NumPy for High Energy Physics

How to manipulate high dimension data with NumPy

Romain Madar

Laboratoire de Physique de Clermont-Ferrand (UCA, CNRS/IN2P3) – FRANCE

Contact: romain.madar@clermont.in2p3.fr

March 13, 2023

Contents

Preamble

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.

Chapter 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 preferences and your purposes, python can be a very good option 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)
```

```
204 ms ± 27.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

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

```
188 ms ± 42.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
# Using vectorization
%timeit 1./a
```

```
116 µs ± 13.7 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        : [22 84 90]
a^2      : [ 484 7056 8100]
a/(1-a^a): [-4.41611606e-18  8.40000000e+01  9.00000000e+01]
cos(a)   : [-0.99996083 -0.6800235  -0.44807362]
exp(a)   : [3.58491285e+09 3.02507732e+36 1.22040329e+39]
```

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 = [47.6 53.6 61. ]
distances = [ 76.4852927   45.14421336 142.91955779 128.23805987  97.49871794]
```

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:
 [[-1.21022003 -1.11118338]
 [-0.85932207  1.10591019]
```

```
 [-0.74296716  1.82954158]]

points+d0:
 [[-0.21022003  2.88881662]
 [ 0.14067793  5.10591019]
 [ 0.25703284  5.82954158]]
```

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,)
```

```
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 syntax 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.

```
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 = [27 74  9 99 49 21 79 54 97 71]
a[2] = 9
a[-1] = 71
a[[1, 2, 5]] = [74  9 21]
```

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] =
 [[27 74]
 [ 9 99]
 [49 21]]
```

```
# 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 =
 [[[8 6]
  [5 8]
  [4 1]]

 [[7 2]
  [7 0]
  [9 2]]]
a[indices_big] =
 [[[97 79]
  [21 97]
```

```
  [49 74]]

 [[54  9]
  [54 27]
  [71  9]]]
```

Rule 3: There is a smart way to access sub-arrays with the syntax `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 syntax works also for n-dimensional array, where each dimension is sperated by a comma. 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              = [85 57 89 81 79 14 50 79  5  1]
from 0 to 1: a[:2]        = [85 57]
from 4 to end: a[4:]      = [79 14 50 79  5  1]
reverse order: a[::-1]    = [ 1  5 79 50 14 79 81 89 57 85]
all even elements: a[::2] = [85 89 79 50  5]
```

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

```
a =
[[[90 81 16]
  [87  2 33]]

 [[97 57 17]
  [21 39 87]]

 [[99 12 69]
  [18 23 30]]

 [[95 90 73]
  [76 99 50]]

 [[71 86 20]
  [67 50 63]]]
```

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] =
 [[90 81]
 [97 57]
 [99 12]
 [95 90]
 [71 86]]
```

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

```
a[:, ::-1, :] =
[[[87  2 33]
  [90 81 16]]

 [[21 39 87]
  [97 57 17]]

 [[18 23 30]
  [99 12 69]]

 [[76 99 50]
  [95 90 73]]

 [[67 50 63]
  [71 86 20]]]
```

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 =
[[-97 -59  35]
 [-92  72  74]
 [-60 -42 -35]
 [-93 -37 -60]
 [ 20  33  59]]

mask =
```

```
 [[False False  True]
 [False  True  True]
 [False False False]
 [False False False]
 [ True  True  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] =
 [35 72 74 20 33 59]

a*mask =
 [[ 0  0 35]
 [ 0 72 74]
 [ 0  0  0]
 [ 0  0  0]
 [20 33 59]]

a[~mask] =
 [-97 -59 -92 -60 -42 -35 -93 -37 -60]

a*~mask  =
 [[-97 -59   0]
 [-92   0   0]
 [-60 -42 -35]
 [-93 -37 -60]
 [  0   0   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 =
[[-97 -59  36]
 [-92  73  75]
 [-60 -42 -35]
 [-93 -37 -60]
 [ 21  34  60]]
```

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=
[[ 84  22  42]
 [-78  79 -55]
 [-68 -70  45]
 [-29  -4  91]
 [-72 -38  58]]

After: a=
[[  84   22   42]
 [6084   79 3025]
 [4624 4900   45]
 [ 841   16   91]
 [5184 1444   58]]
```

1.3 Few powerfull tools: matplotlib, pandas and scipy

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
```

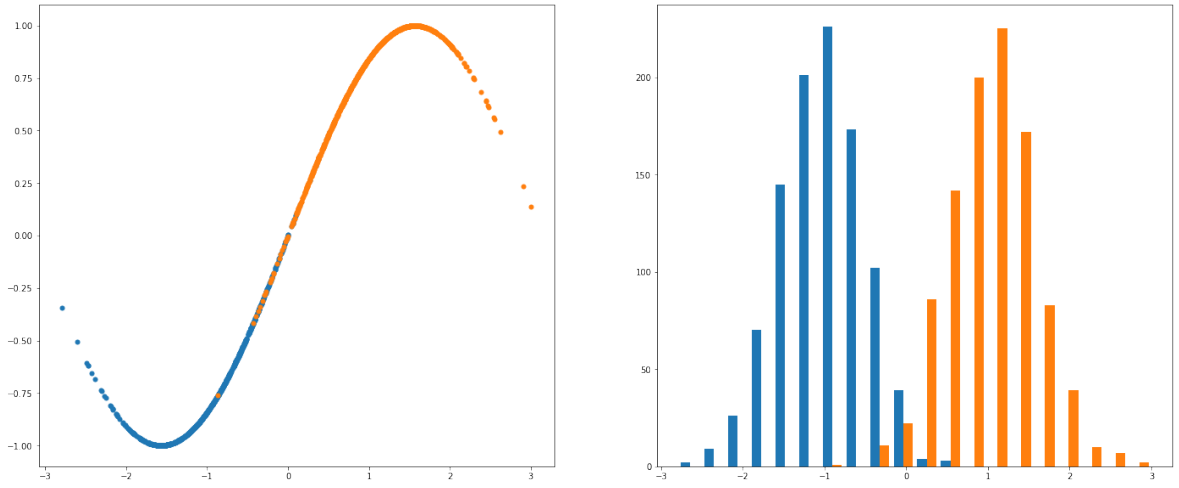
1.3.1.1 Example of 1D plots and histograms

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
plt.plot(x, y, marker='o', markersize=5, linewidth=0.0)
plt.subplot(122) # 122 means 1 line, 2 column, 2nd plot
plt.hist(x, bins=20);
```



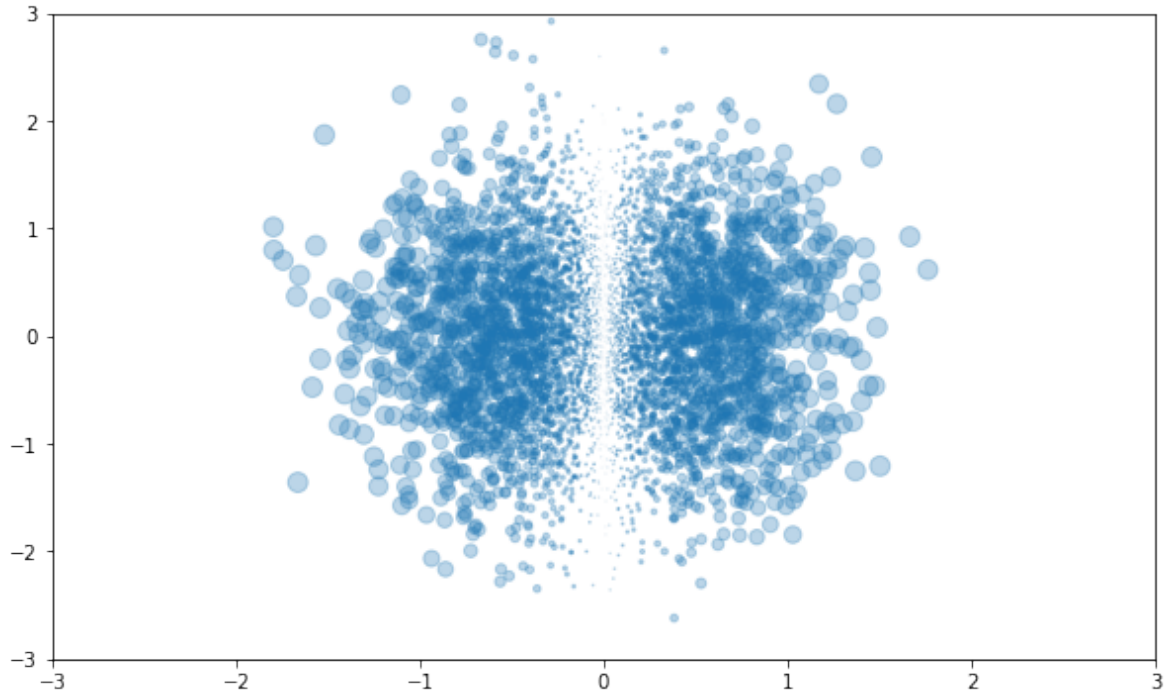
1.3.1.2 Example of 2D scatter plot

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 (σ_1, σ_2) = (0.5, 0.8) in a numpy array `points` of shape `points.shape=(5000,2)`. These two sets of numbers are then interepreted as (x, y) positions being loaded in two `(5000, 1)` arrays `x` and `y`:

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

```
plt.figure(figsize=(10,6))
plt.scatter(x, y, s=100*(np.sin(x))**2, marker='o', alpha=0.3)
plt.xlim(-3, 3)
plt.ylim(-3, 3);
```



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
```

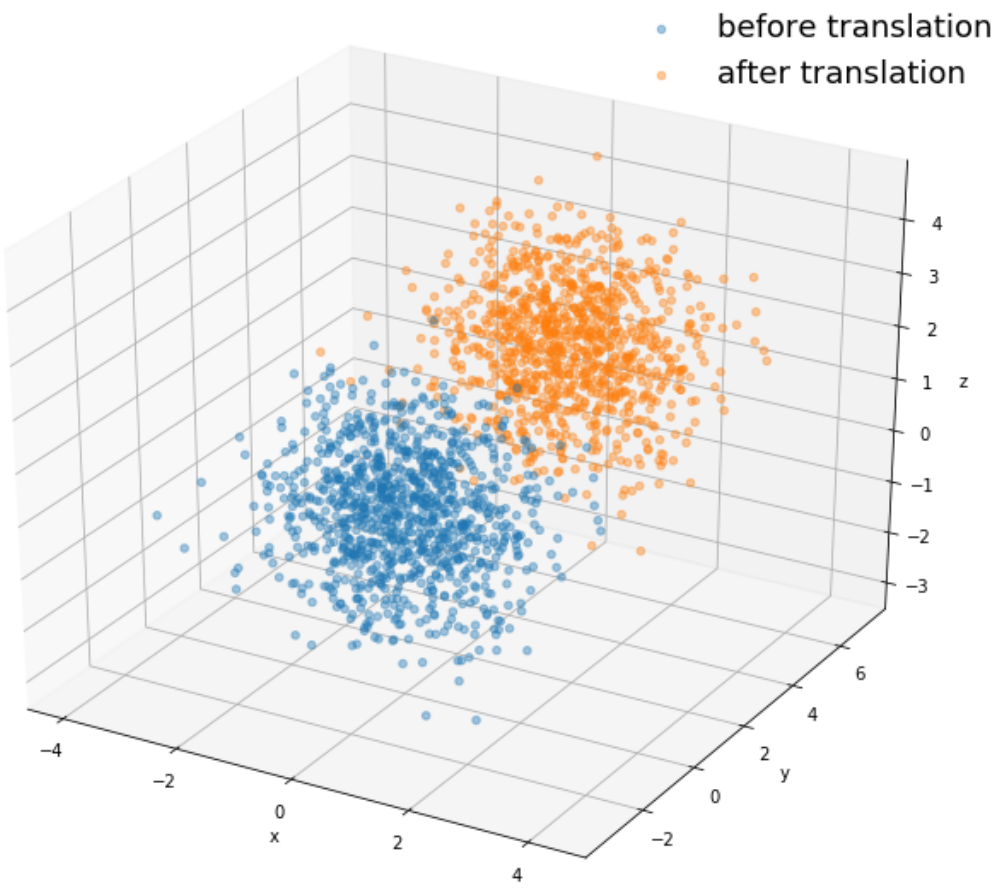
It is then easy to get back the spatial initial (*i.e.* before translation) and final (*i.e.* after translation) coordinates:

```
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
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);
```



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
```

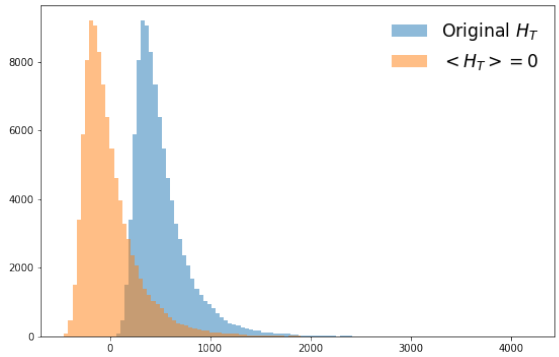
On of the nice feature of pandas is to be able to easily get numpy array, compute and store the result as a new column. For instance, it's a common practice in machine learning to *normalize* the input variables, *i.e.* transform them to have a mean of 0 and a variance of 1.0. The following example shows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

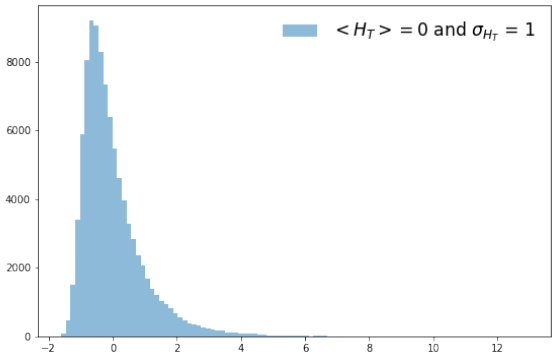
```
# Get a numpy arrays
ht = df['HT'].values

