation) and final (*i.e.* after translation) coordinates:

```
xi, yi, zi = data[:,0], data[:,1], data[:,2]
xf, yf, zf = data_trans[:,0], data_trans[:,1], data_trans[:,2]
```

An additional module must be imported in order to plot data in three dimensions, and the projection has to be stated. Once it's done, a simple call to `ax.scatter3D(x,y,z)` does the plot. Note that we call a function of `ax` and not `plt` as before. This is due to the `ax = plt.axes(projection='3d')` command which is needed for 3D plotting. More details are available on the matplotlib 3D tutorial.

```
from mpl_toolkits import mplot3d
fig = plt.figure(figsize=(12,10))
ax = plt.axes(projection='3d')
_ = ax.scatter3D(xi, yi, zi, alpha=0.4, label='before translation')
_ = ax.scatter3D(xf, yf, zf, alpha=0.4, label='after translation')
_ = ax.set_xlabel('x')
_ = ax.set_ylabel('y')
_ = ax.set_zlabel('z')
_ = ax.legend(frameon=False, fontsize=18)
```

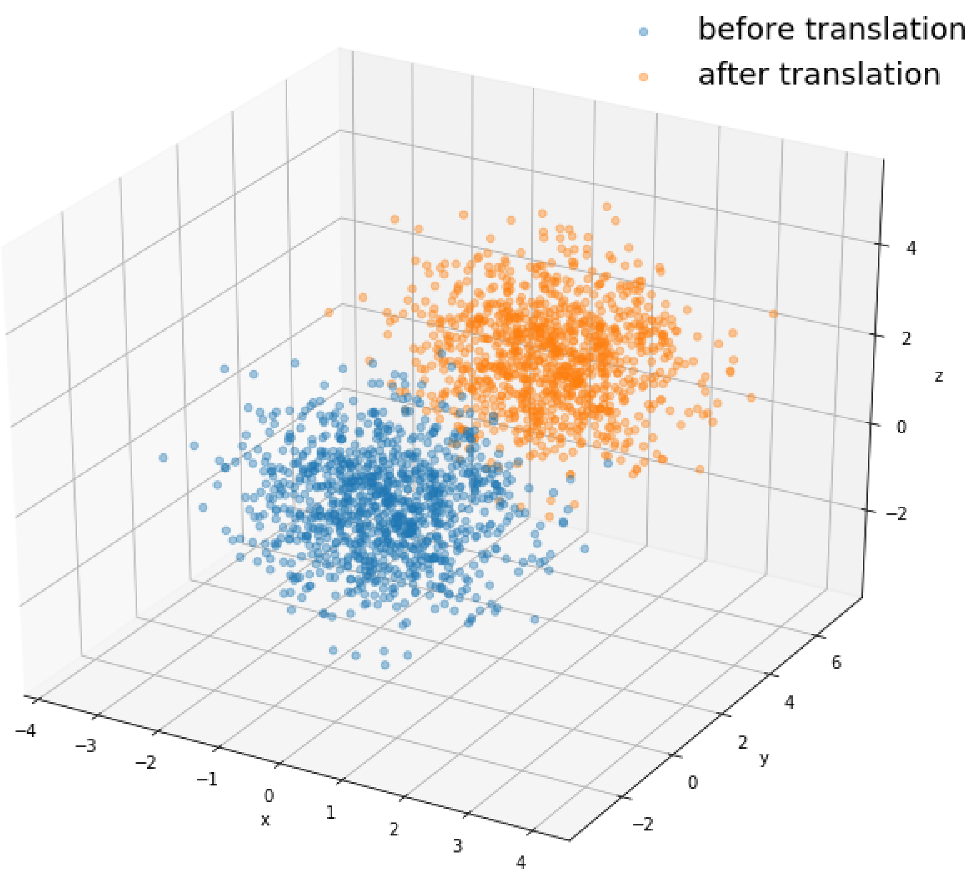

Figure 3

1.3.2 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 contanining a lot more features). There is no way to fully desribe this package here, the goal is simply to give functional and concrete example easily usable. More more details, please check the pandas webpage.

Many build-in functions are available to import data as pandas dataframe. One, which is particularly convenient, directly reads csv files (one can specify the columns to loads, the row to skip, and many other options . . .):

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

```
            HT  njet  nlep  pt_1st_bjet
0   262.100311     2     2    48.112684
1   447.937225     4     4   118.460391
2  1287.348022     6     6    89.715039
3   453.677887     6     6    88.535555
4   268.445099     2     2   116.625023
```

On of the nice feature of pandas is to be able to easily get numpy array, compute and store the result as a new column. For instance, it's a common practice in machine learning to *normalize* the input variables, *i.e.* transform them to have a mean of 0 and a variance of 1.0. The following example shows how to add new $H_T$ distributions (the meaning of this variable doesn't matter for now) as new columns:

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

# Compute 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

# Print the result
cols_to_print = ['HT', 'HT_centered', 'HT_normalized']
print(df[cols_to_print].head())
```

```
            HT  HT_centered  HT_normalized
0   262.100311  -254.826585      -0.895919
1   447.937225   -68.989671      -0.242554
2  1287.348022   770.421127       2.708646
3   453.677887   -63.249009      -0.222371
4   268.445099  -248.481797      -0.873612
```

One can simply plot the content of a pandas dataframe using the name of the column. For instance, one can compare the evolution of $H_T$ after each transformation (which is trivial in this illustrative case):

```
plt.figure(figsize=(20, 6))
plt.subplot(121)
ax = plt.hist(df['HT'], bins=100, alpha=0.5, label='Original $H_T$')
ax = plt.hist(df['HT_centered'], bins=100, alpha=0.5, label='$<H_T> = 0$')
ax = plt.legend(frameon=False, fontsize='xx-large')
plt.subplot(122)
ax = plt.hist(df['HT_normalized'], bins=100, alpha=0.5, label='$<H_T> = 0$
 ↪  and $\sigma_{H_T}$ = 1')
ax = plt.legend(frameon=False, fontsize='xx-large')
```

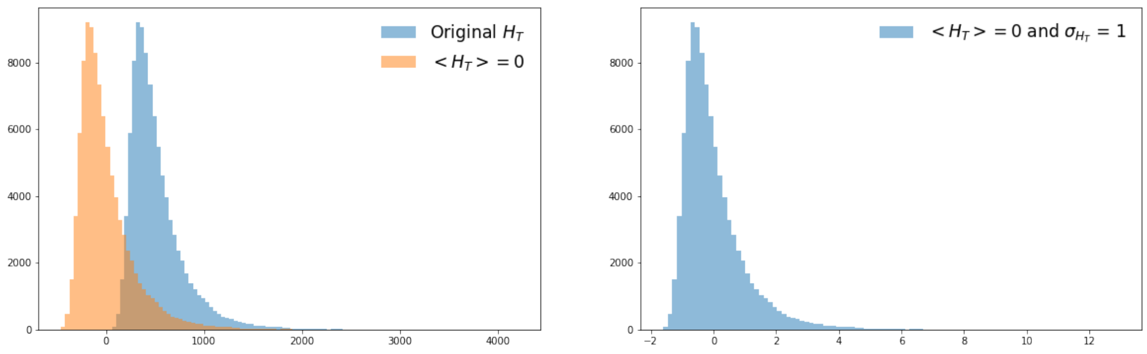

Figure 4

There are also many plotting function already included into the pandas library. To show only one example (all functions are decribed in the pandas visualization tutorial), here is the *scatter matrix* between variables (defined as a subset of the ones stored the dataframe) obtained in a single line of code:

```
from pandas.plotting import scatter_matrix
var_to_plot = ['HT', 'nlep', 'njet', 'pt_1st_bjet']
_ = scatter_matrix(df[var_to_plot], figsize=(12, 12), alpha=0.5)
```

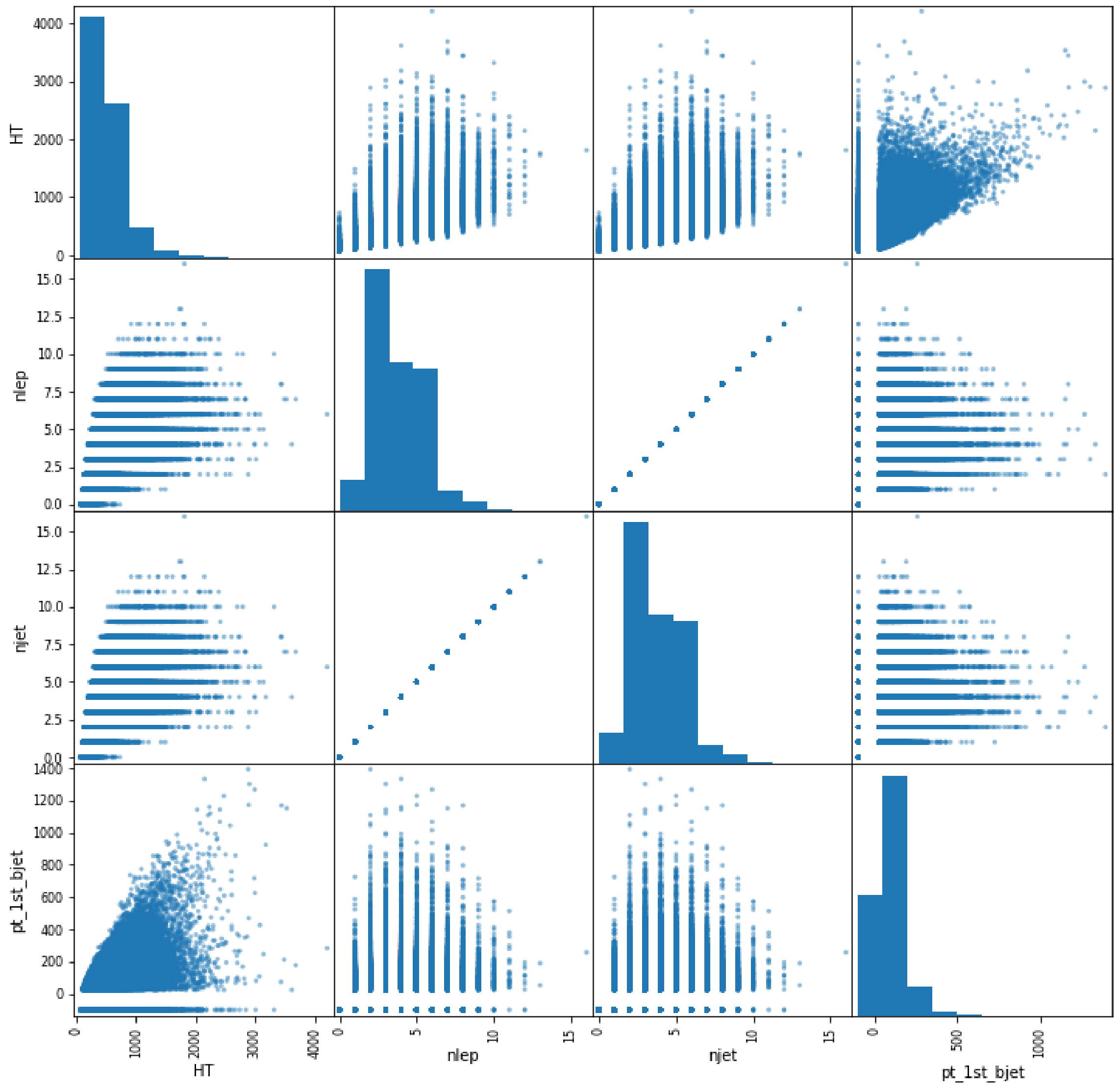

Figure 5

2 Use cases in high energy physics

The present chapter makes use of the concept previously introduced to perform computation that one would do in high energy (collider) physics. Indeed, in order to study collisions, one has to loop over several objects inside the collision, group them by pair, check which pair has the smallest angle, etc ... Since we want to use the full power of numpy, all these computation cannot be done with explicit loop over events and/or over objects. This chapter consider few of these typical use cases and their implementation using numpy, using a simple toy dataset made by hand. The case of more realistic collider data is treated in the next chapter.

Let's first perform the usual imports:

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

Then, one can setup he default matplolib style for all following plots (more details on available option can be found on how to customize matplotlib):

```
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 observations, each defined by ten 3D vectors $(r_0, ..., r_9)$ where $r_i = (x, y, z)$ (arrow for vector will be omitted from now on). 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 computation 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 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))
```

It is possible to print the first two observations as follow:

```
print(r[0:2])
```

```
[[[0.68548818 0.39730097 0.88989824]
  [0.71305264 0.54824144 0.37250525]
  [0.5849863  0.86030876 0.44899007]
  [0.05034541 0.14527878 0.03570552]
  [0.62308684 0.79489919 0.75938364]
  [0.23736053 0.19652811 0.11488176]
  [0.89063226 0.82614026 0.26800766]
  [0.27824247 0.74457781 0.8787299 ]
  [0.28506537 0.02729525 0.10248878]
  [0.35002015 0.56944671 0.40636309]]

 [[0.24409011 0.55433571 0.18935967]
  [0.54735398 0.93962054 0.13600911]
  [0.6752747  0.77728755 0.90581612]
  [0.3347568  0.22088931 0.73492331]
  [0.42393648 0.91903437 0.49677664]
  [0.05562273 0.65510172 0.31408092]
  [0.11290216 0.44595636 0.29465718]
  [0.9484048  0.18699368 0.41164779]
  [0.73753048 0.66599917 0.48844576]
  [0.52801902 0.00306445 0.35884336]]]
```

2.2 Mean over the differents axis

2.2.1 Mean over observations (axis=0)

This mean will average all observations *i.e.* 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 observations.