# 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):

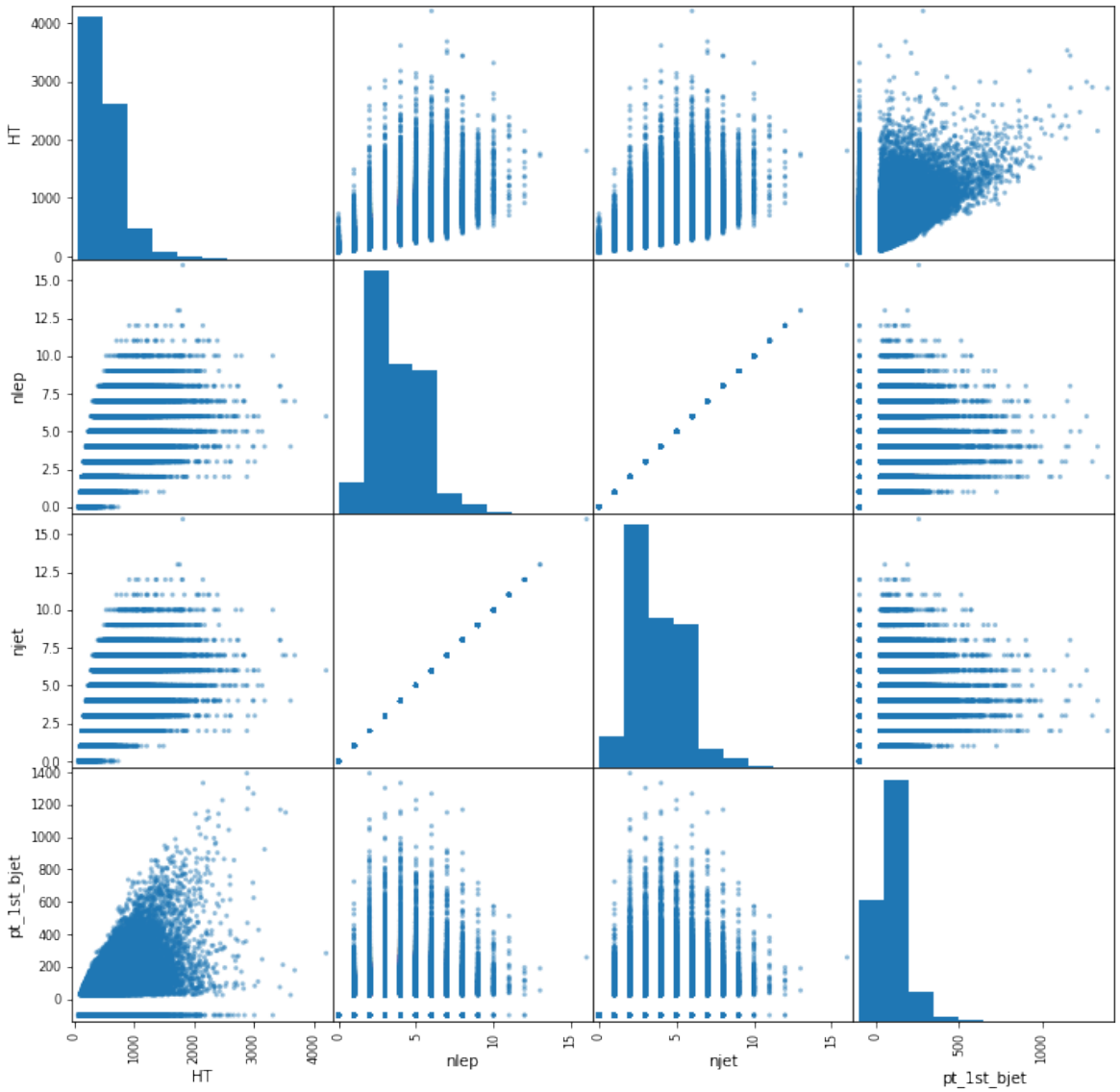
```
plt.figure(figsize=(20, 6))
plt.subplot(121)
plt.hist(df['HT'], bins=100, alpha=0.5, label='Original $H_T$')
plt.hist(df['HT_centered'], bins=100, alpha=0.5, label='$<H_T> = 0$')
plt.legend(frameon=False, fontsize='xx-large')
plt.subplot(122)
plt.hist(df['HT_normalized'], bins=100, alpha=0.5,
  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





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);
```



1.3.3 Mathematics, physics and engineering with scipy

The scipy project is python-based ecosystem of open-source software for mathematics, science, and engineering. In particular, the following core package are part of it: NumPy, matplotlib, pandas, scipy library (very quickly introduced here) and SymPy (symbolic calculations with mathematical expressions *a la* mathematica).

Obviously, there is no way to extensively present the scipy library in this short introduction, but one can quickly summarize few features and illustrate one with a concrete and useful example: fitting data points with a function. Among the main features, the SciPy library contains:

- Integration (`scipy.integrate`): integrals, differential equations, etc ...
- Optimization (`scipy.optimize`): minimization, fits, etc ...
- Interpolation (`scipy.interpolate`): smoothing methods, etc ...
- Fourier Transforms (`scipy.fftpack`): spectral analysis, etc ...
- Signal Processing (`scipy.signal`): transfer functions, filtering, etc ...

- Linear Algebra (`scipy.linalg`): matrix operation, diagonalisation, determinant, etc ...
- Statistics (`scipy.stats`): random number, probability density function, cumulative distribution, etc ...

```
from scipy import optimize
from scipy import stats
```

Let's now show how to perform a fit of data with error bar using one particular function of `scipy.optimize`. First, we need to generate some data where we choose 20 measurements, with some noise of ~30% and an combined uncertainty of an absolute 0.1 uncertainty and 10% relative uncertainy:

```
Npoints, Nsampling = 20, 1000
xcont = np.linspace(-5.0, 3.5, Nsampling)
x = np.linspace(-5, 3.0, Npoints)
y = 2*(np.sin(x/2)**2 + np.random.random(Npoints)*0.3)
dy = np.sqrt(0.10**2 + (0.10*y)**2)
```

Then we need to define functions with which we want to fit our data, for example a degree 1 polynoms. The syntax has to be `func(x, *pars)`:

```
def pol1(x, p0, p1):
    return p0 + x*p1
```

The following lines actually perform the fit and return both the optimal parameters and the covariances for the dgree 1 polynom:

```
p, cov = optimize.curve_fit(pol1, x, y, sigma=dy)
```

One can then generalize the procedure by plotting the result of the fit for polynoms of several degrees, after having plotted the data. This is a good way to compare different models for the same data. First, we define an arbitrary degree polynom `plo_func()` and we *vectorize* it using `np.vectorize` so that it can accept NumPy arrays:

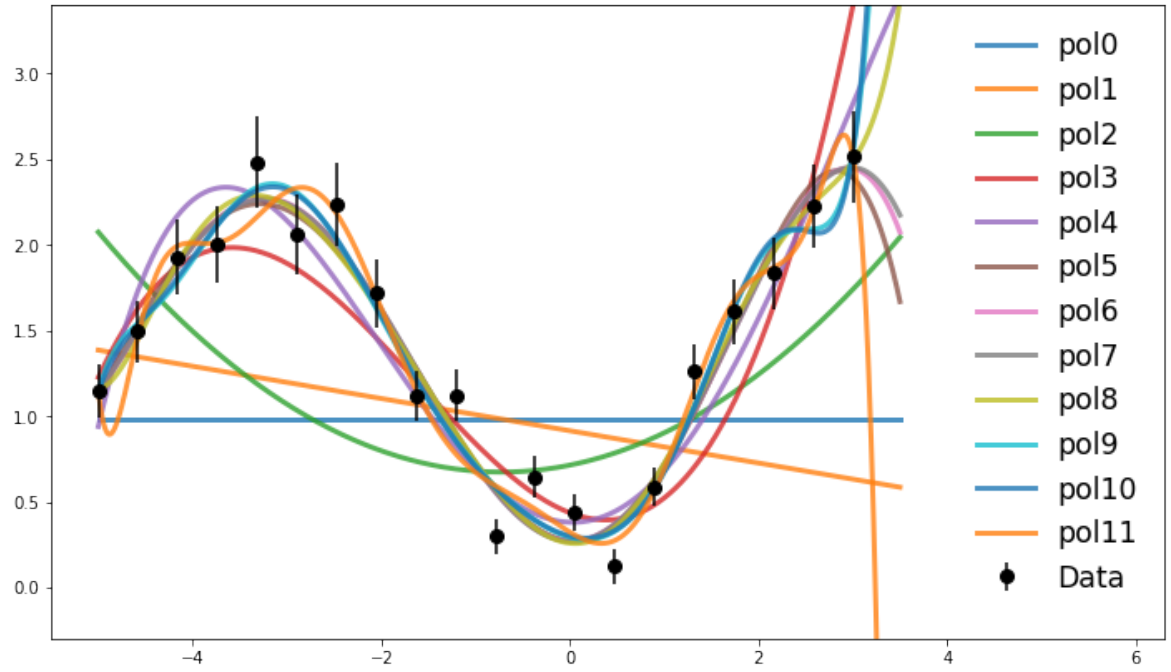
```
def pol_func(x, *coeff):
    a = np.array([coeff[i]*x**i for i in range(len(coeff))])
    return np.sum(a)

pol_func = np.vectorize(pol_func)
```

In the previous call for `optimize.curve_fit()`, we didn't use additional arguement. For this example, we need to specify at least the starting point of the parameters `p0` *because the number of paramter will be assessed using `len(p0)` (it's not known a priori since it is dynamically allocated)*. Other options can be specified, such as the miminum and maximum allowed values of parameters. Here is a wrapp-up function performing the fit for an arbitrary polynom degree:

```
def fit_polynom(degree):
    nPars = degree+1
    p0, pmin, pmax = [1.0]*nPars, [-10]*nPars, [10]*nPars
    fit_options = {'p0': p0, 'bounds': (pmin, pmax), 'check_finite': True}
    par, cov = optimize.curve_fit(pol_func, x, y, sigma=dy, **fit_options)
    return par, cov

degree_max = 12
```

The following code try every polynomial functions up to a degree `degree_max=`, perform the fit and overlay the the result for each together with the experimental data on the same figure:

```
# Figure for the result
fig = plt.figure(figsize=(12,7))

# Fitting & plotting
for d in np.arange(0, degree_max):
    par, cov = fit_polynom(d)
    plt.plot(xcont, pol_func(xcont, *par), label='pol{}'.format(d),
             linewidth=3, alpha=0.8)

# Plotting data
style = {'marker': 'o', 'color': 'black', 'markersize': 8,
         'linestyle': '', 'zorder': 10, 'label': 'Data'}
plt.errorbar(x, y, yerr=dy, **style)

# Plot cosmetics
plt.xlim(-5.5, 6.3)
plt.ylim(-0.3, 3.4)
plt.legend(frameon=False, fontsize='xx-large');
```



It is possible to quantify how well a given model explain the observations, computing what we call the *goodness of fit*. In a frequentist approach, this can be assessed by the fraction of pseudo-data coming from - in principle - repeating the exact same experiment, with to a worst agreement for a given model. The agreement can be quantified using $\chi^2 = \sum_{i=1}^{n} \frac{(y_i - f(x_i))^2}{\sigma_i^2}$ and its probability density function (PDF) directly gives access to the fraction of "worst pseudo-data" (by integrating the PDF from χ^2 to ∞). More precisely, one can use the cumulative distribution function (CDF) of χ^2 computed with n degrees of freedom, for instance `Npoins`, *i.e.* `len(x)`. More details can be found, for examble, in the statistics review of the Particle Data Group. The following two functions allow to compute the goodness of fit:

```
def get_chi2_nDOF(y, dy, yfit):
    r = (y-yfit)/dy
    return  np.sum(r**2), len(y)

def get_pvalue(chi2, nDOF):
    return 1-stats.chi2.cdf(chi2, df=nDOF)
```

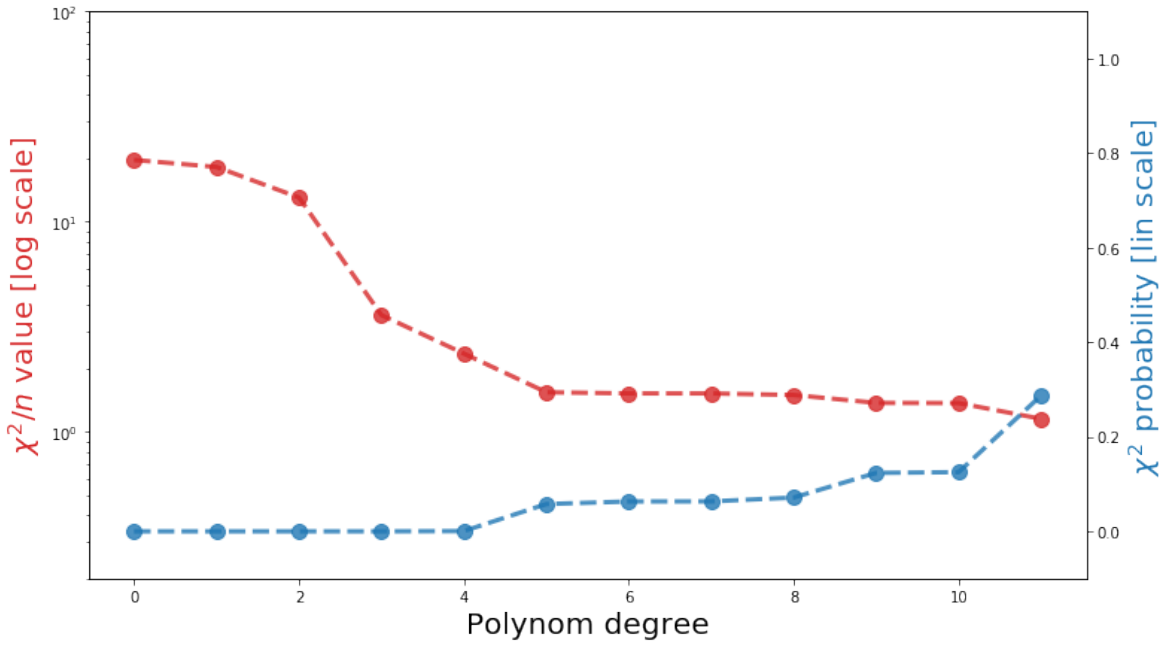
We can now perform all these fits and extract the goodness of fit (χ^2 and p-value) for each model:

```
# Fitting and getting p-value
degree, chiSquare, pvalue = [], [], []
for d in np.arange(degree_max):
    par, cov = fit_polynom(d)
    c2, n = get_chi2_nDOF(y, dy, pol_func(x, *par))
    degree.append(d), chiSquare.append(c2), pvalue.append(get_pvalue(c2, n))
```

The following piece of code plot both the χ^2 and the p-value versus the degree of the polynom using two different y-axis. This gives another way to use matplolib by defining explicit object such as `ax` and `fig` and

call methods on those (called *stateless appraoch*), instead of using function on `plt` (called *stateful approach*). For more details on these different approaches, see this RealPython post.

```
# Plotting the result with 2 different axis
fig, ax1 = plt.subplots(figsize=(12,7))
ax1.set_xlabel('Polynom degree', fontsize=20)
style = {'marker': 'o', 'markersize': 10, 'alpha': 0.8,
         'linestyle': '--', 'linewidth': 3}

# Plot chi2/n
ax1.semilogy(degree, np.array(chiSquare)/Npoints, color='tab:red', **style)
ax1.set_ylim(0.2, 100)
ax1.set_ylabel('$\chi^2/n$ value [log scale]', color='tab:red', fontsize=20)

# Plot p-values
ax2 = ax1.twinx()
ax2.plot(degree, pvalue, color='tab:blue', **style)
ax2.set_ylim(-0.1, 1.1)
ax2.set_ylabel('$\chi^2$ probability [lin scale]', color='tab:blue', fontsize=20);
```



This is also possible to know whether two models give a similar description or if one model is better than the other. This is called the *F-test*, based on the residual sum of square $RSS \equiv \sum(y_i - f(x_i))^2$ (RSS). For a fit of n data points with a model M1 defined by p_1 parameters with RSS_1 and a model M2 defined by p_2 parameters with RSS_2 *with* $p_2 > p_1$, the F-test is defined by:

$$F(M1, M2) = \frac{\left(\frac{RSS_1 - RSS_2}{p_2 - p_1}\right)}{\left(\frac{RSS_2}{n - p_2}\right)}$$

If the p-value of F is close to 1.0, it means that the two model are equally compatible with data and there

is no indication that a choice should be made. In that case, one would favour the simplest model with less paramters. If the p-value of F is close to 0, then the two compared model are actually different and one has to select one.

Let's first define the RSS function and compute it over all the polynoms:

```
def get_RSS(y, yfit):
    return np.sum((y-yfit)**2)

RSS = []
for d in np.arange(degree_max):
    par, cov = fit_polynom(d)
    RSS.append(get_RSS(y=y, yfit=pol_func(x, *par)))
```

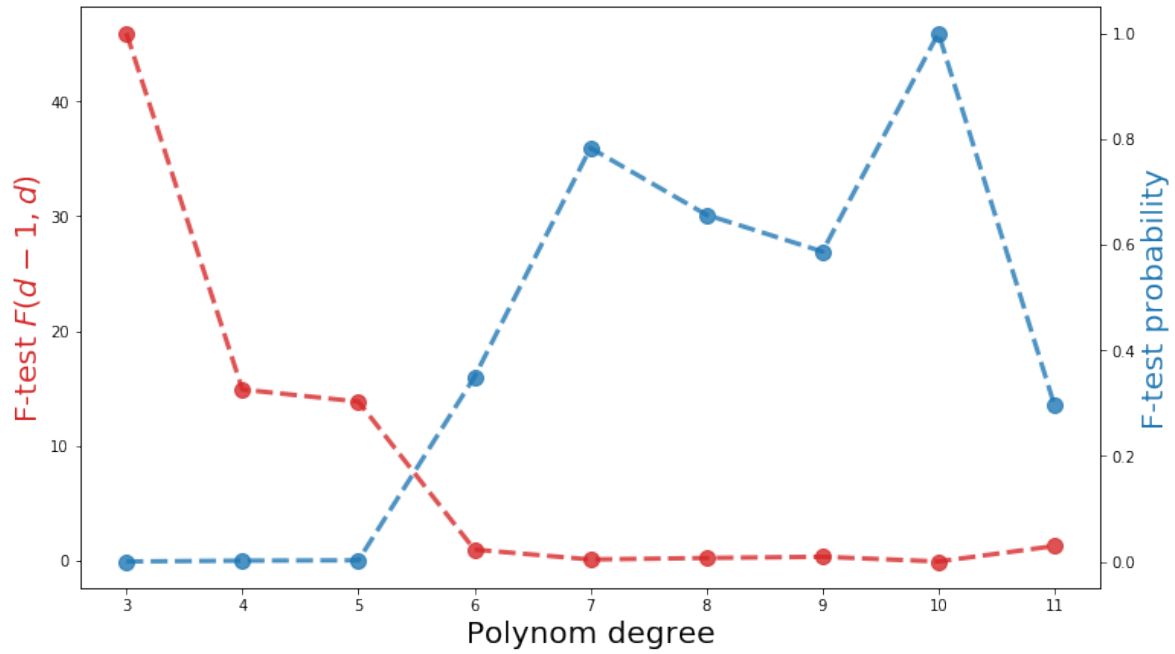
Then, one can define a function to actually compare two degrees where d_2 must be larger than d_1:

```
def compare_pol_pq(d1, d2):

    # Sanity checks
    if d1 not in degree or d2 not in degree:
        raise NameError('Degree no supported')
    if d2<d1:
        d2, d1 = d1, d2

    # Extract M1 and M2 numbers
    RSS1, p1 = RSS[d1], d1+1
    RSS2, p2 = RSS[d2], d2+1

    # F-value & p-value
    Fval = (RSS1-RSS2)/(p2-p1) * (Npoints-p2)/RSS2
    pval = 1-stats.f.cdf(x=Fval, dfn=p2-p1, dfd=Npoints-p2)

    return Fval, pval
```

Plot the F-test values and its p-values for the comparison between $d-1$ and d starting from $d=3$. This plot shows that from $d=6$, there is a faire compatibility between the models (at worst 15%):

```
# Computing F-test comparisons starting from degree=4
dmin, compare = 3, compare_pol_pq
dinf, dsup = np.arange(dmin-1, degree_max), np.arange(dmin, degree_max)
result = np.array([compare(d1, d2) for d1, d2 in zip(dinf, dsup)])
Ftest, pval = result[:, 0], result[:, 1]

# Plotting figure and syle the results
fig, ax1 = plt.subplots(figsize=(12,7))
ax1.set_xlabel('Polynom degree', fontsize=20)
style = {'marker': 'o', 'markersize': 10, 'alpha': 0.8,
         'linestyle': '--', 'linewidth': 3}
```

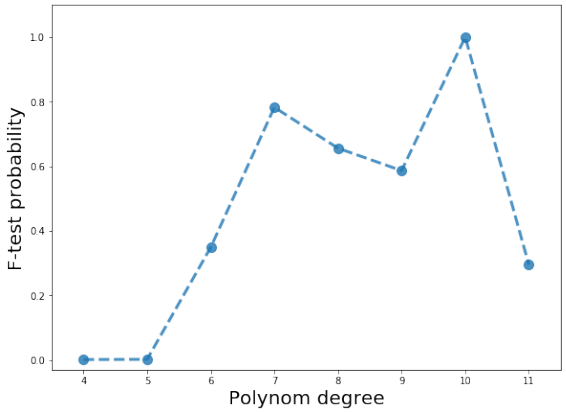
```
# Plot of F-test values
ax1.plot(dsup, Ftest, color='tab:red', **style)
ax1.set_ylabel('F-test $F(d-1,d)$', color='tab:red', fontsize=20)

# Plot of p-values
ax2 = ax1.twinx()
ax2.plot(dsup, pval, color='tab:blue', **style)
ax2.set_ylabel('F-test probability', color='tab:blue', fontsize=20);
```

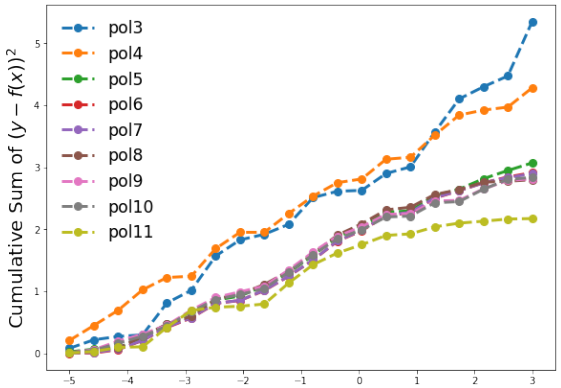


In order to effectively check by eye that the F-test give a sensible information, one can plot the cumulative sum of fit residus $(y - f(x))^2$ and the F-test probability side-by-side:

```
# Plotting figure and syle the results
plt.figure(figsize=(21,7))
style = {'marker': 'o', 'markersize': 10, 'alpha': 0.8,
         'linestyle': '--', 'linewidth': 3}

# Plot of F-test values
plt.subplot('121')
plt.plot(dsup[1:], pval[1:], **style)
plt.ylabel('F-test probability' , fontsize=20);
plt.xlabel('Polynom degree', fontsize=20)
plt.xlim(3.5, 11.5)
plt.ylim(-0.03, 1.1)

# Putting the cumsum of fit residus aside
plt.subplot('122')
style = {'marker': 'o', 'markersize': 8,
         'linestyle': '--', 'linewidth': 3}
```

```
for d in np.arange(3, degree_max):
    par, cov = fit_polynom(d)
    RSScum = np.cumsum(np.abs(y-pol_func(x, *par)))
    plt.plot(x, RSScum, label='pol{}'.format(d), **style)
plt.legend(frameon=False, fontsize='xx-large');
plt.ylabel('Cumulative Sum of $(y-f(x))^2$', fontsize=20);
```





We can then see on these 2 plots that indeed, comparing low degrees polynoms (like $d = 4$ and $d = 3$) leads to q large difference (and thus a low F-test probability), while comparing high degrees polynoms (*e.g.* $d = 6$ and $d = 5$) leads to very similar prediction meaning a sizable F-test probability. On sees that the probability that $d = 10$ is similar to $d = 10$ is quite low, which can be probably understood by a quite different RSS (last point on the right plot above for `pol11` and `pol10`).

Chapter 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])
```

```
[[[0.15624688 0.65075903 0.73146179]
  [0.73724685 0.48307419 0.77974699]
  [0.20971887 0.59327557 0.6531156 ]
  [0.35159907 0.52204891 0.25537105]
  [0.48437309 0.82028797 0.30840494]
  [0.02253355 0.74355187 0.49186388]
  [0.89405241 0.17862992 0.82465271]
  [0.89267253 0.72194804 0.58563866]
  [0.80358019 0.36464842 0.67233442]
  [0.25564683 0.6235393  0.74733499]]

 [[0.91650213 0.05817777 0.01940356]
  [0.70698068 0.85873894 0.84390599]
  [0.86004332 0.36252308 0.98642545]
  [0.40568495 0.01213849 0.67142505]
  [0.42802106 0.34523561 0.97850495]
  [0.41930931 0.59545754 0.11757192]
  [0.03047176 0.5600329  0.66797962]
  [0.6613735  0.9528616  0.23113316]
  [0.56349177 0.31426095 0.43091974]
  [0.25061127 0.91620815 0.32174233]]]
```

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)
```

```
44.7 ms ± 612 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

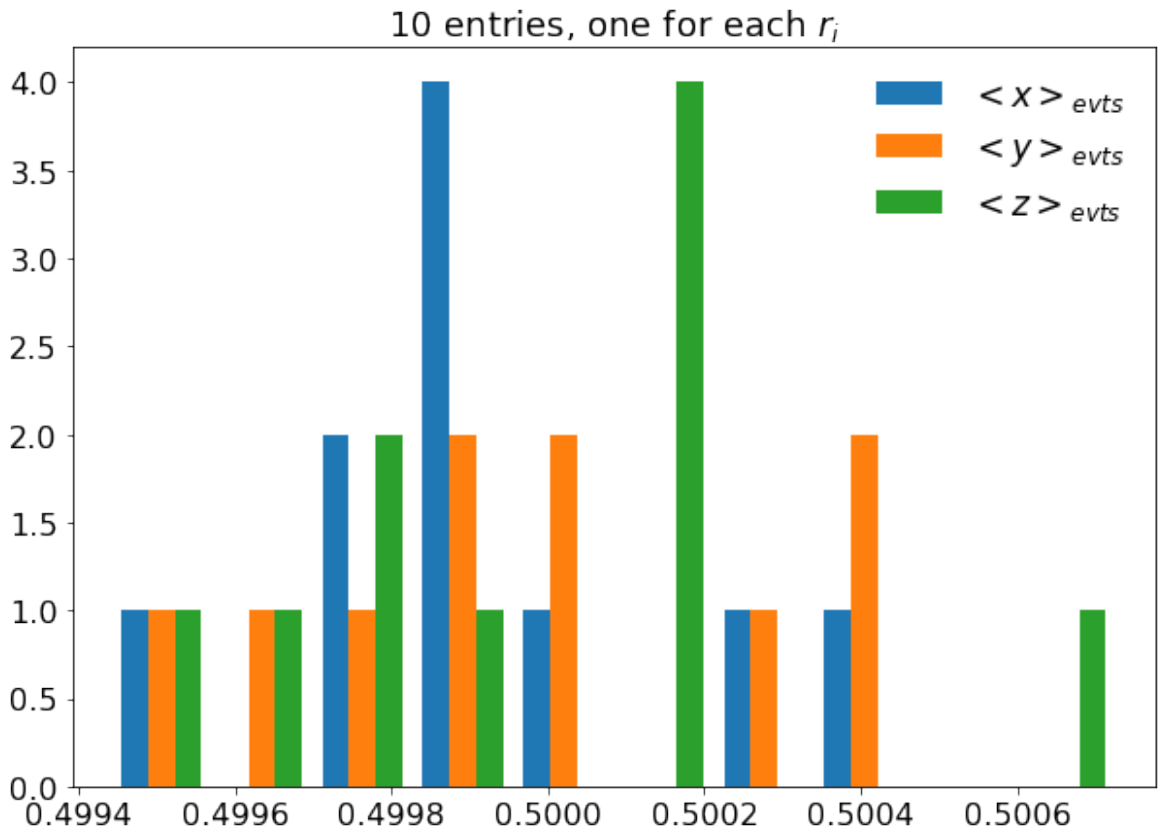
While it takes 10 times longer for a *single mean* over a million number with an explicit loop, so **the gain of vectorization is a factor 300**:

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

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