```
m0 = np.mean(r, axis=0)
print(m0.shape)
```

```
(10, 3)
```

Note the computation time of 30ms for 30 averages over a million number:

```
%timeit np.mean(r, axis=0)
```

```
33.9 ms ± 1.55 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

While it takes 10 times longer for a *single mean* over a million number with an explicit loop, so **the gain of vectorization is a factor 300**:

```
def explicit_loop(array):
    res=0
    for a in array:
        res += a/len(array)

%timeit explicit_loop(np.random.random_sample(size=1000000))
```

```
313 ms ± 6.33 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

The distributions of `m0` obtained with `plt.hist()` results into three separate histograms (one for each x, y, x) each having 10 entries (one per r_i):

```
ax = plt.hist(m0, label=['$<x>_{evts}$', '$<y>_{evts}$', '$<z>_{evts}$'])
ax = plt.title('10 entries, one for each $r_i$')
ax = plt.legend()
```

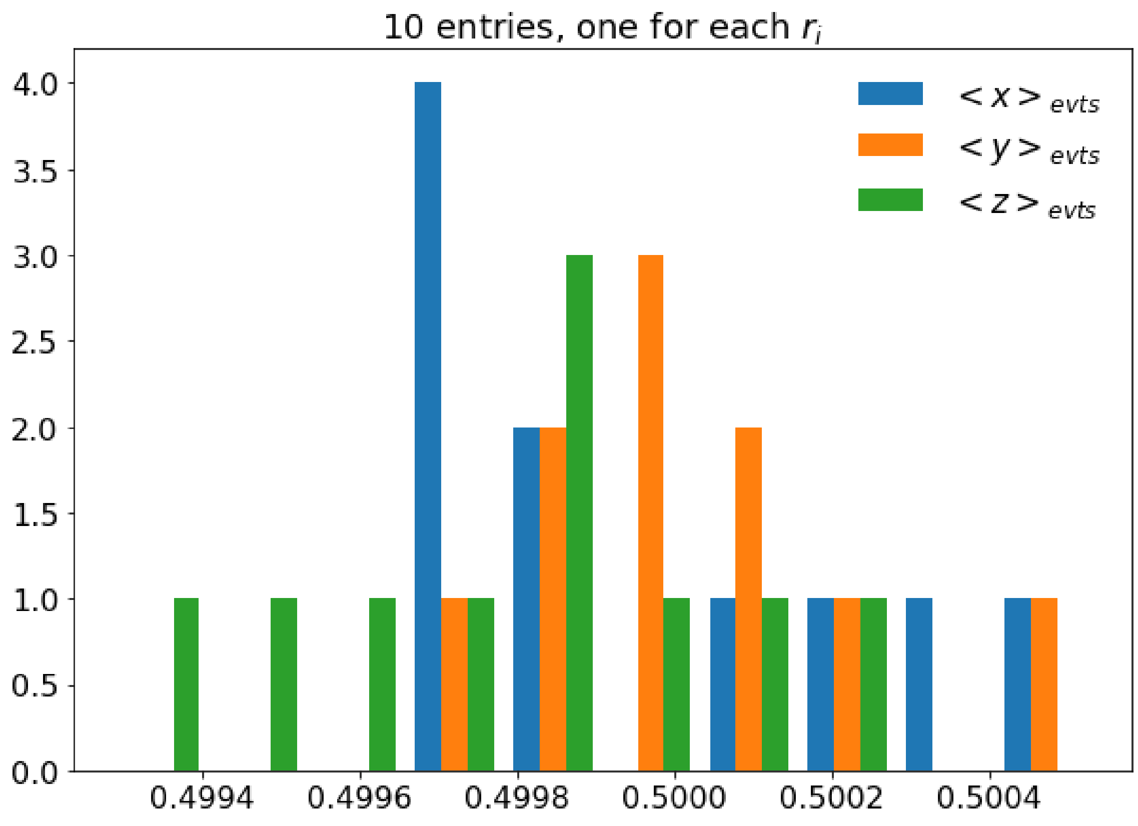

Figure 6

2.2.2 Mean over the 10 vectors (axis=1)

This one will compute the average over the 10 vectors, for each observations, reducing into a `(1000000, 3)` shape array, as seen below. This is 3D barycenter of each observation.

```
m1 = np.mean(r, axis=1)
print(m1.shape)
```

```
(1000000, 3)
```

One can plot the obtained array `m1` using `plt.hist()`, which results into 3 histograms of a million entry each:

```
ax = plt.hist(m1, label=['$<x>_{i}$', '$<y>_{i}$', '$<z>_{i}$'])
ax = plt.title('$10^6$ entries, one per event')
ax = plt.legend()
```

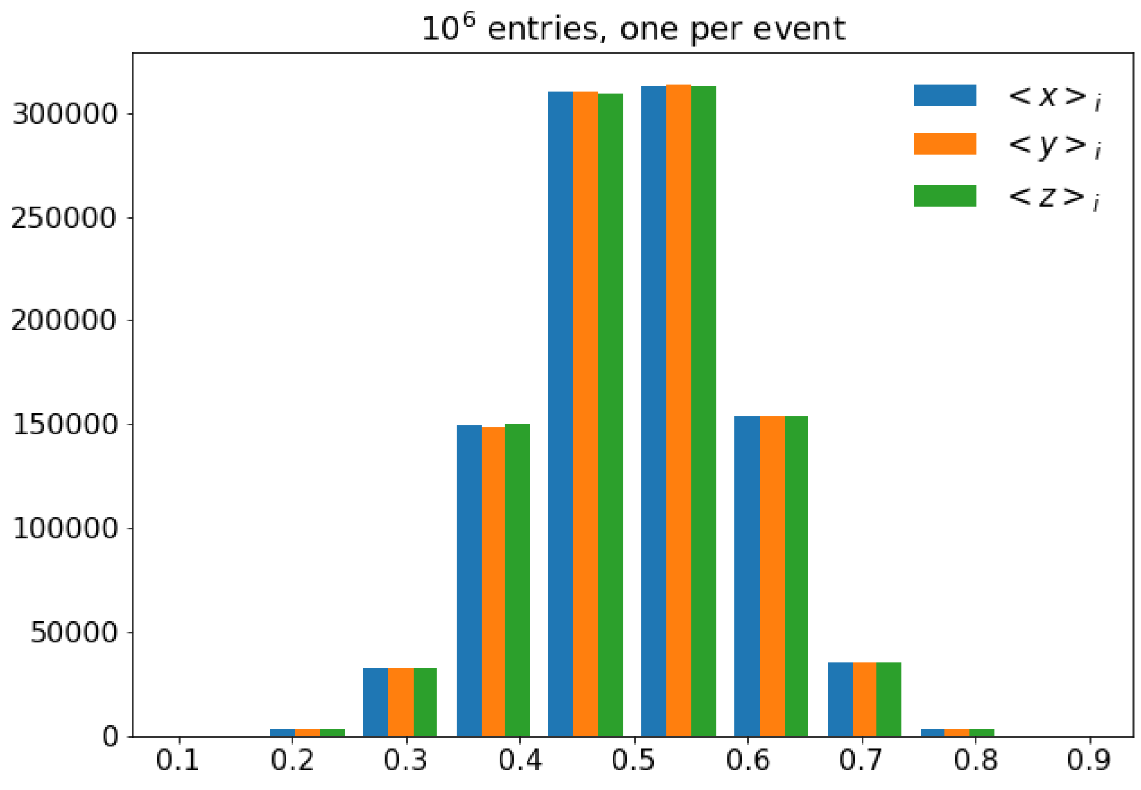

Figure 7

2.2.3 Mean over the coordintates (axis=2)

This directly computes the average over the three coordinates $(x+y+z)/3$ for each vector of each event, resulting in 10 values per event:

```
m2 = np.mean(r, axis=2)
print(m2.shape)
```

```
(1000000, 10)
```

The `plt.hist()` of the resulting array `m2` corresponds then to 10 histograms of a million entries each:

```
names = ['$(x+y+z)/3|_{'+'{}'.format(i)+'}$' for i in range(1, 11)]
ax = plt.hist(m2, label=names)
ax = plt.title('$10^6$ entries, one per event')
ax = plt.xlim(0, 1.5)
ax = plt.legend()
```

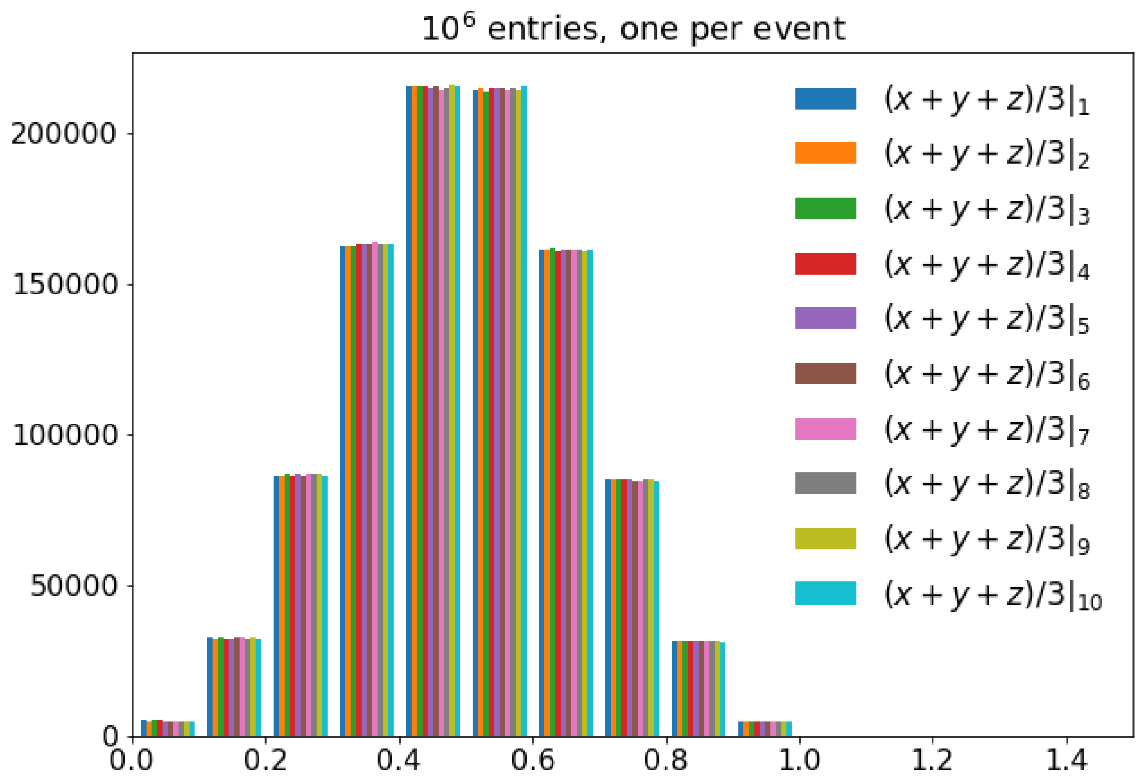

Figure 8

2.3 Distance computation

Computing particular distances inside a given event is relevant for many applications (distances here can be seen as any type of metric). For example, these computation are crucial in learning algorithms based on nearest neighbor approach. In collider physics, it's always useful to compute angle between two objects (tracks, deposit, particles, ...) in order to compute invariant masses, or isolation in a given cone, etc ...

2.3.1 Distance to a reference r_0

We can start simple by defining a new origin `r0`

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

and compute the distance to this new origin for all points, using `**2` to square all numbers, perform the sum over the coordinate (`axis=2`) and square-root everything with `**0.5`:

```
d = np.sum((r-r0)**2, axis=2)**0.5
print(d.shape)
```

```
(1000000, 10)
```

As expected the result is 10 numbers for each of the events, which can be easily plotted:

```
names = ['$d(r_{'+'{}'.format(i)+'},r_0)$' for i in range(1, 11)]
ax = plt.hist(d, label=names)
ax = plt.title('$10^6$ entries, one per event')
ax = plt.xlim(0.5, 3)
ax = plt.legend()
```

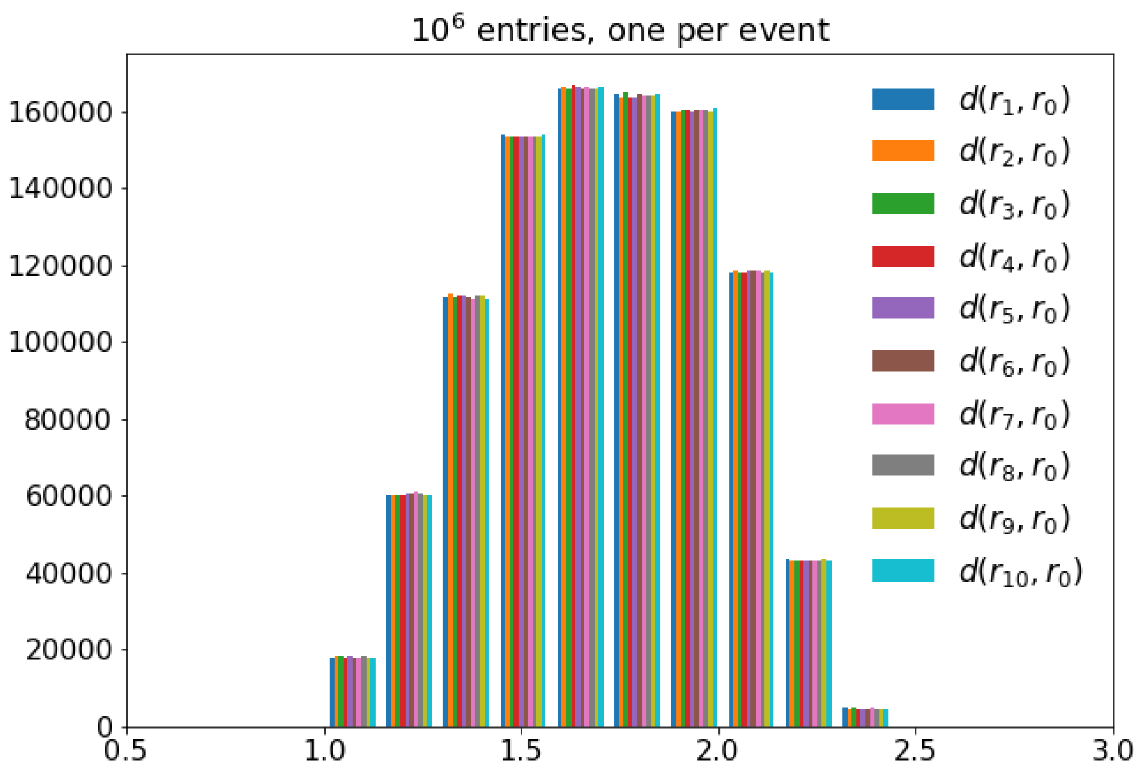

Figure 9

2.3.2 Distance between r_i and $< r >_i$ for each event

Another calculation is to compute the averaged position for each event and see how distant each vector is from this position. To perform such a calculation, we will use numpy array broadcasting. Let's first compute the average position for every events:

```
r_mean = np.mean(r, axis=1)
```

Now, let's broadcast this array of shape `(1e6, 3)` with the full dataset, *i.e.* an array of shape `(1e6, 10, 3)`, by computing the distance for each point:

```
try:
    d_to_mean = np.sum((r-r_mean)**2, axis=2)**0.5
except ValueError:
    print('Impossible for {} and {}'.format(r.shape, r_mean.shape))
```

```
Impossible for (1000000, 10, 3) and (1000000, 3)
```

There is one missing dimension, describing the 10 positions, which has to be created so that the array can be copied 10 times over along this dimension:

```
r_mean_3d = r_mean[:, np.newaxis, :]
```

We can now retry the operation:

```
try:
    d_to_mean = np.sum((r-r_mean_3d)**2, axis=2)**0.5
    print('Possible for {} and {}:'.format(r.shape, r_mean_3d.shape))
except ValueError:
    print('Impossible for {} and {}'.format(r.shape, r_mean_3d.shape))
```

```
Possible for (1000000, 10, 3) and (1000000, 1, 3):
```

```
names = ['$d(r_{'+'{}'.format(i)+'},<r>)$' for i in range(1, 11)]
ax = plt.hist(d_to_mean, label=names)
ax = plt.title('$10^6$ entries, one per event')
ax = plt.legend()
```

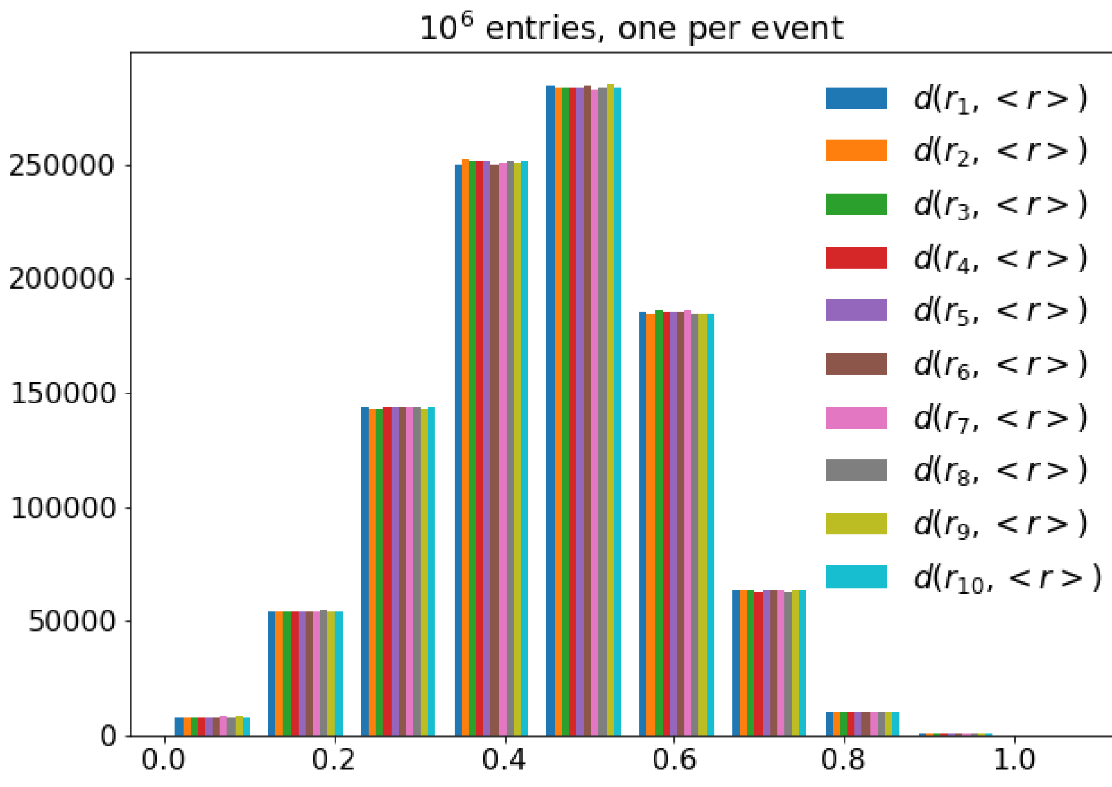

Figure 10

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

Being able to pair objects is obviously important for many type of calculations. This allows to probe correlations at the first order, to identify sub-systems, etc ... In a traditional way, a pairing would involve a *for* loop in which the combinatorics can be done for each event. Working with numpy, one has to perform the combinatorics in a vectorized way and return a new numpy array containing all the pairs. Once done, one can perform many types of computations on this new array.

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

One solution to perform such a task without *for* loop was found on stackoverflow. The idea is to simply work on indices to build the pairs (since it doesn't really matter what are the nature of the objects), and use numpy *fancy* indexing. Let proceed step by step with a smallest array to understand the procedure (namely 2 observations of 5 positions):

```
a = r[0:2,0:5]
print(a)
```

```
[[[0.68548818 0.39730097 0.88989824]
  [0.71305264 0.54824144 0.37250525]
  [0.5849863  0.86030876 0.44899007]
  [0.05034541 0.14527878 0.03570552]
  [0.62308684 0.79489919 0.75938364]]

 [[0.24409011 0.55433571 0.18935967]
  [0.54735398 0.93962054 0.13600911]
  [0.6752747  0.77728755 0.90581612]
  [0.3347568  0.22088931 0.73492331]
  [0.42393648 0.91903437 0.49677664]]]
```

Since we want to work with the indicies of the 5 vectors, we create a numpy array of integer going from 0 to 4 (`a.shape[1]` is the number of elements along the second dimension, *i.e.* 5):

```
array_indices = np.arange(a.shape[1])
print(array_indices)
```

```
[0 1 2 3 4]
```

Then, we use the package `itertools` to deal with the combinatorics. This will return an *iterator* that can be turned into a numpy array using `np.fromiter()`. But this function requires to specify the data type `dt`, which is done using a structured array synthax here (*i.e.*

`[(varName1,type1), (varName2,type2)])`. For more details on data type, check this documentation page.

```
dt = np.dtype([('index1', np.intp), ('index2', np.intp)])
print(dt)
```

```
[('index1', '<i8'), ('index2', '<i8')]
```

```
array_indice_comb = np.fromiter(itertools.combinations(array_indices, 2), dt)
print(array_indice_comb)
```

```
[(0, 1) (0, 2) (0, 3) (0, 4) (1, 2) (1, 3) (1, 4) (2, 3) (2, 4) (3, 4)]
```

The next step is to format these numbers in a indices array with the proper dimension, so that when we do `a[indices]`, we get all the pairs. For instance, we need to have all 10 pairs, each with two elements corresponding to a shape `indices.shape=(10,2)`. We can achieved this in two steps:

1. `array_indice_comb.view(np.intp)` return the exact same data of `array_indice_comb` as a 1D array of positive integer.
2. we reshape the resulting array with `reshape(-1, 2)`, where -1 means "compute the size of the first dimension to have 2 objects (we wants pair!) in the second dimension.