```
415 ms ± 6.02 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):

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



2.2.2 Mean over the 10 vectors (axis=1)

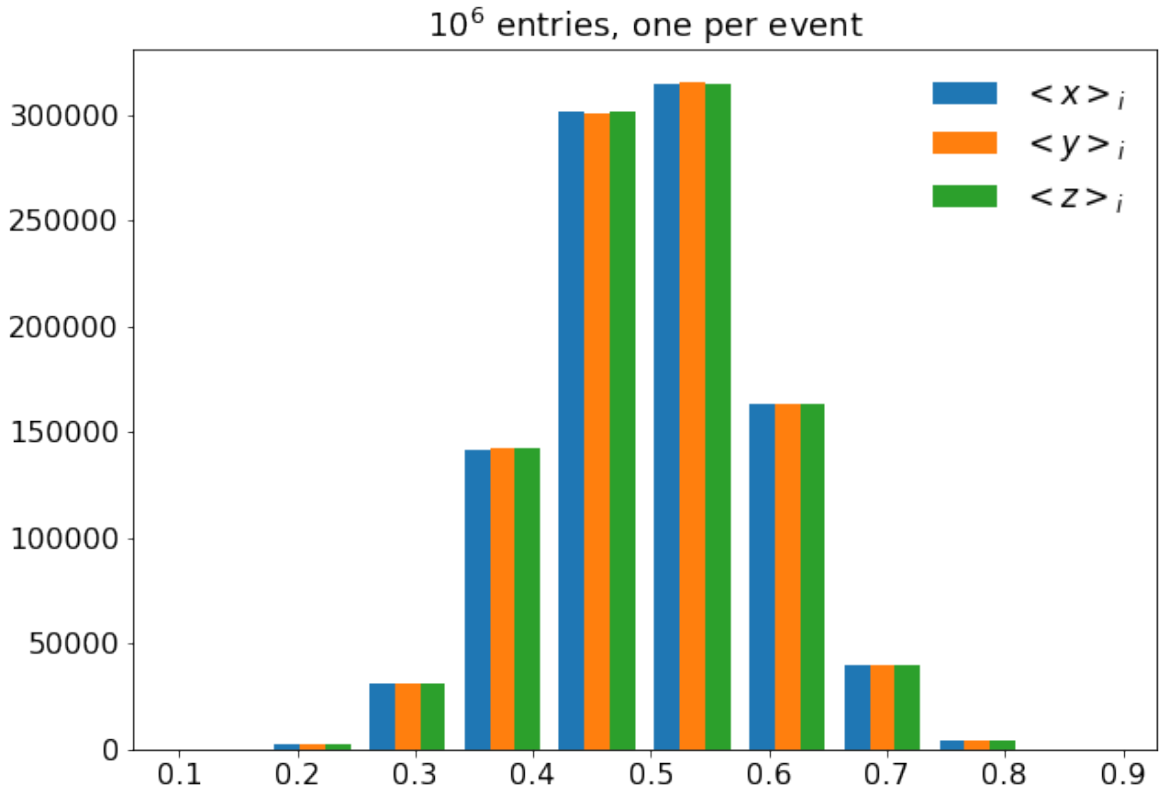
This one will compute the average over the 10 vectors, for each observations, reducing into a `(1000000, 3)` shape array, as seen below. This is 3D barycenter of each observation.

```
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:

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



2.2.3 Mean over the coordintates (axis=2)

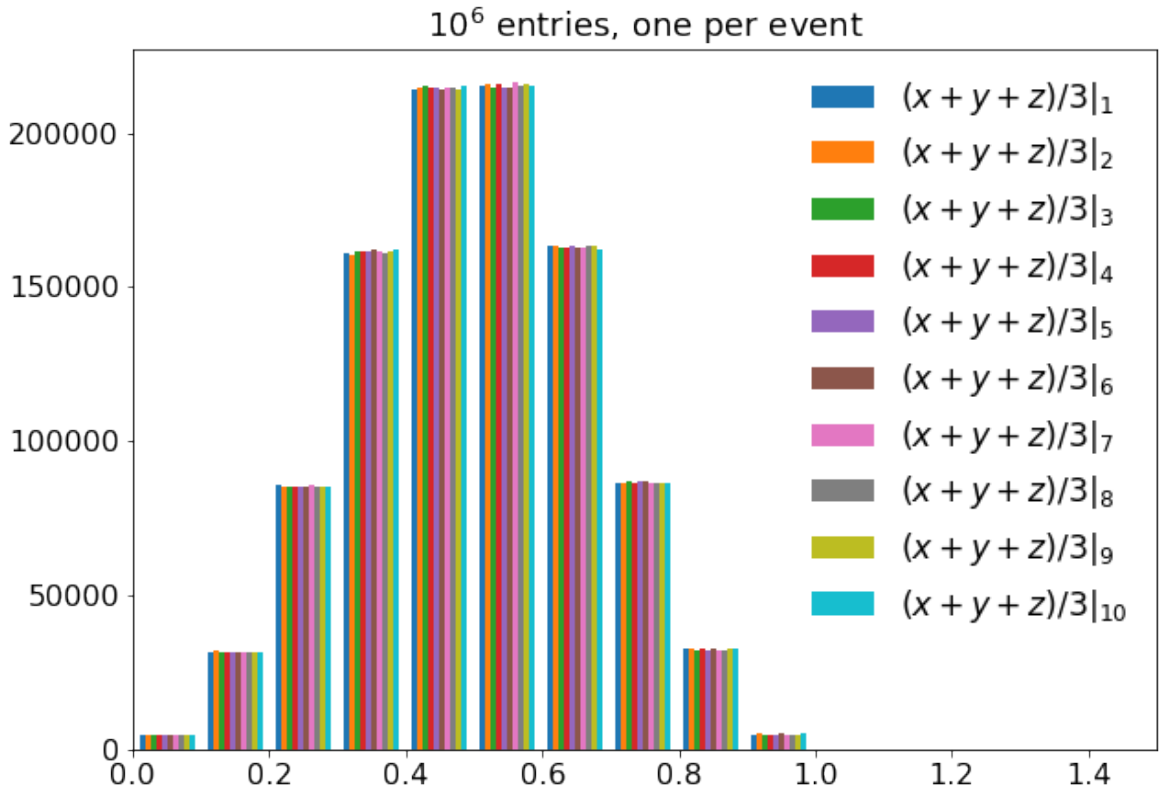
This directly computes the average over the three coordinates $(x + y + z)/3$ for each vector of each event, resulting in 10 values per event:

```
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)]
plt.hist(m2, label=names)
plt.title('$10^6$ entries, one per event')
plt.xlim(0, 1.5)
plt.legend();
```



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])
```

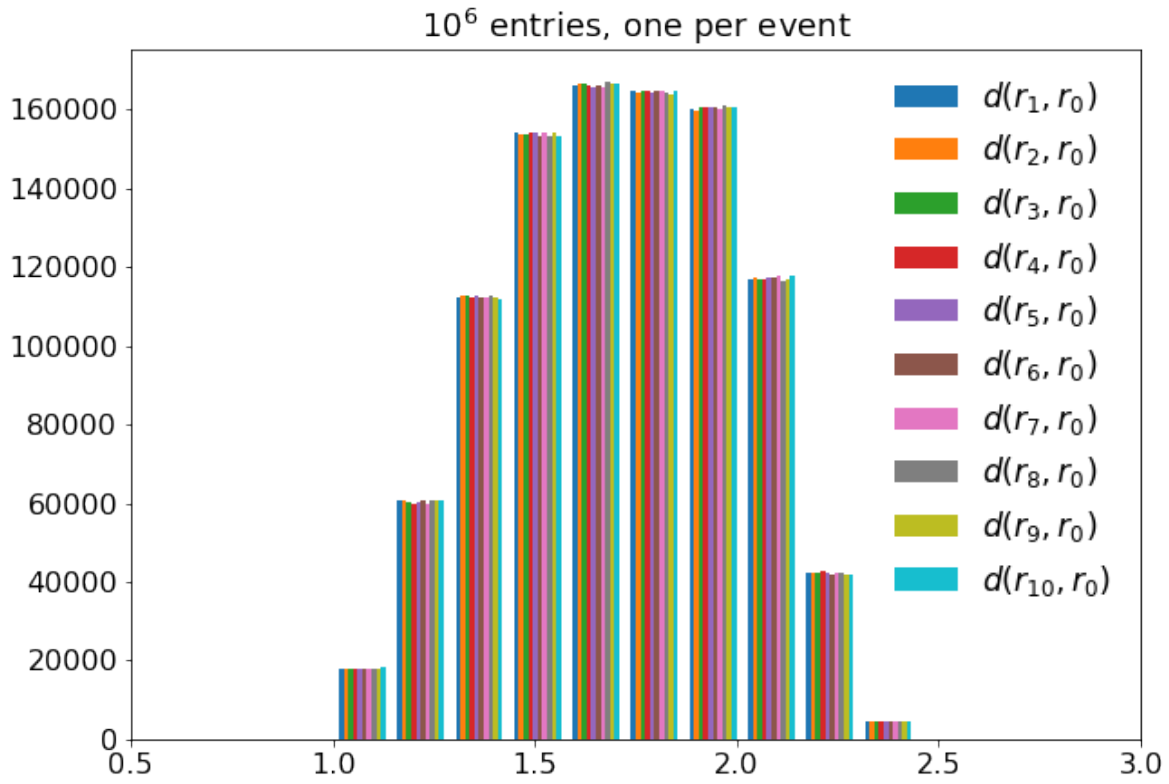
and compute the distance to this new origin for all points, using `**2` to square all numbers, perform the sum over the coordinate (`axis=2`) and square-root everything with `**0.5`:

```
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)]
plt.hist(d, label=names)
plt.title('$10^6$ entries, one per event')
plt.xlim(0.5, 3)
plt.legend();
```



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:

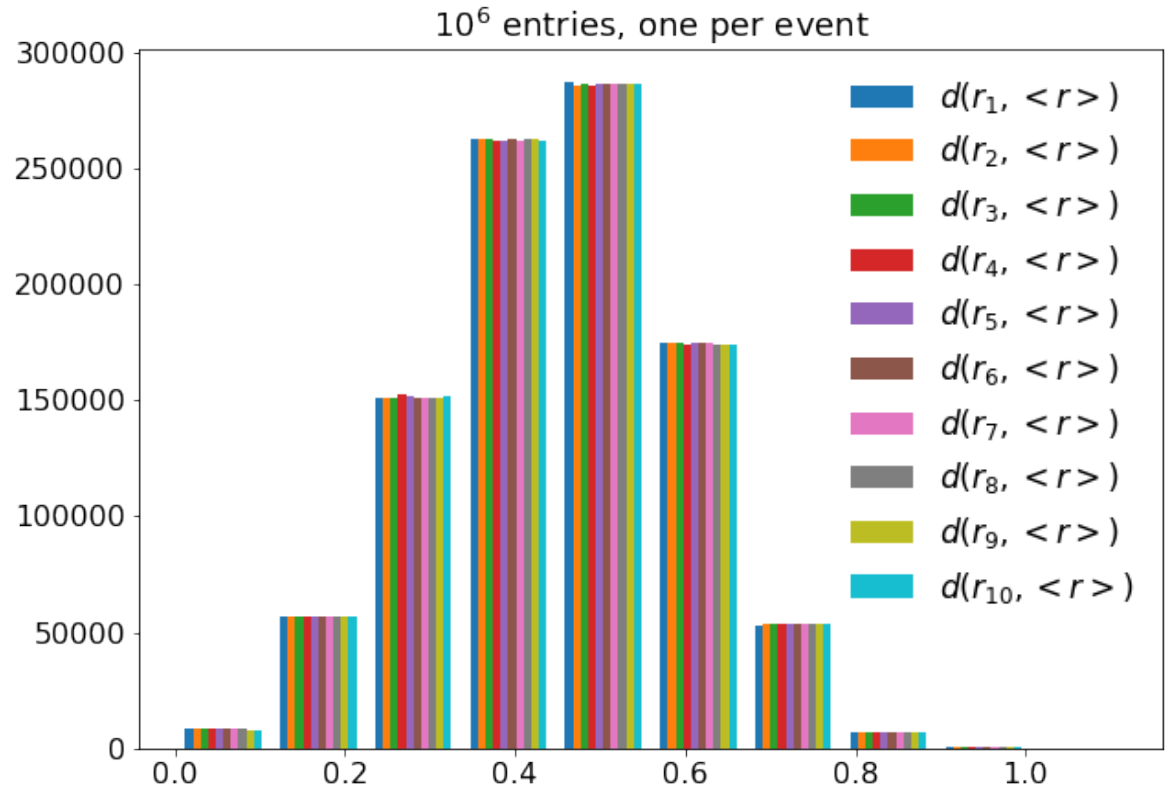
```
r_mean_3d = r_mean[:, np.newaxis, :]
```