```
indices = array_indice_comb.view(np.intp).reshape(-1, 2)
print(indices)
```

```
[[0 1]
 [0 2]
 [0 3]
 [0 4]
 [1 2]
 [1 3]
 [1 4]
 [2 3]
 [2 4]
 [3 4]]
```

The final steps is exploit fancy indexing along `axis=1` *i.e.* the 5 spatial positions. In practice, for each observation `iobs`, we want to have `a[iobs, indices]`. There are two ways to do this: (a) `a[:, indices]` or (b) using the numpy function `np.take(a, indices, axis)` which makes the code more independant from the structure of `a`:

```
a_pairs = np.take(a, indices, axis=1)
print(a_pairs.shape)
```

```
(2, 10, 2, 3)
```

```
a_pairs = a[:,indices]
print(a_pairs.shape)
```

```
(2, 10, 2, 3)
```

We have now 2 events, each having 10 pairs, each having 2 objects (still a pair!), each having 3 coordinates (spatial positions). We can print all the 10 pairs for the first observation:

```
print(a_pairs[0])
```

```
[[[0.68548818 0.39730097 0.88989824]
  [0.71305264 0.54824144 0.37250525]]

 [[0.68548818 0.39730097 0.88989824]
  [0.5849863  0.86030876 0.44899007]]

 [[0.68548818 0.39730097 0.88989824]
  [0.05034541 0.14527878 0.03570552]]

 [[0.68548818 0.39730097 0.88989824]
  [0.62308684 0.79489919 0.75938364]]

 [[0.71305264 0.54824144 0.37250525]
  [0.5849863  0.86030876 0.44899007]]

 [[0.71305264 0.54824144 0.37250525]
  [0.05034541 0.14527878 0.03570552]]

 [[0.71305264 0.54824144 0.37250525]
  [0.62308684 0.79489919 0.75938364]]

 [[0.5849863  0.86030876 0.44899007]
  [0.05034541 0.14527878 0.03570552]]

 [[0.5849863  0.86030876 0.44899007]
  [0.62308684 0.79489919 0.75938364]]
 [[0.05034541 0.14527878 0.03570552]
  [0.62308684 0.79489919 0.75938364]]]
```

Once understood, we can wrapp-up this code into a function where we generalize the number of objects we want to group `n` and the axis along which we want to group `axis`:

```
def combs_nd(a, n, axis=0):
    i = np.arange(a.shape[axis])
    dt = np.dtype([('', np.intp)]*n)
    i = np.fromiter(itertools.combinations(i, n), dt)
    i = i.view(np.intp).reshape(-1, n)
    return np.take(a, i, axis=axis)
```

As a sanity check, we can re-compute `a_pair` and compare with the previous results:

```
a_pairs = combs_nd(a=r[0:2,0:5], n=2, axis=1)
print(a_pairs[0])
```

```
[[[0.68548818 0.39730097 0.88989824]
  [0.71305264 0.54824144 0.37250525]]

 [[0.68548818 0.39730097 0.88989824]
  [0.5849863  0.86030876 0.44899007]]

 [[0.68548818 0.39730097 0.88989824]
  [0.05034541 0.14527878 0.03570552]]

 [[0.68548818 0.39730097 0.88989824]
  [0.62308684 0.79489919 0.75938364]]

 [[0.71305264 0.54824144 0.37250525]
  [0.5849863  0.86030876 0.44899007]]

 [[0.71305264 0.54824144 0.37250525]
  [0.05034541 0.14527878 0.03570552]]

 [[0.71305264 0.54824144 0.37250525]
  [0.62308684 0.79489919 0.75938364]]

 [[0.5849863  0.86030876 0.44899007]
  [0.05034541 0.14527878 0.03570552]]
 [[0.5849863  0.86030876 0.44899007]
  [0.62308684 0.79489919 0.75938364]]

 [[0.05034541 0.14527878 0.03570552]
  [0.62308684 0.79489919 0.75938364]]]
```

It can be intersting to see that this operation takes less than a second for a million observations of 10 vectors, meaning 45 pairs:

```
%timeit combs_nd(a=r, n=2, axis=1)
```

```
862 ms ± 29.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

2.4.2 Computing (minimum) distances on these pairs

Once we have these pairs, we can for example computes all the distances and find which pair has the closest objects. Starting with the pairs:

```
pairs = combs_nd(a=r, n=2, axis=1)
```

We can then define the vectorial difference between the two position of a pair, and compute the euclidiean distance:

```
dp = pairs[:, :, 0, :]-pairs[:, :, 1, :]
distances = (np.sum(dp**2, axis=2))**0.5
```

And get the minimum distance for each event:

```
smallest_distance = np.min(distances, axis=1)
print(smallest_distance.shape)
```

```
(1000000,)
```

All these instructions can be put into a function which can be timed:

```
def compute_dr_min(a):
    pairs = combs_nd(a, 2, axis=1)
    i1 = tuple([None, None, 0, None])
    i2 = tuple([None, None, 1, None])
    return np.min(np.sum((pairs[i1]-pairs[i2])**2, axis=2)**0.5, axis=1)
```

```
%timeit compute_dr_min(r)
```

```
841 ms ± 8.04 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Note that doing all operations in the less possible amount of lines can significantly speed up the process. Let's define another function where the difference between the pair elements is done separately:

```
def compute_dr_min_more_steps(a):
    pairs = combs_nd(a, 2, axis=1)
    dp = pairs[:, :, 0, :]-pairs[:, :, 1, :]
    return np.min(np.sum(dp**2, axis=2)**0.5, axis=1)
```

And let's compare the performance on 0.2 million observations:

```
%timeit compute_dr_min(a=r[:200000])
%timeit compute_dr_min_more_steps(a=r[:200000])
```

```
168 ms ± 1.23 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
570 ms ± 3.84 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Let's now plot the distributions of all distances for all the pairs (using `flatten()` function which returns a 1D array), and only the pair having the smallest distances:

```
plot_style = {
    'bins': np.linspace(0, 2, 100),
    'alpha': 0.5,
    'density': True,
}
ax = plt.hist(distances.flatten(), label='All pairs' ,**plot_style)
ax = plt.hist(smallest_distance, label='Closest pair', **plot_style)
ax = plt.xlabel('Distance')
ax = plt.ylabel('Arbitrary Unit')
ax = plt.legend()
```

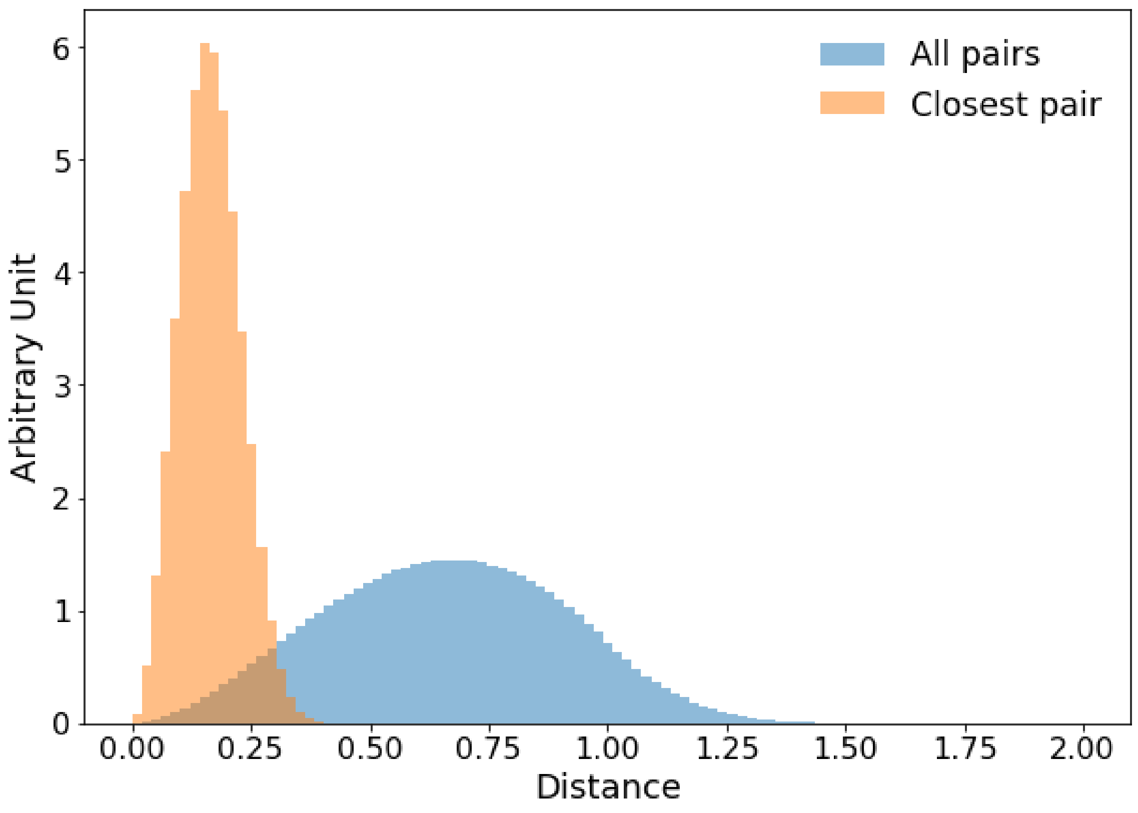

Figure 11

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

The next step in our exploration “loop-less calculations” is to be able to perform the same kind of computation described above but only on a subset of positions, selected according to a given criteria. For example, we might want to keep particles only if there have positive charge. Many obvious application can be found in other physics field and/or machine learning. Let’s start with accessing the three arrays of coordinates in order to select points based on some easy criteria.

```
x, y, z = r[:, :, 0], r[:, :, 1], r[:, :, 2]
```

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

We will use the numpy masking feature described in the first chapter, defining a index of boolean based on x and y arrays:

```
idx = x > y
print(idx.shape)
```

```
(1000000, 10)
```

We can quickly check the distribution for the selected coordinates: x and y are anty correlated - as expected - while z is flat - as expected.

```
ax = plt.hist(x[idx], bins=100, alpha=0.2, label='x')
ax = plt.hist(y[idx], bins=100, alpha=0.2, label='y')
ax = plt.hist(z[idx], bins=100, alpha=0.3, label='z')
ax = plt.legend()
```

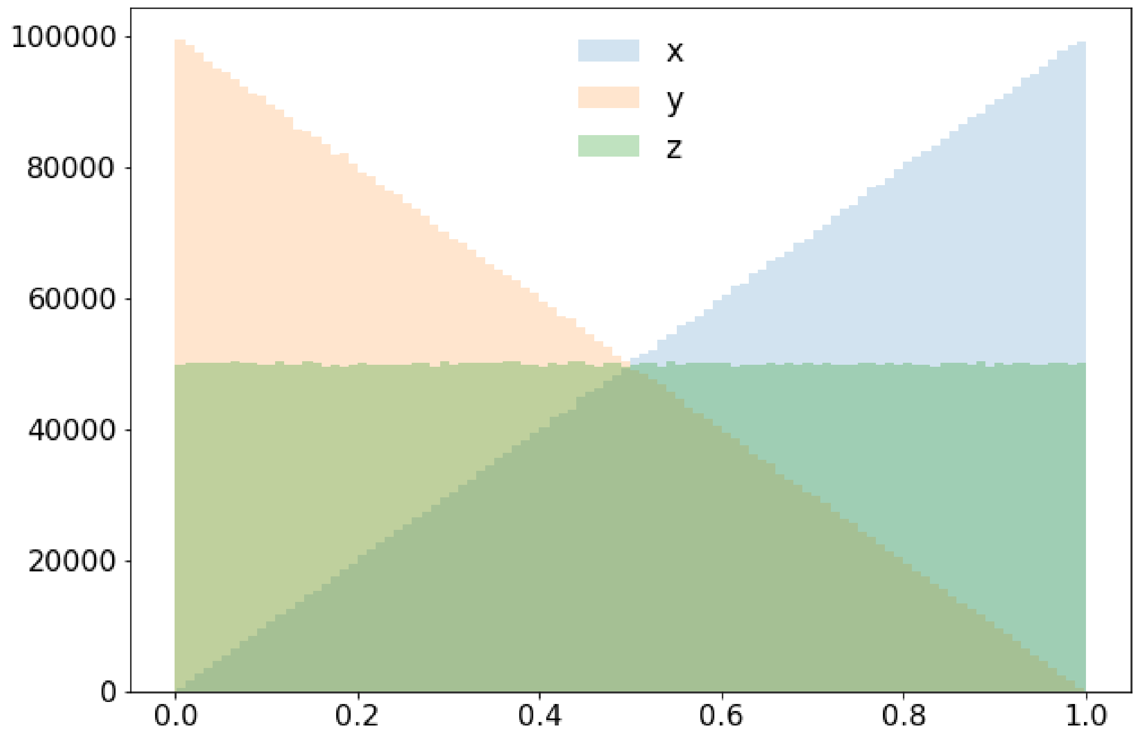

Figure 12

If we want to better understand how this selection affect our data, one might want to count the number of points per event satisfying this selection, using `np.count_nonzero()` on the boolean array along the axis representing the 10 vectors `axis=1`:

```
c = np.count_nonzero(idx, axis=1)
print(c.shape)
```

```
(1000000,)
```

We can then plot the distribution of this number over all the events:

```
ax = plt.hist(c, bins=20, alpha=0.5)
```

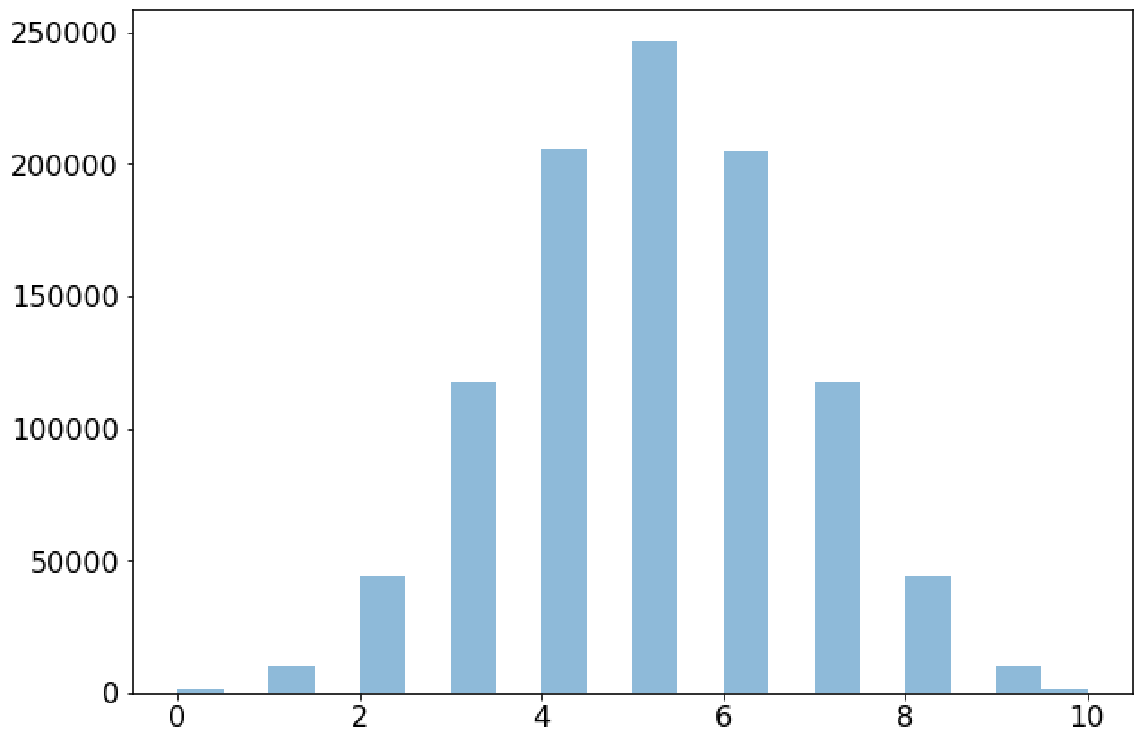

Figure 13

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

This is obviously useful to inspect the different populations - something we want to do very often. For the plotting purpose, let's consider only the 500 first observations that we dump into `sx`, `sy`, `sz` (s for small):

```
sx, sy, sz = x[0:500, ...], y[0:500, ...], z[0:500, ...]
```

We define the mask computed on these small arrays `smask`:

```
smask = sx>sy
```

And we can plot the result in the 2D plane (x, y) with the z coordinate as marker size, for instance $1/(z + 10^{-3})$. The two populations are defined using both `smask` and `~smask` to make sure the union of the two is the original dataset:

```
ax = plt.scatter(sx[smask], sy[smask], s=(sz[smask]+1e-3)**-1, label='$x>y$')
ax = plt.scatter(sx[~smask], sy[~smask], s=(sz[~smask]+1e-3)**-1,
↪  label='$x\leq y$')
ax = plt.xlabel('x')
ax = plt.ylabel('y')
ax = plt.xlim(-0.03, 1.3)
ax = plt.legend()
```

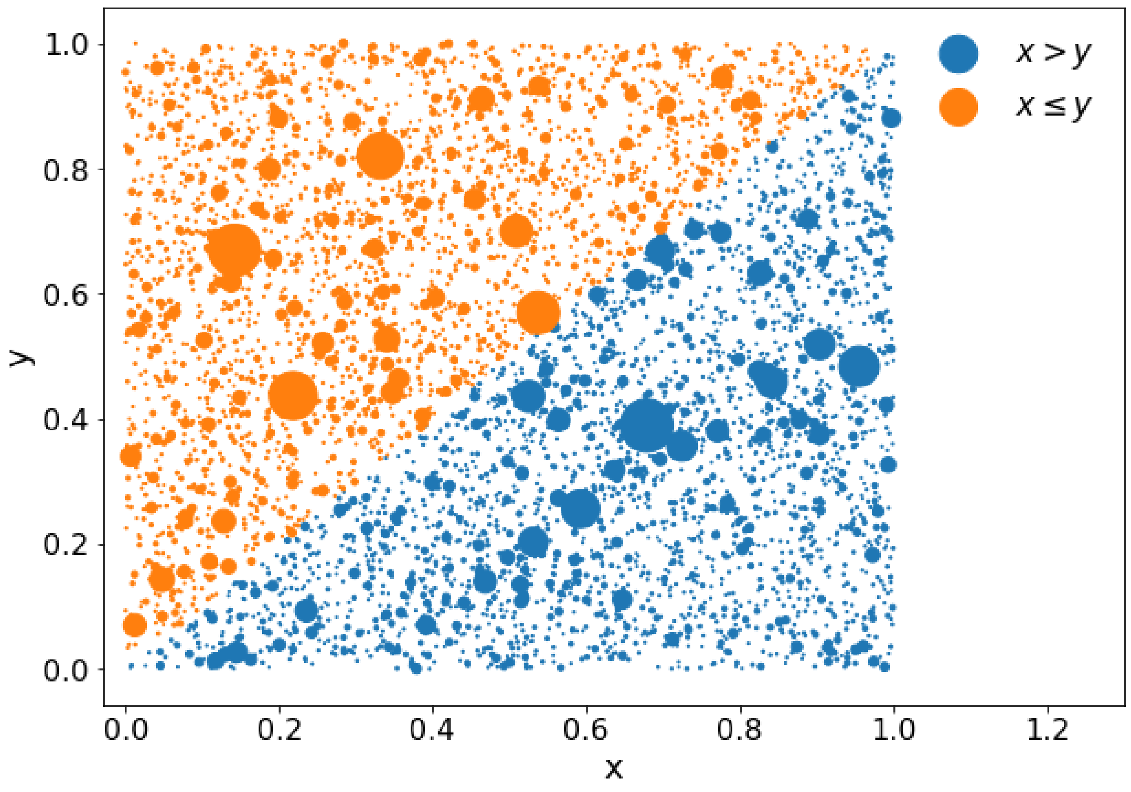

Figure 14

2.5.3 Computation of $x_i + y_i + z_i$ sum over a subset of the 10 positions

Once we are able to isolate a subset of points, we might to computes new numbers only based on those. This is what is proposed here with the sum of the three coordintates. Let's first compute and the sum, called `ht`, over all the 10 points:

```
ht1 = np.sum(x+y+z, axis=1)
print(ht1.shape)
```

```
(1000000,)
```

Apply now a selection, which multiply the value by 0 (*i.e.* `False`) if the condition is not satistifed:

```
selection = x>y
ht2 = np.sum((x+y+z)*selection, axis=1)
```

Of course, this works only for computation which is not affected by a 0: if we want to compute the product of coordinate, this approach will obvioulsy not work.

```
prod = np.product((x+y+z)*selection, axis=1)
eff  = np.count_nonzero(prod>0)/len(prod)
print('Efficiency of prod>0: {:.5f}'.format(eff))
```

```
Efficiency of prod>0: 0.00096
```

In a more general manner, we should use *masked arrays* which completely remove the masked elements from any computations:

```
mx = np.ma.array(x, mask=selection)
my = np.ma.array(y, mask=selection)
mz = np.ma.array(z, mask=selection)
prod = np.product((mx+my+mz), axis=1)
eff  = np.count_nonzero(prod>0)/len(prod)
print('Efficiency of prod>0: {:.5f}'.format(eff))
```

```
Efficiency of prod>0: 1.00000
```

Finally one can plot the result, removing the observation with `ht2==0` (case where all the 10 points have $x \leq y$):

```
ax = plt.hist(ht1, bins=100, alpha=0.4, label='All points')
ax = plt.hist(ht2[ht2>0], bins=100, alpha=0.4, label='Only x>y')
ax = plt.xlabel('$\sum_{i} \; (x_i+y_i+z_i)$')
ax = plt.legend()
```

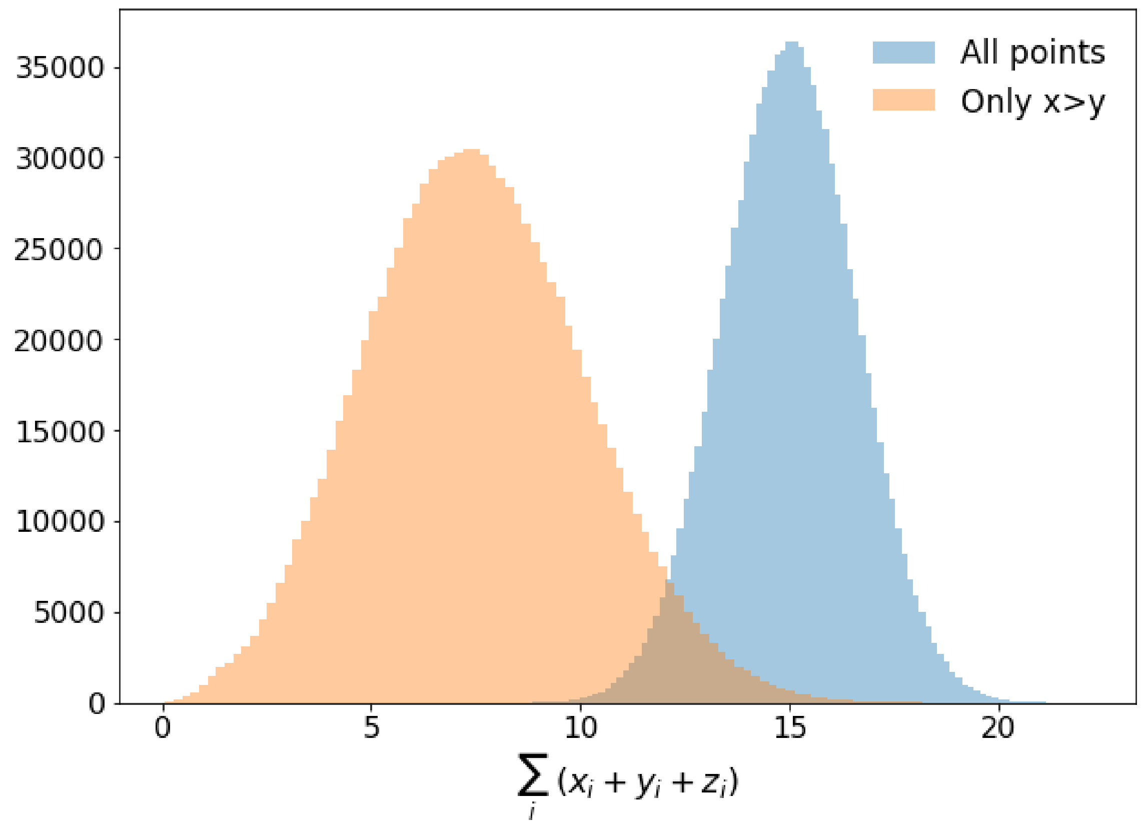

Figure 15

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

Another computation would be to redo the pairing on the subset of selected position. In order to do so, we follow the same logic, expect that we will directly replace removed values by nan in order to be easily identifiable in after the pairing. It's *very important to copy the orignal data with the module* copy, otherwise, the orignal data will be modifed in the following piece of code:

```
import copy
selection = x>y
selected_r = copy.copy(r)
selected_r[selection] = np.nan
print(selected_r[0])
```

```
[[0.34732569 0.54325773 0.21379696]
 [       nan        nan        nan]
 [       nan        nan        nan]
 [       nan        nan        nan]
 [0.51031398 0.66611858 0.40716321]
 [0.25743862 0.26478221 0.06134177]
 [       nan        nan        nan]
 [       nan        nan        nan]
 [0.61071772 0.84854567 0.1629833 ]
 [       nan        nan        nan]]
```