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)]
plt.hist(d_to_mean, label=names)
plt.title('$10^6$ entries, one per event')
plt.legend();
```



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)
```

```
[[[0.15624688 0.65075903 0.73146179]
  [0.73724685 0.48307419 0.77974699]
  [0.20971887 0.59327557 0.6531156 ]
  [0.35159907 0.52204891 0.25537105]
  [0.48437309 0.82028797 0.30840494]]

 [[0.91650213 0.05817777 0.01940356]
  [0.70698068 0.85873894 0.84390599]
  [0.86004332 0.36252308 0.98642545]
  [0.40568495 0.01213849 0.67142505]
  [0.42802106 0.34523561 0.97850495]]]
```

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])
```

```
[[[0.15624688 0.65075903 0.73146179]
  [0.73724685 0.48307419 0.77974699]]

 [[0.15624688 0.65075903 0.73146179]
  [0.20971887 0.59327557 0.6531156 ]]

 [[0.15624688 0.65075903 0.73146179]
  [0.35159907 0.52204891 0.25537105]]

 [[0.15624688 0.65075903 0.73146179]
  [0.48437309 0.82028797 0.30840494]]

 [[0.73724685 0.48307419 0.77974699]
  [0.20971887 0.59327557 0.6531156 ]]

 [[0.73724685 0.48307419 0.77974699]
  [0.35159907 0.52204891 0.25537105]]

 [[0.73724685 0.48307419 0.77974699]
  [0.48437309 0.82028797 0.30840494]]

 [[0.20971887 0.59327557 0.6531156 ]
  [0.35159907 0.52204891 0.25537105]]

 [[0.20971887 0.59327557 0.6531156 ]
  [0.48437309 0.82028797 0.30840494]]

 [[0.35159907 0.52204891 0.25537105]
  [0.48437309 0.82028797 0.30840494]]]
```

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])
```

```
[[[0.15624688 0.65075903 0.73146179]
  [0.73724685 0.48307419 0.77974699]]

 [[0.15624688 0.65075903 0.73146179]
  [0.20971887 0.59327557 0.6531156 ]]

 [[0.15624688 0.65075903 0.73146179]
  [0.35159907 0.52204891 0.25537105]]

 [[0.15624688 0.65075903 0.73146179]
  [0.48437309 0.82028797 0.30840494]]

 [[0.73724685 0.48307419 0.77974699]
  [0.20971887 0.59327557 0.6531156 ]]

 [[0.73724685 0.48307419 0.77974699]
  [0.35159907 0.52204891 0.25537105]]

 [[0.73724685 0.48307419 0.77974699]
  [0.48437309 0.82028797 0.30840494]]

 [[0.20971887 0.59327557 0.6531156 ]
  [0.35159907 0.52204891 0.25537105]]

 [[0.20971887 0.59327557 0.6531156 ]
  [0.48437309 0.82028797 0.30840494]]

 [[0.35159907 0.52204891 0.25537105]
  [0.48437309 0.82028797 0.30840494]]]
```

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)
```

```
935 ms ± 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)
```

```
949 ms ± 6.89 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)
```

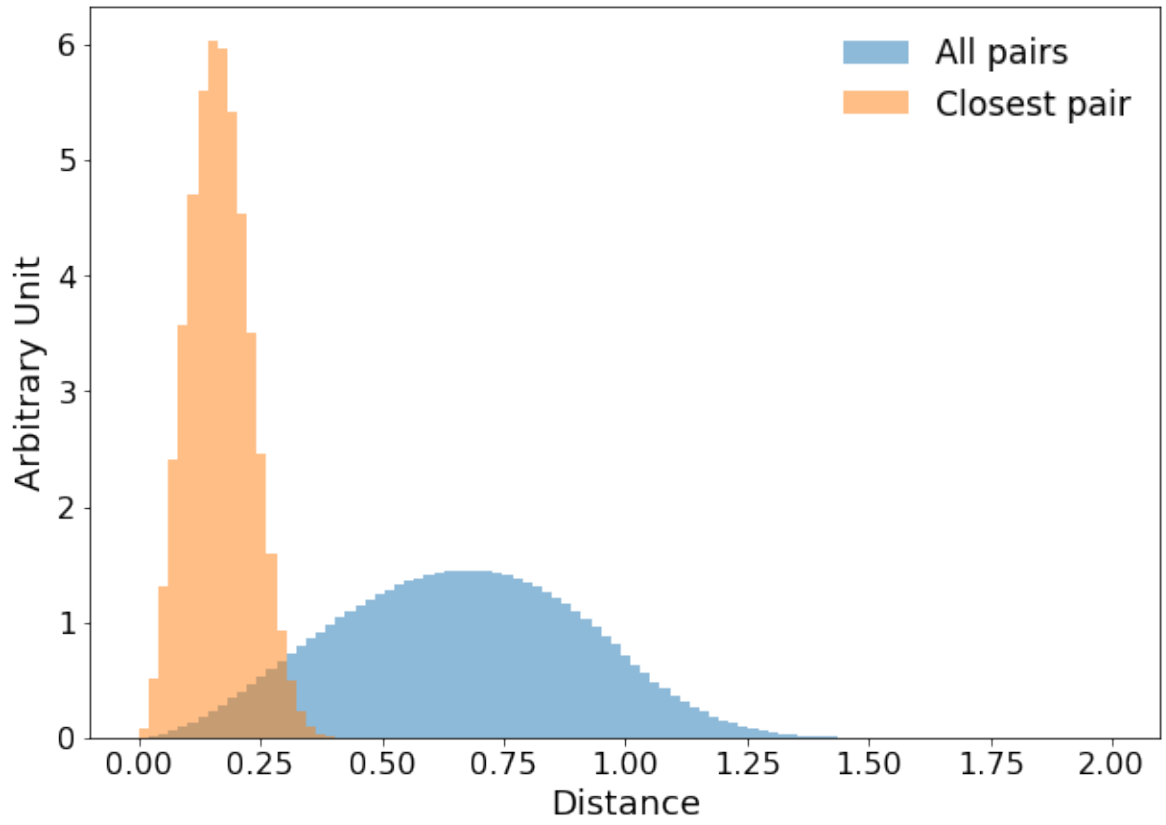
And let's compare the performance on 0.2 million observations:

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

```
193 ms ± 1.02 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
619 ms ± 7.06 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,
}

plt.hist(distances.flatten(), label='All pairs' ,**plot_style)
plt.hist(smallest_distance, label='Closest pair', **plot_style)
plt.xlabel('Distance')
plt.ylabel('Arbitrary Unit')
plt.legend();
```



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

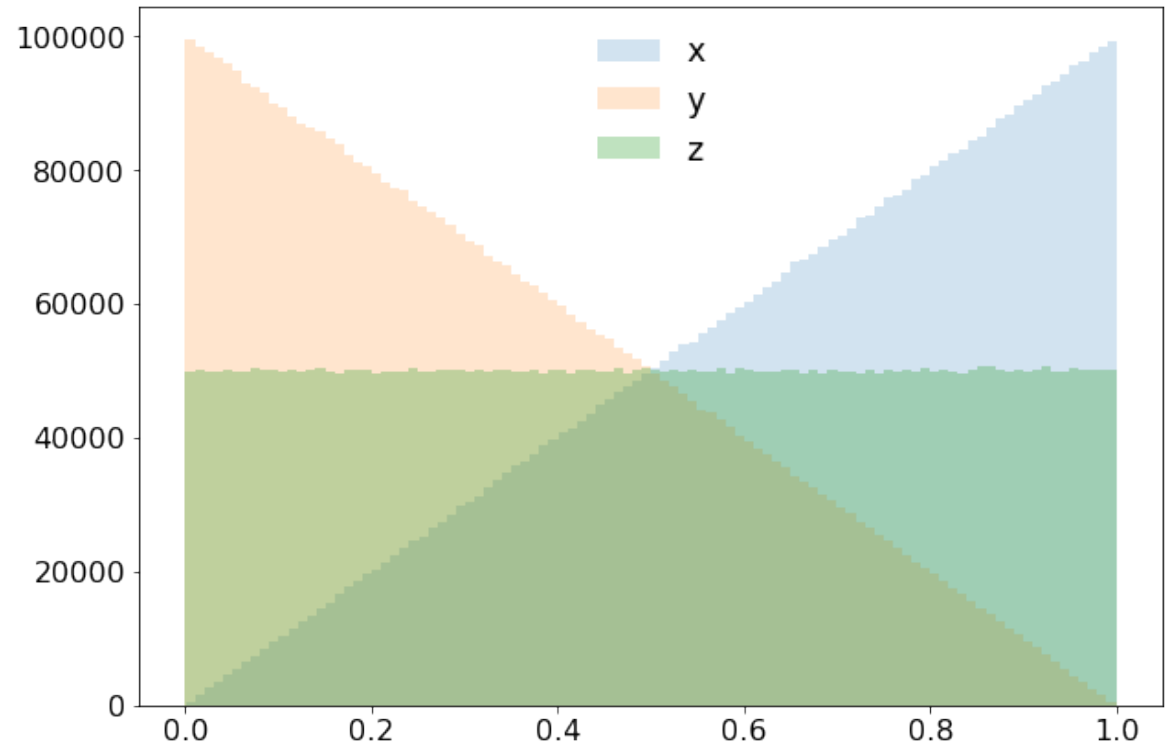
We will use the numpy masking feature described in the first chapter, defining a index of boolean based on `x` and `y` arrays:

```
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.

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



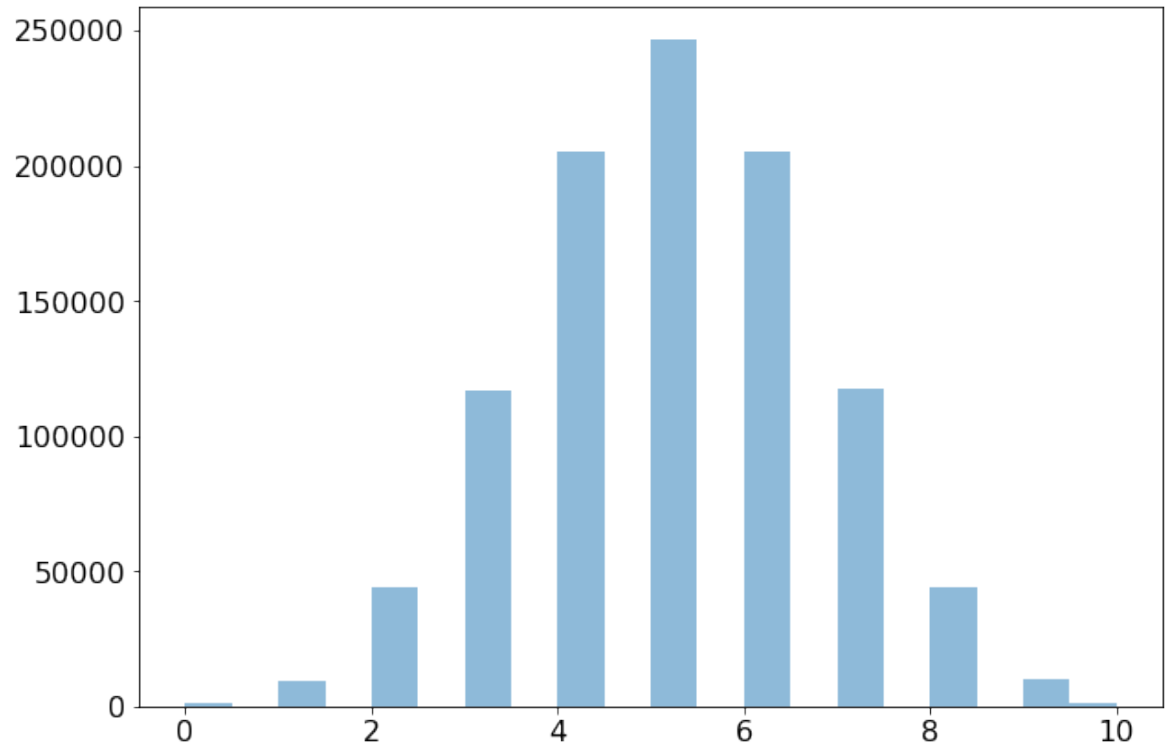
If we want to better understand how this selection affect our data, one might want to count the number of points per event satisfying this selection, using `np.count_nonzero()` on the boolean array along the axis representing the 10 vectors `axis=1`:

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

```
(1000000,)
```

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

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



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):

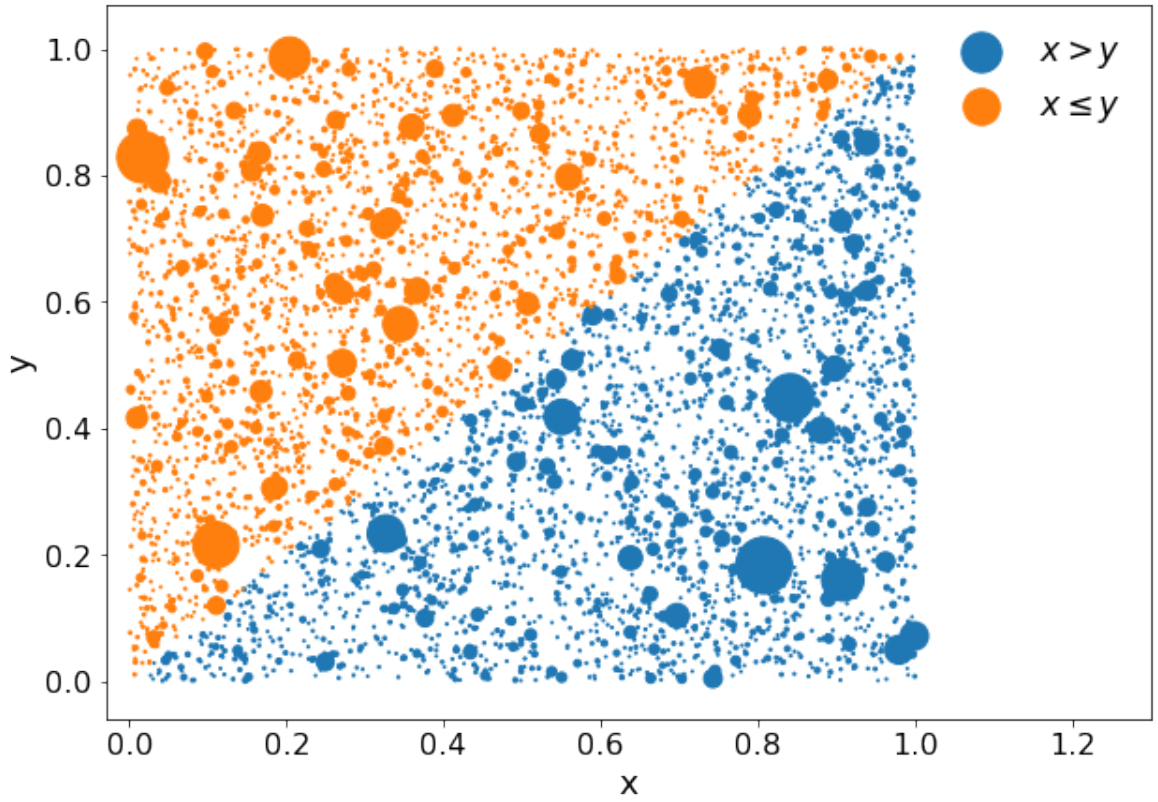
```
sx, sy, sz = x[0:500, ...], y[0:500, ...], z[0:500, ...]
```