On can now calling the paring function on the filtered dataset:

```
selected_pairs = combs_nd(selected_r, n=2, axis=1)
```

And compute the distances, but replacing back the `np.nan` by a default values that will not be seen on a plot.

```
p1, p2 = pairs[:, :, 0, :], pairs[:, :, 1, :]
dp = np.sum((p1-p2)**2, axis=2)**0.5
dp[np.isnan(dp)] = 999
```

And plotting the distributions of both all distances and minimum distances for pairs made out of points verifying $x > y$:

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

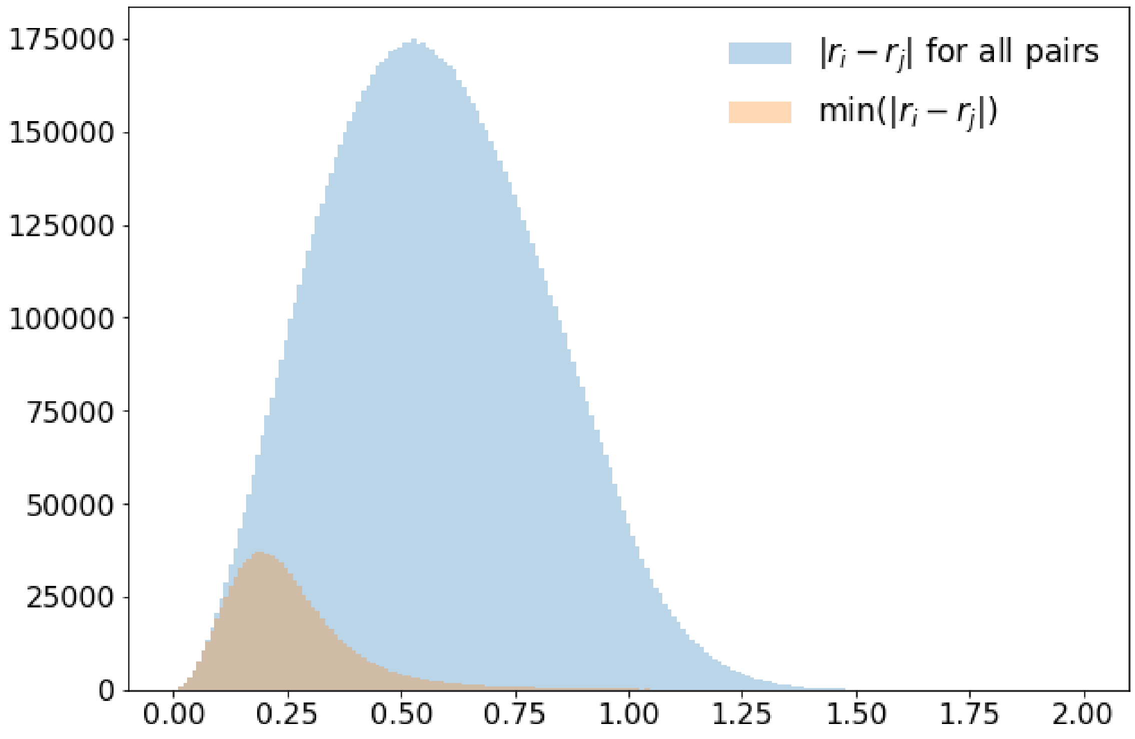

Figure 16

2.6 Some concluding comments

Manipulating numpy array is quite powerful and fast for both computation and plotting, at the condition that we use numpy optimization, namely vectorization, indexing and broadcasting. This is ofter possible when this has also some limitations as we saw above. Namely, we add to play a bit with “patchwork approaches” to achieve what we want without loops in the last two sections. Typically, what will work for one computation will not work for another (replacing rejected values by 0 works for an addition and not for a product). For the pairing as well, we had to replace all rejected values by `np.nan` in order to filter them later on. This kind of practice makes things less readable when complexity increases - according to me. Or maybe there are smarter ways to do things.

3 Collider data analysis and limitations

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

# ROOT numpy interface (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)
print(df[['jet_pt', 'el_pt']].head())
```

```
                                              jet_pt                    el_pt
0                             [169695.5, 122250.03]  [55366.094, 38978.633]
1           [92278.93, 70800.66, 69653.164, 27776.486]                      []
2  [56349.285, 43751.82, 36588.938, 35095.082, 27...             [76494.64]
3                            [59820.547, 41592.062]             [39917.418]
4  [196711.52, 123898.07, 87307.625, 82197.49, 41...             [197385.73]
```

```
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-dimenional 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 formating of the array. The construction of this function is detailed (and timed) after.

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

```python
# 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 raws of a jagged 2D array.

    This function equalizes the the size of every raw (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 raws.
    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 µs 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 False]
 [ True  True  True  True False 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.]]
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 `(Nevts,Njets,2)`, where 2 corresponds to the number of variables. This function is 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_isbtagged_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'])
```

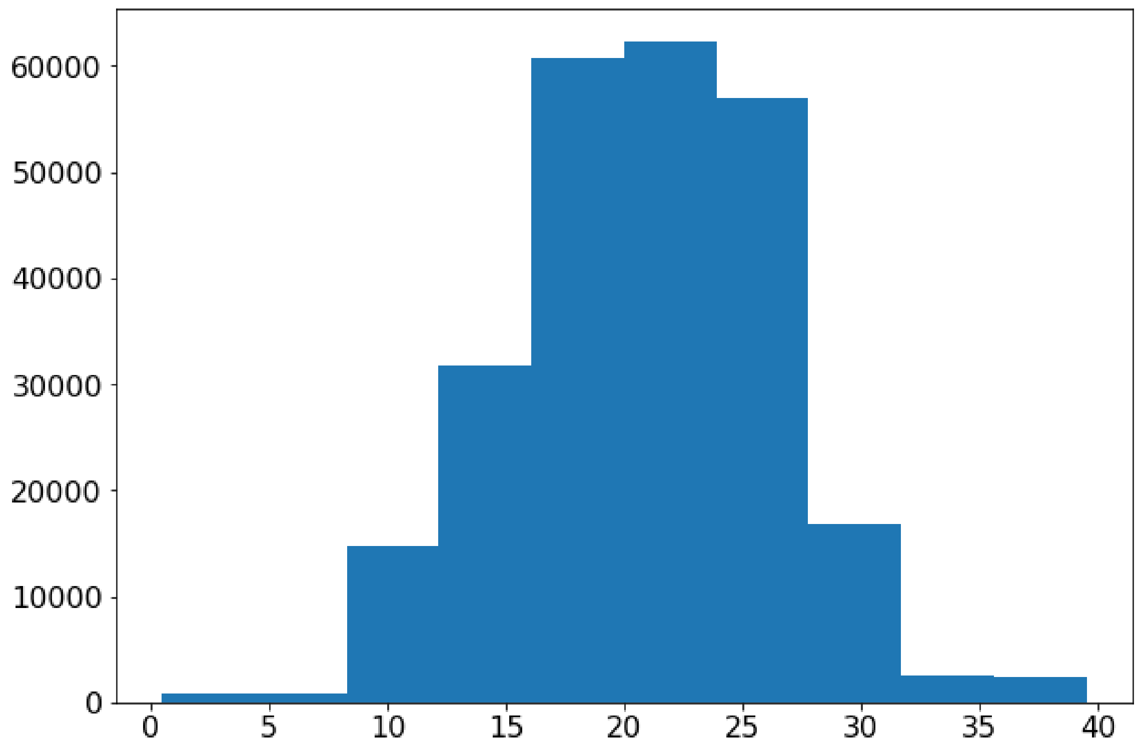

Figure 17

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 dimention but the last one is inclusive (here it means all events and all jets for each events), 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 arrays filled of shape (Nevts,Njets) with `True` and `False` depending on wether the element is above `pt` or not; + `np.count_nonzero(jets[...,0]>pt, axis=1)` is 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')
```

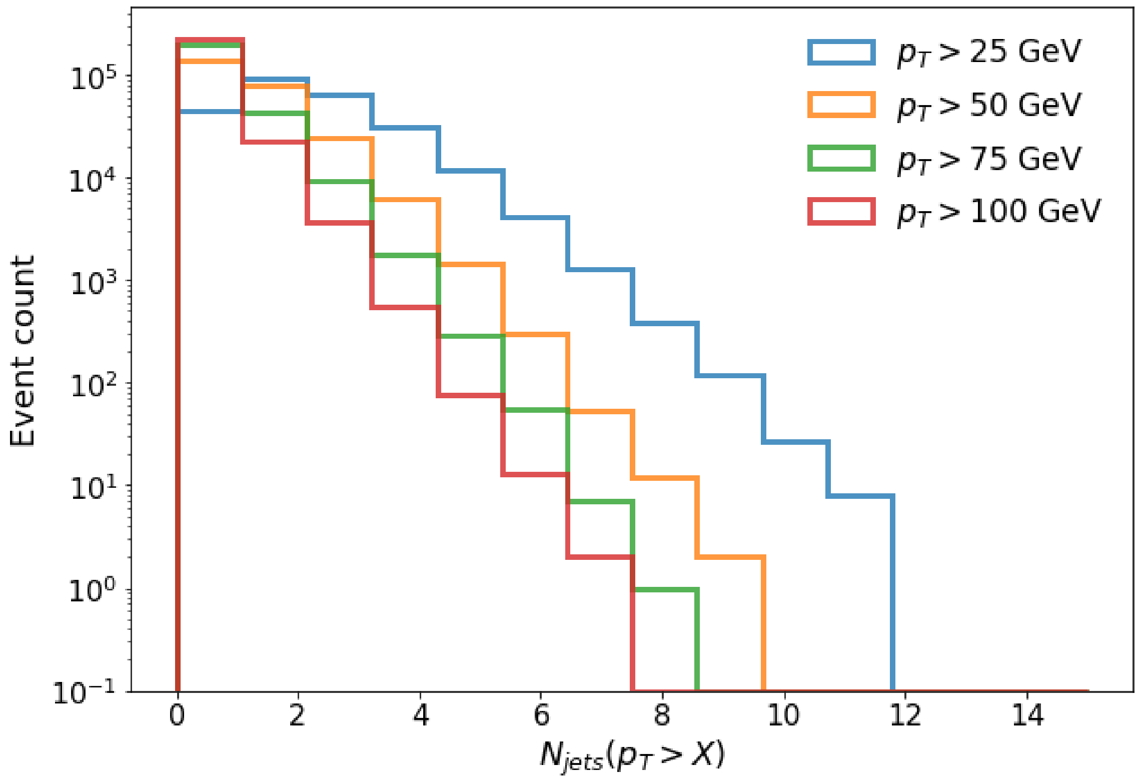

Figure 18

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')
```

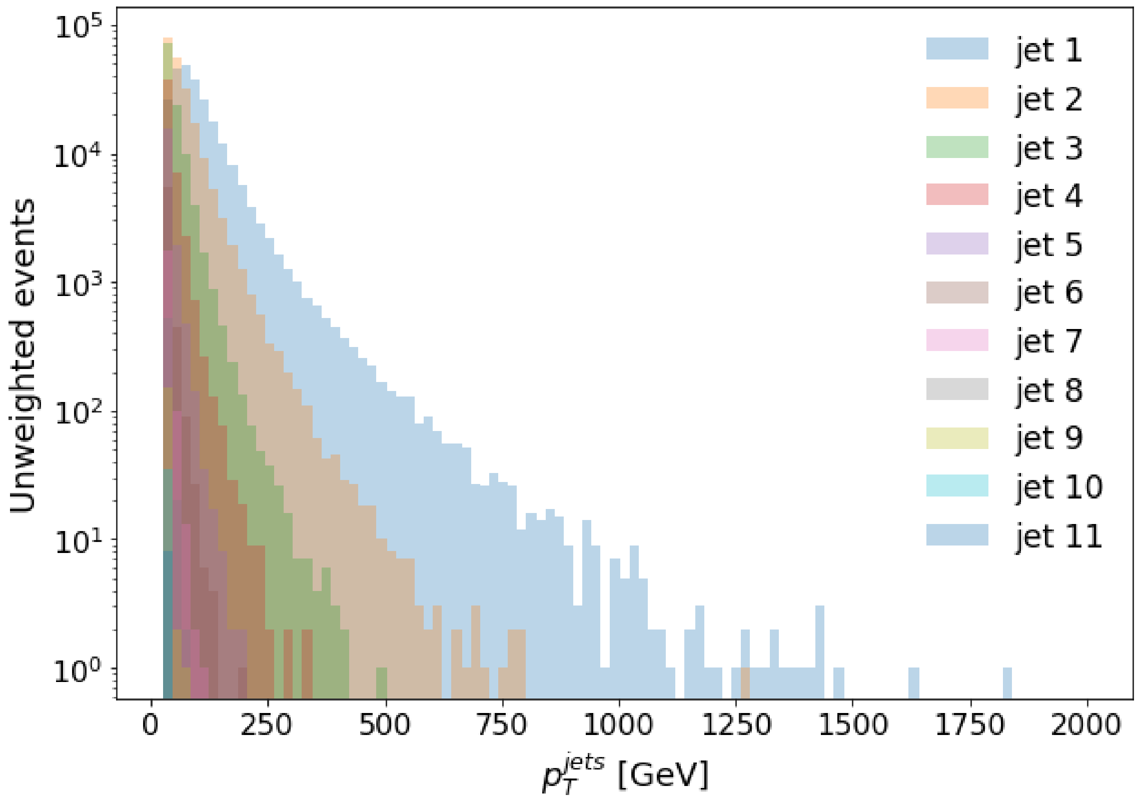

Figure 19

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 diffent commands is: + `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 by `False` or `0`. + if `a` is multi-dimentional, `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 depends 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
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); + Compte 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; + Compte 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='$t\\bar{t}$')
```

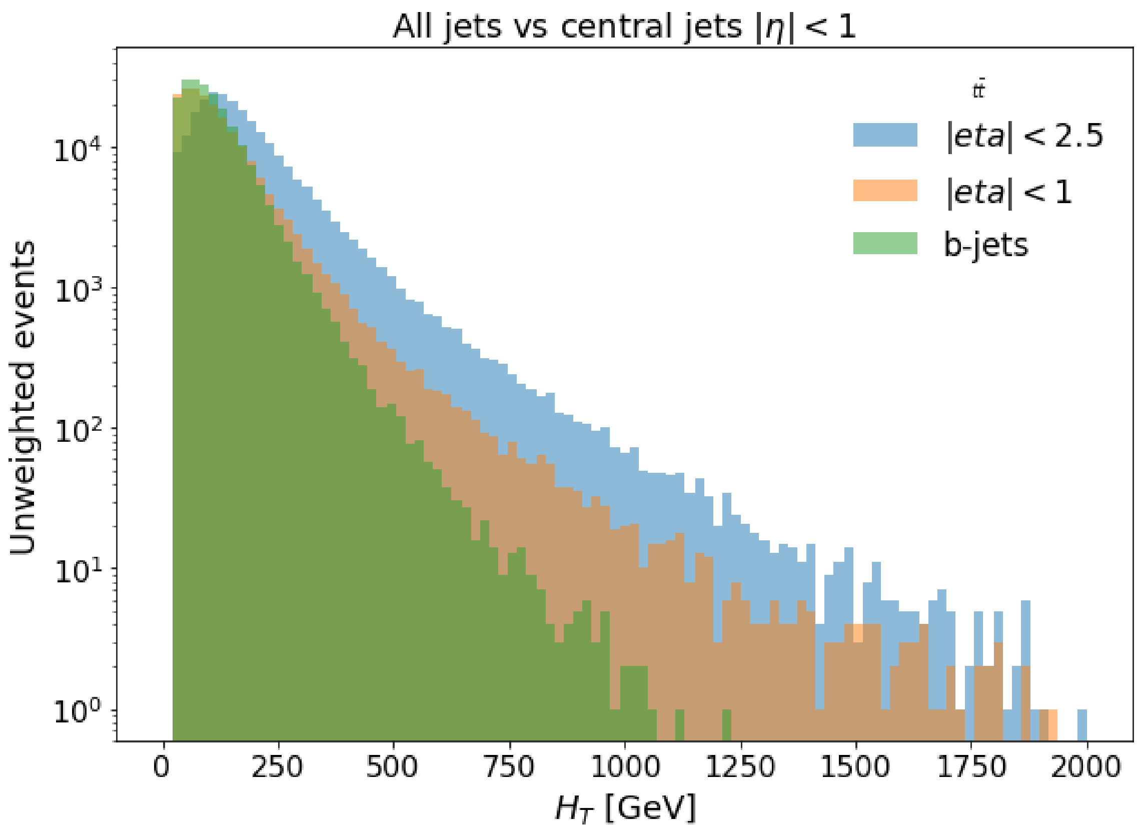

Figure 20

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 anthing else based on `j1` and `j2`. 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 $\Delta R$')
ax = plt.xlabel('$\Delta R(j,j$)')
ax = plt.ylabel('Unweighted events')
ax = plt.legend()
```

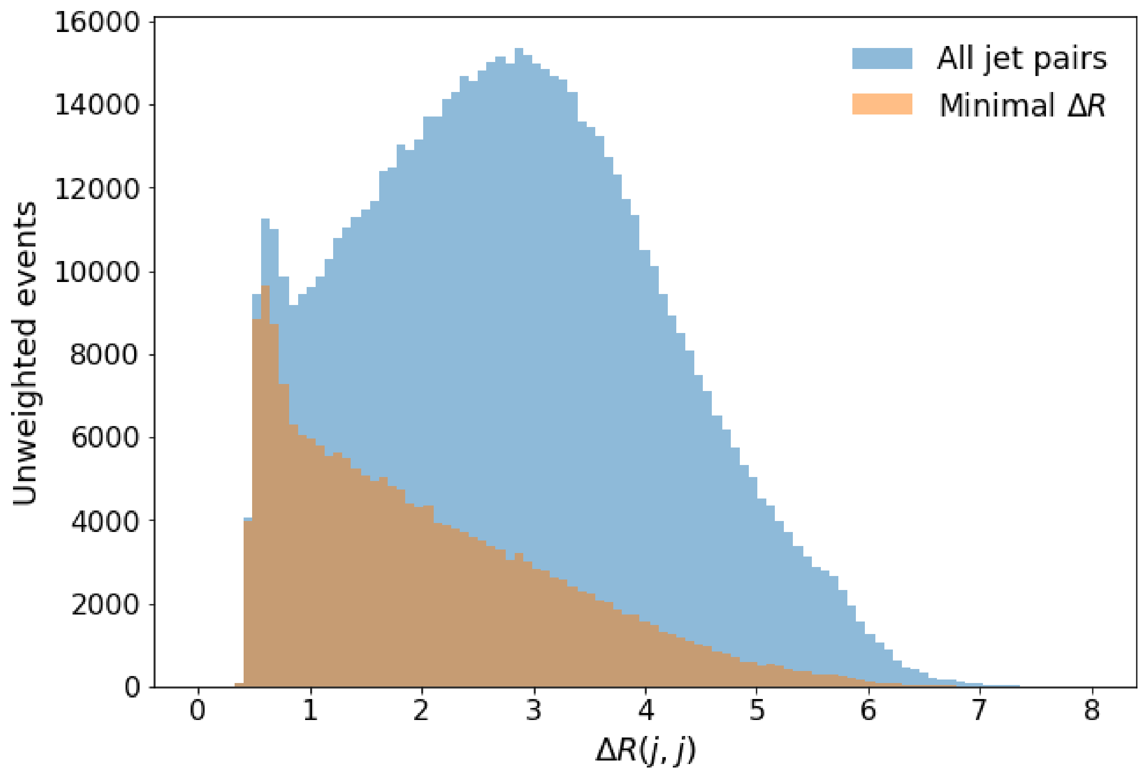

Figure 21

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 $\Delta R$', **style)
ax = plt.xlabel('$\Delta R(j,e)$')
ax = plt.ylabel('Unweighted events')
ax = plt.legend()
```

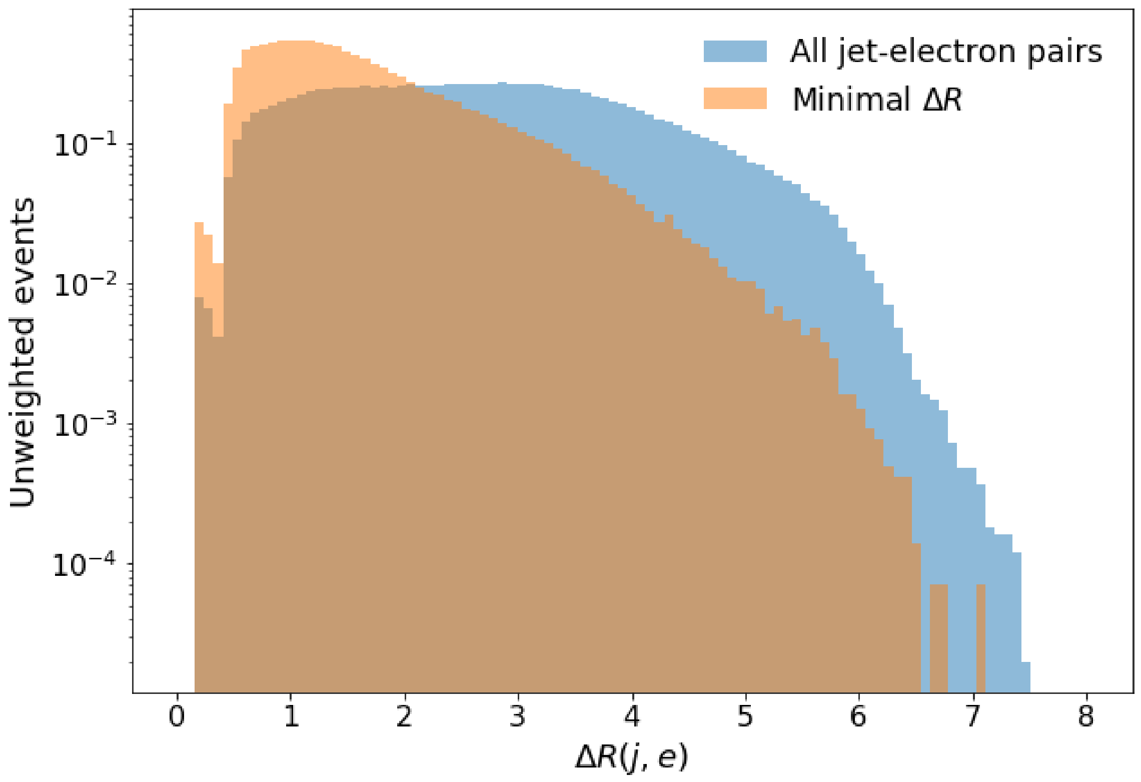

Figure 22

3.4.4 Di-jet invariant masses

Let's take the example of the invariant mass betwee `j1` and `j2`:

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