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:

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



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: 0.00100
```

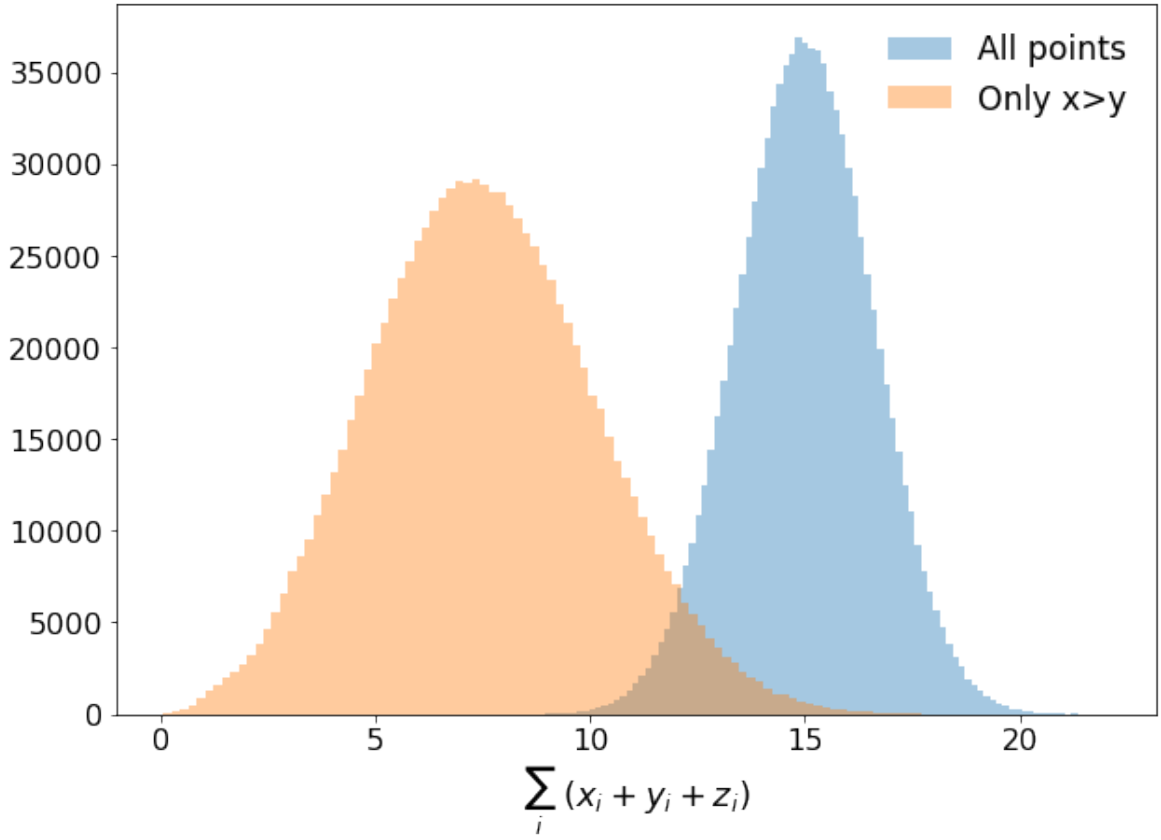
In a more general manner, we should use *masked arrays* which completely remove the masked elements from any computations:

```
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 $x \leq y$):

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



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])
```

```
[[0.15624688 0.65075903 0.73146179]
 [        nan         nan         nan]
 [0.20971887 0.59327557 0.6531156 ]
 [0.35159907 0.52204891 0.25537105]
 [0.48437309 0.82028797 0.30840494]
 [0.02253355 0.74355187 0.49186388]
 [        nan         nan         nan]
 [        nan         nan         nan]
 [        nan         nan         nan]
 [0.25564683 0.6235393  0.74733499]]
```

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

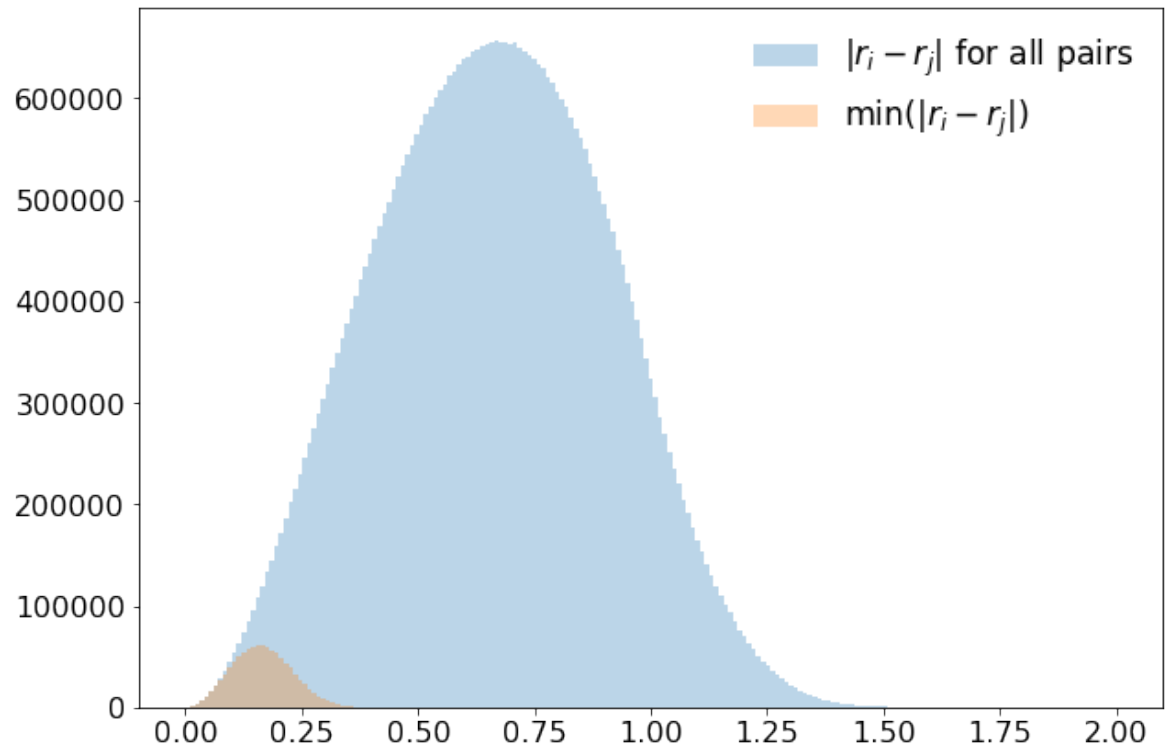
```
selected_pairs = combs_nd(selected_r, n=2, axis=1)
```

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}
plt.hist(dp.flatten(), label='$|r_i-r_j|$ for all pairs', **plot_style)
plt.hist(np.min(dp, axis=1), label='min$(|r_i-r_j|)$', **plot_style)
plt.legend();
```



2.6 Some 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.

Chapter 3

Collider data analysis and limitations

The goal of this chapter is to apply the HEP-like computation using simulated collider data. This will allows to work in more realistic cases and go a bit further. Indeed, NumPy has some fundamental limitations that we propose to put in evidence as a first step. In a seconde step, we will try to work-around these limitations by writting purely vectorized functions performing operations which would normally require a *for* loop.

Note: the module uproot need to be installed, which can be done by running this command into a cell inside this notebook:

```
!pip install uproot
```

```
# 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
import uproot
```

```
# 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

The very first step is to load data of a TTree in python, using uproot:

```
rootfile = uproot.open("collisions.root")
uptree = rootfile['event_tree']
```

Then, there are different ways to export this TTree into a NumPy-related object. More details can be found in the uproot documentation but we will just use the `uptree.arrays()` method which returns a dictionnary of *jagged arrays* (*i.e.* array with variable size):

```
ar_dict = uptree.arrays(namedecode='utf-8')
```

One can simply make a pandas dataframe out of these arrays and finally having all our collider data into the dataframe `df`:

```
df = pd.DataFrame(ar_dict)
```

A quick invesigation of this dataframe can be the nember of events and a print out of two columns:

```
print('Number of events: {:.0f}'.format(len(df)))
print(df[['jet_pt', 'el_pt']].head())
```

```
Number of events: 250000
                                              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]
```

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 (*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**:

```
jet_pt_df = df['jet_pt'].values
print('shape: {}'.format(jet_pt_df.shape))
print('2 first events: \n{}'.format(jet_pt_df[0:2]))
```

```
shape: (250000,)
2 first events:
[array([169695.5 , 122250.03], dtype=float32)
 array([92278.93 , 70800.66 , 69653.164, 27776.486], dtype=float32)]
```

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). Several other operations which are rather standard in HEP (and probably in any highly dimensional data) are also implemented in a pure vectorized way in the module `np_utils.py`:

```
import np_utils as npu
```

The squaring of arrays is done by the function `square_jagged_2Darray(a,val=value,nobj=Nmax)`, of which the `docstring` is printed below showing what can be done:

```
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)
```

One can test this function on the jet p_T array for the three first event, for different formating of the array. First, we can use only the `val` argument to specify the default value used to fill the "missing jets":

```
jet_pt_np = npu.square_jagged_2Darray(jet_pt_df, val=0.0)
print('shape: {}'.format(jet_pt_np.shape))
print('2 first events: \n{}'.format(jet_pt_np[0:2]))
```

```
shape: (250000, 11)
2 first events:
[[169695.5   122250.03       0.         0.         0.         0.
       0.         0.         0.         0.         0.   ]
 [ 92278.93   70800.66   69653.164  27776.486      0.         0.
       0.         0.         0.         0.         0.   ]]
```

If the maximum size is expected to be really large (*e.g.* only one event has 53 jets and the next highest value is only 8), one can specify the maximum number of object to take with `nobj` argument:

```
jet_pt_np_max3 = npu.square_jagged_2Darray(jet_pt_df, val=-999, nobj=3)
print('shape: {}'.format(jet_pt_np_max3.shape))
print('2 first events: \n{}'.format(jet_pt_np_max3[0:2]))
```

```
shape: (250000, 3)
2 first events:
[[169695.5   122250.03    -999.   ]
 [ 92278.93   70800.66   69653.164]]
```

3.2.2 Timing and impact of vectorialization

Squaring array if obviously longer than taking the direct data from the dataframe, and limiting the number of objects doesn't change the picture (as explained later):

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

# Squaring the array with a default value of 0
%timeit npu.square_jagged_2Darray(jet_pt_df, val=0)

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

```
2.39 µs ± 60.2 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
128 ms ± 949 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
126 ms ± 540 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

It can be seen that the `square_jagged_2Darray()` is longer than directly taking the numpy array (and the reason will be explained just below). However, once it's done, one can perform operation much faster. By compare the computation time for a simple operation like counting the number of jets with a $p_T \geq 50$ GeV using both the squared array and a comprehensive loop on the original array from `df`, we see a factor *~500 faster* with squared array:

```
# Comprehensive loop for Njets with pT>50 GeV
%timeit Njets=[np.count_nonzero(j>50e3) for j in jet_pt_df]

# Squared array for Njets with pT>50 GeV
%timeit Njets=np.count_nonzero(jet_pt_np>50e3)
```

```
648 ms ± 6.23 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.1 ms ± 11.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

3.2.3 Detail of `square_jagged_2Darray()` function

Now, we want to analyze how this function works and understand where the long execution time comes from. There are mostly two long 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:

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:{}'.format(lens[:3]))
```

```
34.1 ms ± 336 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
lens:[2 4 5]
```

2. Create a mask (boolean array) to know which value should be filled, using broadcasting and `np.newaxis` to combine a `lens.shape = (250000,)` and `np.arange(lens.max()) = (11,)`:

```
%timeit mask = lens[:, np.newaxis] > np.arange(lens.max())
mask = lens[:, np.newaxis] > np.arange(lens.max())
print('mask:\n {}'.format(mask[:3]))
```

```
5.93 ms ± 61.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
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]]
```

3. Initialize the final squared array with the proper size:

```
%timeit out = np.zeros(mask.shape, dtype='float32')
out = np.zeros(mask.shape, dtype='float32')
print('out:\n {}'.format(out[:3]))
```

```
1.39 ms ± 15.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
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.]]
```

4. Fill the default values everywhere:

```
%timeit out.fill(999)
out.fill(999)
print('out:\n {}'.format(out[0:3]))
```

```
1.16 ms ± 23.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
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.]]
```

5. Overwrite the values where a jet exist (1D array `out[mask]`) with all the jet p_T (1D array `np.concatenate(jet_pt_df)`):

```
%timeit out[mask] = np.concatenate(jet_pt_df)
out[mask] = np.concatenate(jet_pt_df)
print(('out:\n {}'.format(out[0:3])))
```

```
66.7 ms ± 5.07 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
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)
```

What follows illustrate this function with 3 examples running over 1000 events only, each time printing the shape of the array:

```
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)
```