```
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()
```

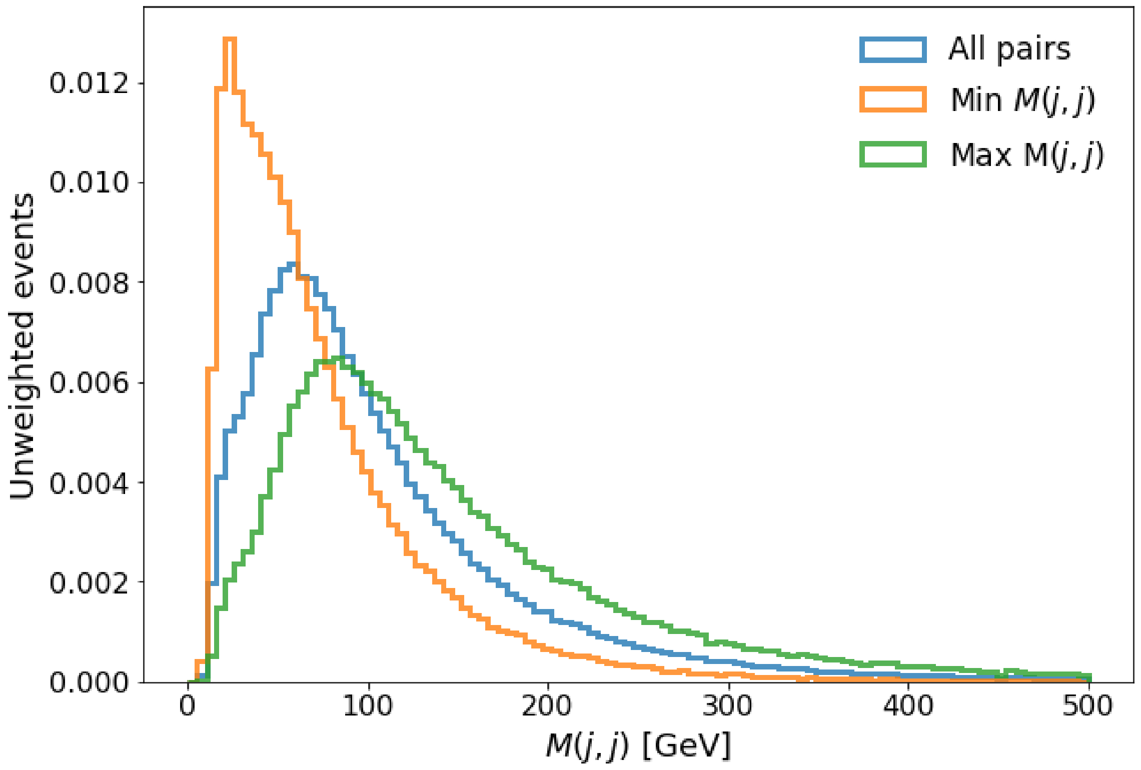

Figure 23

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

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

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

This would be useful to work around the constrain of having the same number of variable per object. For example, if one want 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 opposit). Concretely, the following code

```
jets      =
↪  df2array(df,['jet_pt','jet_eta','jet_phi','jet_mv2c10','jet_isbtagged_77'])
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, se 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')
```

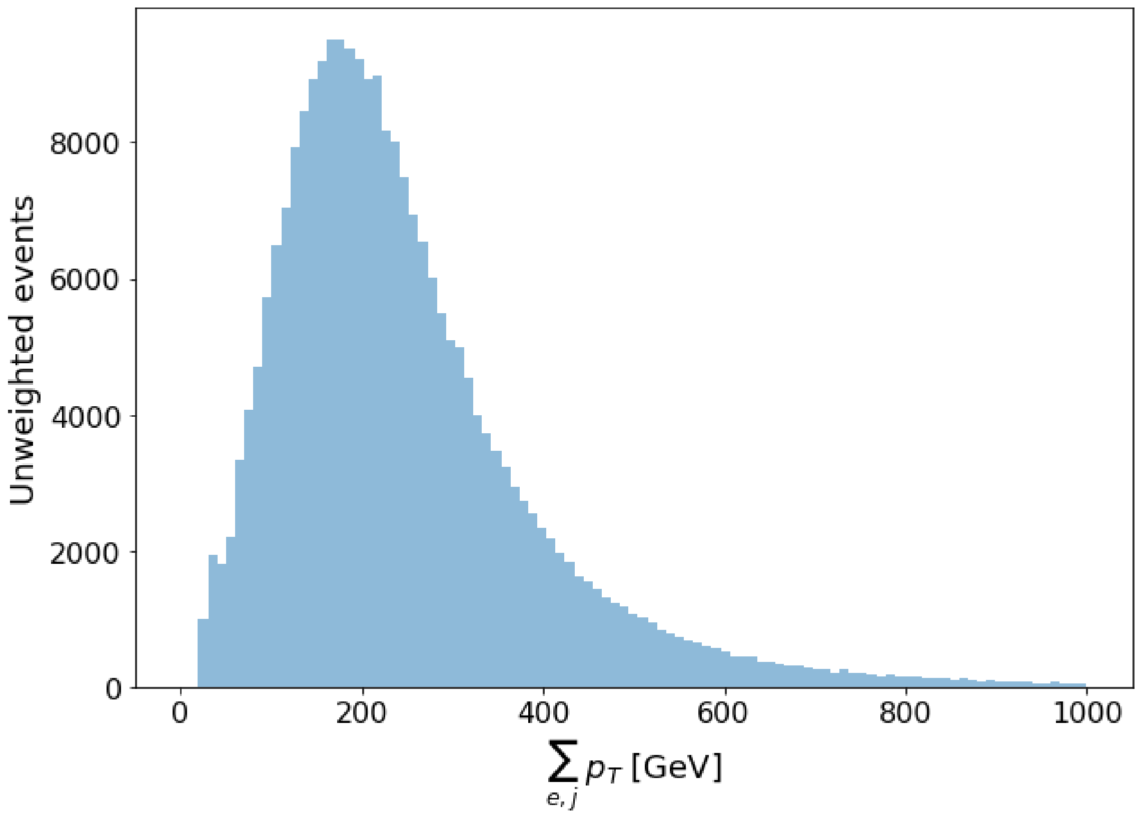

Figure 24

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 min$dR(e, j)$ 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')
```

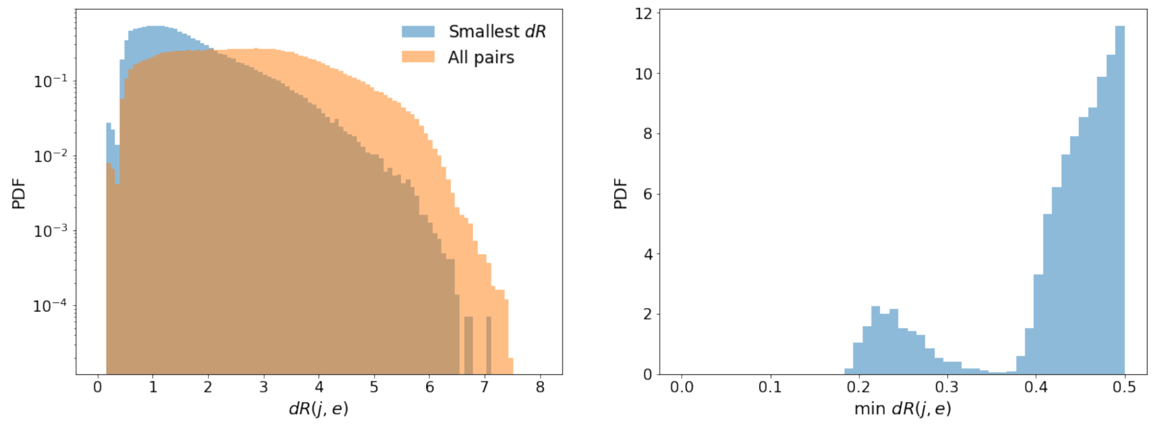

Figure 25

Getting now the index of the pair having the minimal dR using the command `np.argmin(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()
```

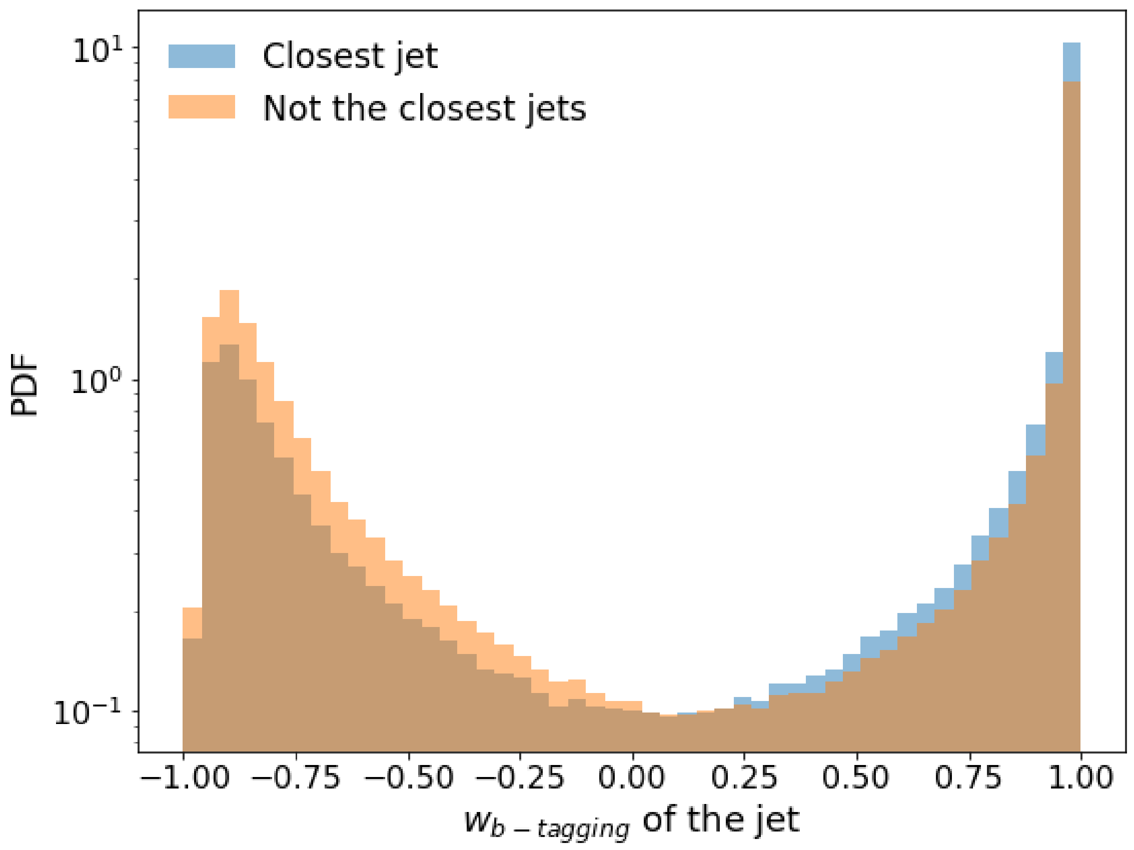

Figure 26

"'python 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'+'{:.2f}$'.format(cut), **style)