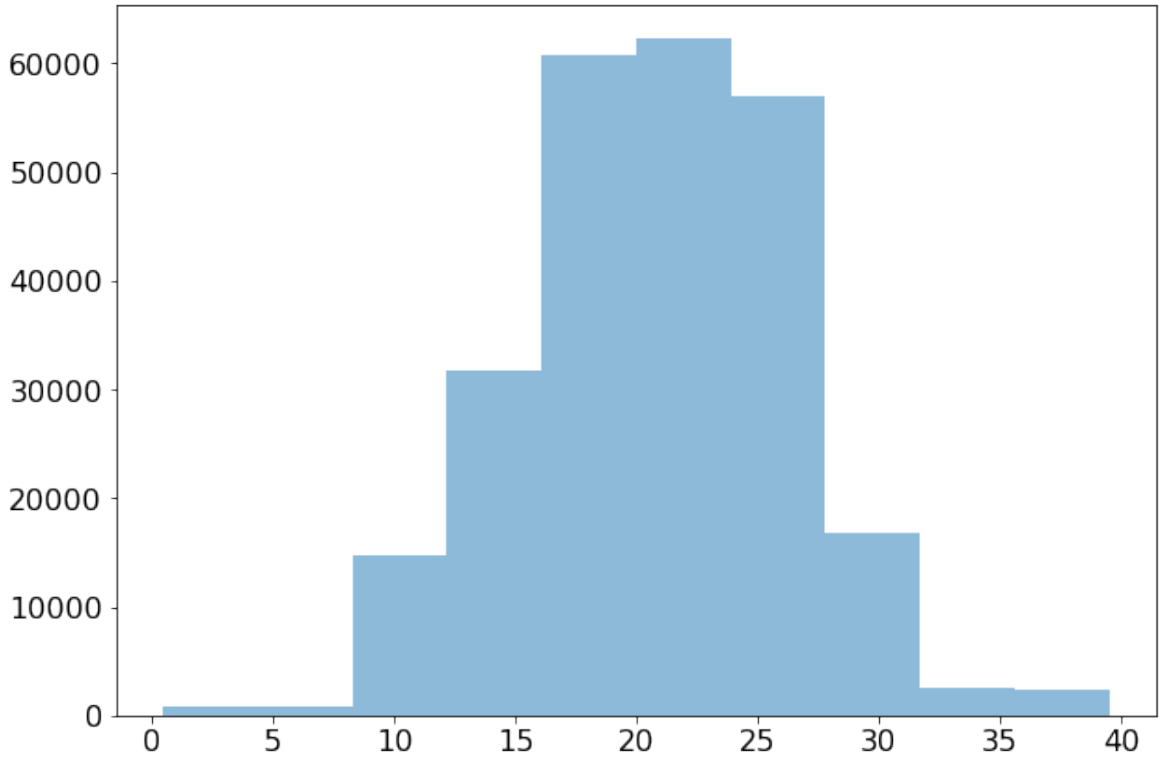
3.3 Producing some non-trivial plots using numpy arrays

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))
plt.hist(df['mu'], alpha=0.5);
```



3.3.1 Jet multiplicity for different p_T thresholds

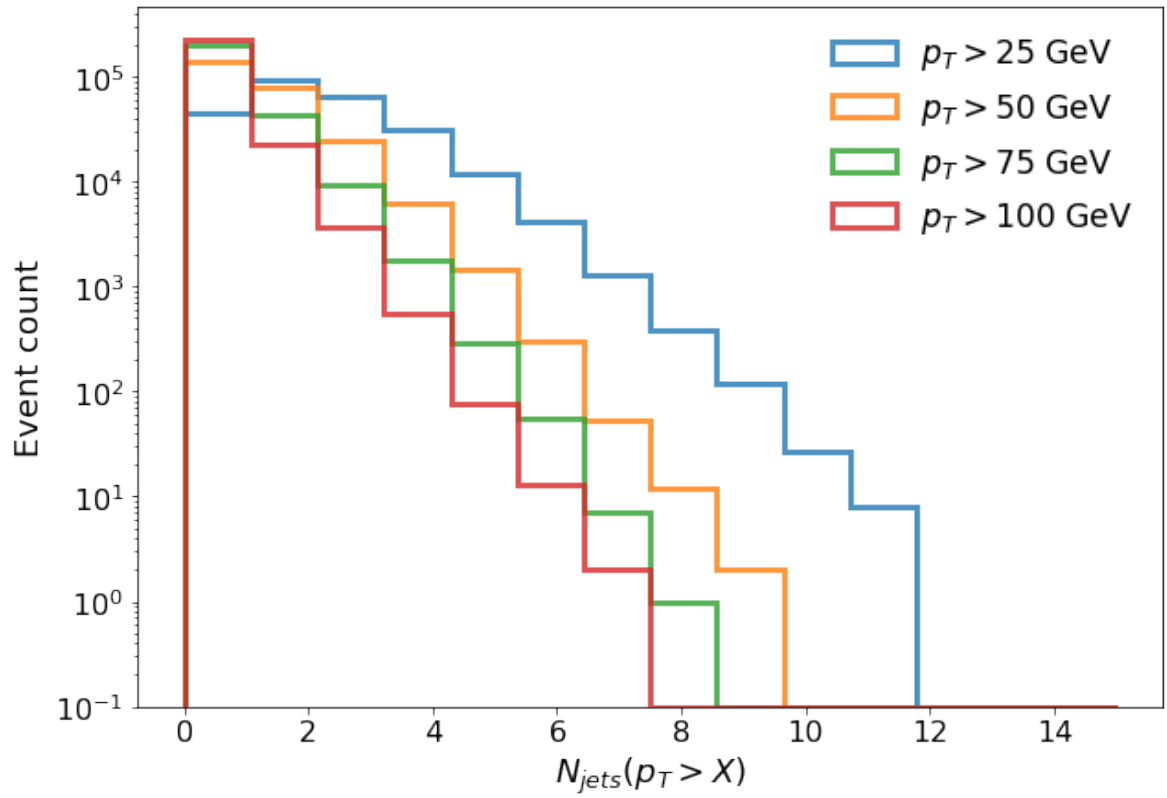
Looking at the jet multiplicity depending on the p_T threshold:

- `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).

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

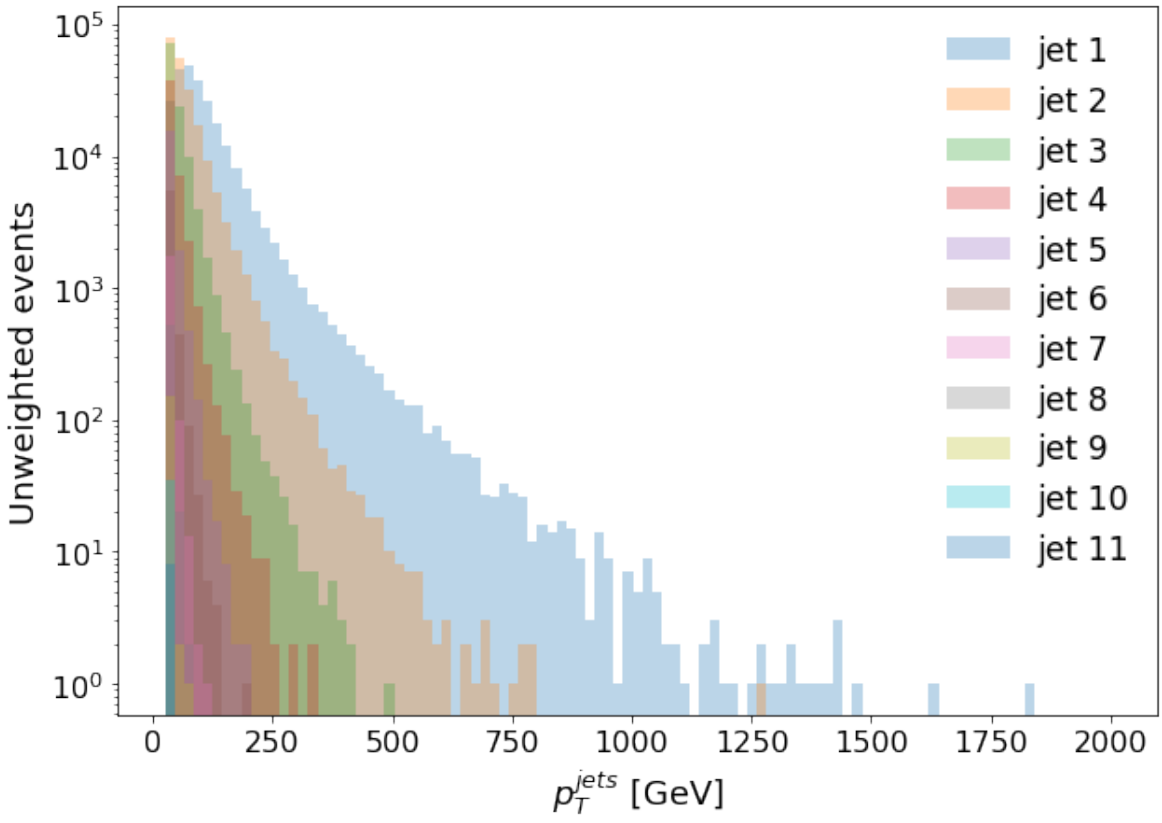
```
plt.figure(figsize=(10, 7))

plot_style = {
              'alpha': 0.8,
              'histtype': 'step',
              'linewidth': 3,
              'bins': np.linspace(0, 15, 15),
              'log': True
             }

for pt in np.linspace(25, 100, 4)*1000:
    plt.hist(np.count_nonzero(jets[..., 0] > pt, axis=1),
             label='$p_T>{:.0f}$ GeV'.format(pt/1000.),
             **plot_style)
plt.legend()
plt.xlabel('$N_{jets}(p_T>X)$')
plt.ylabel('Event count');
```



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))
plot_style = {
               'alpha': 0.3,
               'linewidth': 3,
               'bins': np.linspace(25, 2000, 100),
               'log': True,
              }

Njets = jets_pt_plots.shape[1]
for i in np.arange(Njets):
    plt.hist(jets_pt_plots[:, i]/1000., label='jet {}'.format(i+1),
             **plot_style)
plt.legend()
plt.xlabel('$p^{jets}_T$ [GeV]')
plt.ylabel('Unweighted events');
```



3.3.3 *Apparte:* reminder of the 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]
```

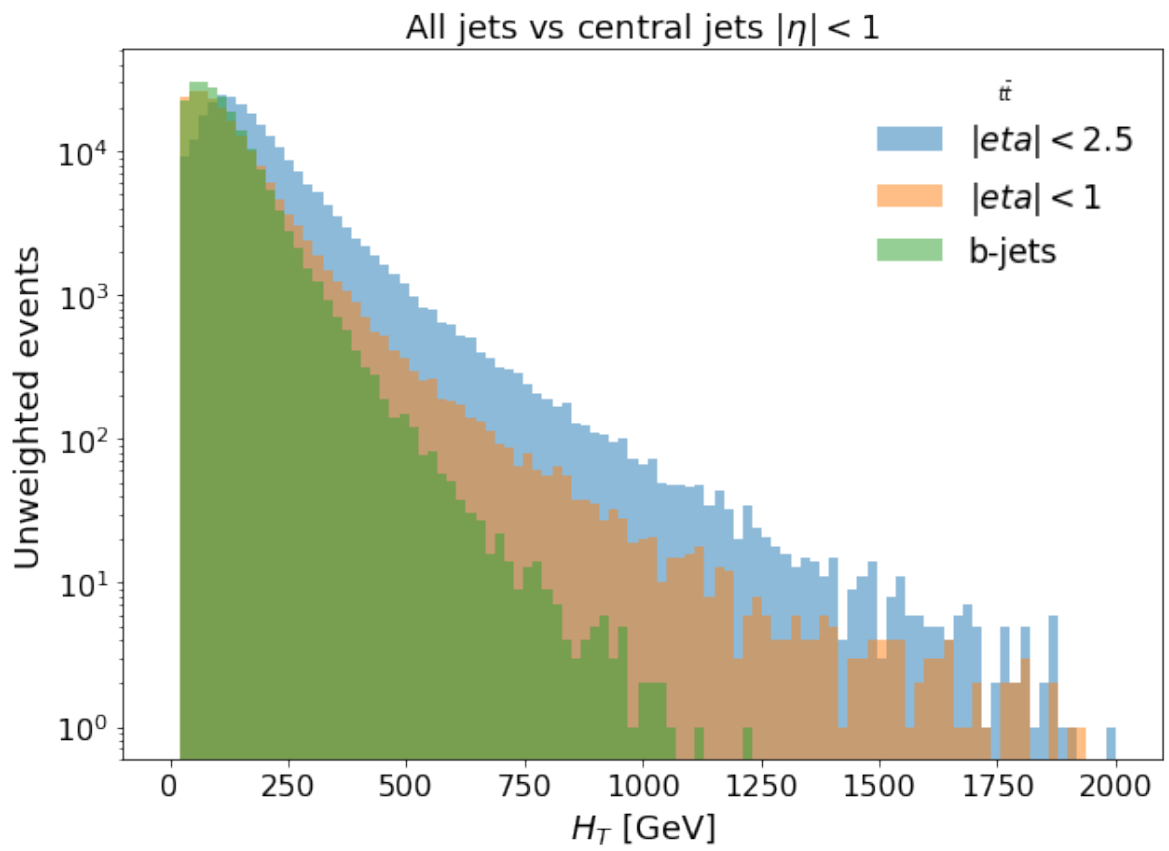
3.3.4 H_T distribution in different configurations

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)
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)
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)
plt.hist(HTbjets[HTbjets > 0], alpha=0.5,
         bins=np.linspace(0, 2000, 100), label='b-jets', log=True)

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



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 11 to 5 (in term of N(N-1)/2 combinations: 55 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):

```
nj=npu.count_nonnan(jets[...,0],axis=1)
```

2. Select all jets and all variables for events with `nj>1`:

```
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)
```

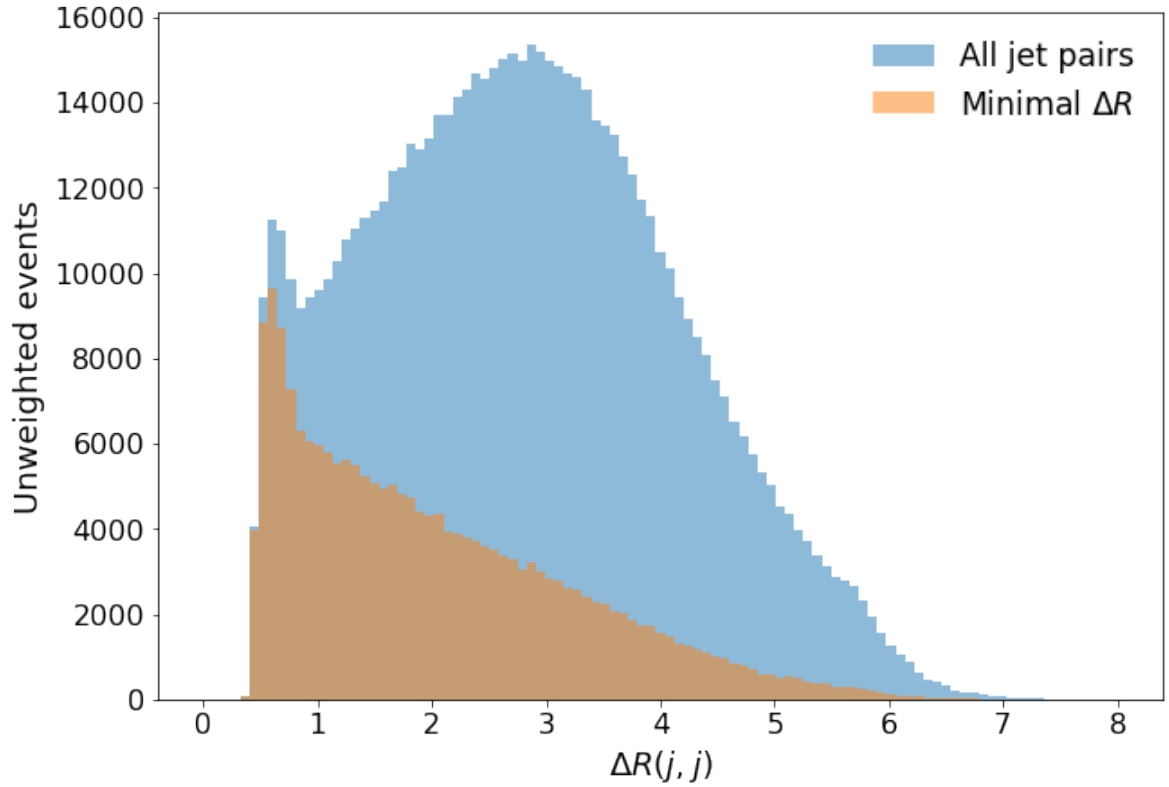
```
dj_etaphi = npu.replace_nan(dj_etaphi, value=999)

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

```
(250000, 55, 2) [[-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]]
```

```
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) [   3.5524466 1412.7993    1412.7993    1412.7993    1412.7993   ]
(250000, 55) [  3.5524466 999.        999.        999.        999.       ]
```

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



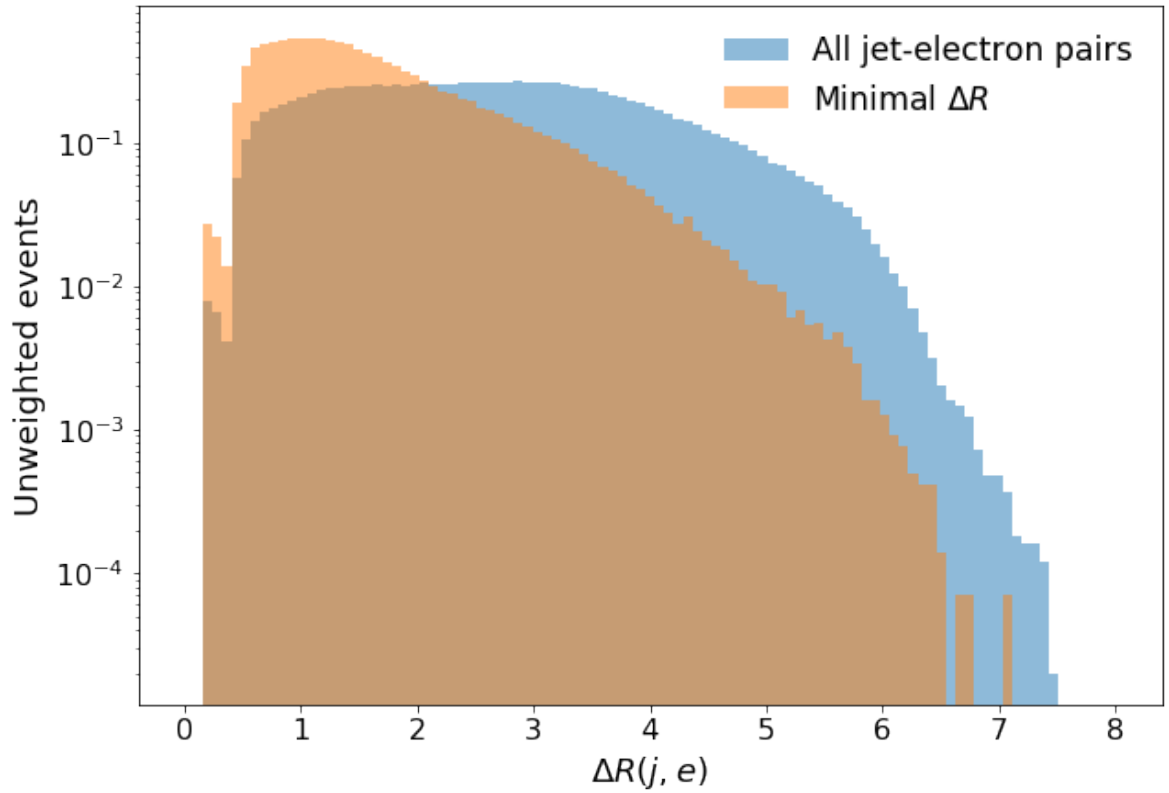
3.4.3.2 Minimum $\Delta R(j, e)$

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



3.4.4 Di-jet invariant masses

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

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

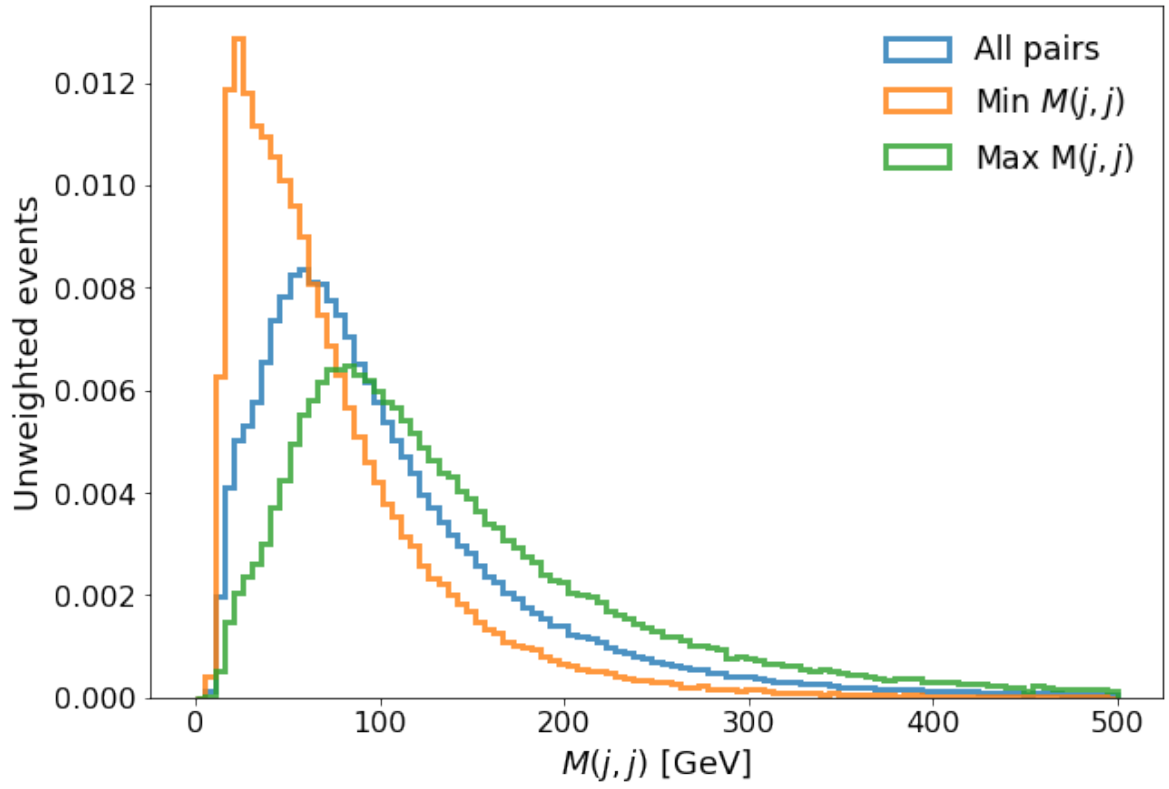
```
deta, dphi = dj_etaphi[..., 0], dj_etaphi[..., 1]
pt1, pt2 = j1[..., 0], j2[..., 0]
print(pt1.shape, deta.shape)
```

```
(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
}

plt.hist(m.flatten(), label='All pairs', **style)
plt.hist(np.min(m, axis=1), label='Min $M(j,j)$', **style)
plt.hist(np.max(npu.replace_val(m, 1e10, -1e10), axis=1),
             label='Max M$(j,j)$', **style)
plt.xlabel('$M(j,j)$ [GeV]')
plt.ylabel('Unweighted events')
plt.legend();
```



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'])
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, 8,45)
  -> 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'])
electrons = df2array(df,['el_pt', 'el_eta', 'el_phi' , '-999'])
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)
```

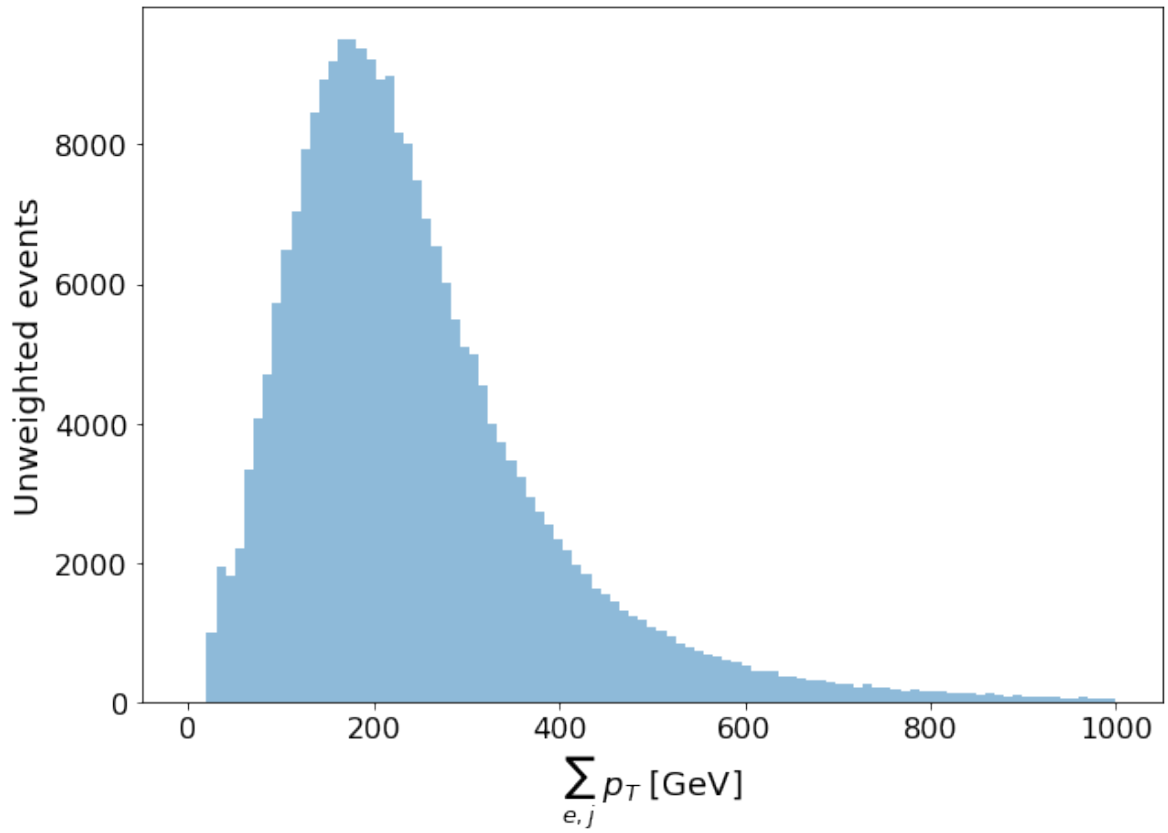
```
(250000, 11, 5) (250000, 3, 5) (250000, 14, 5)
```

In order to compute a given observable, we can replace `nan` values that were set above by a proper one depending on the computation). To compute the sum H_T, `0` is a proper value (keep in mind that for more complex computation, *masked arrays* can be more approriate):

```
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)
plt.xlabel('$\sum_{e,j} \; p_T$ [GeV]')
plt.ylabel('Unweighted events');
```



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 a single object selected based on a more global criteria (activities around this object, its angle with another object, etc ...). One concrete case could be the isolation of the leptons which form a pair having $M(e,e) \sim M(Z)$. Another case could be to look at the b-tagging weight of the jet closest to a muon in the event.

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, not only with the direction but all needed variables (lile b-tagging weight):

```
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) contanining the btagg weight (3rd variable) of the first element (*i.e.* the jet) for any pair: `btagw=jets_ele_pairs[:, :, 0, 3]`

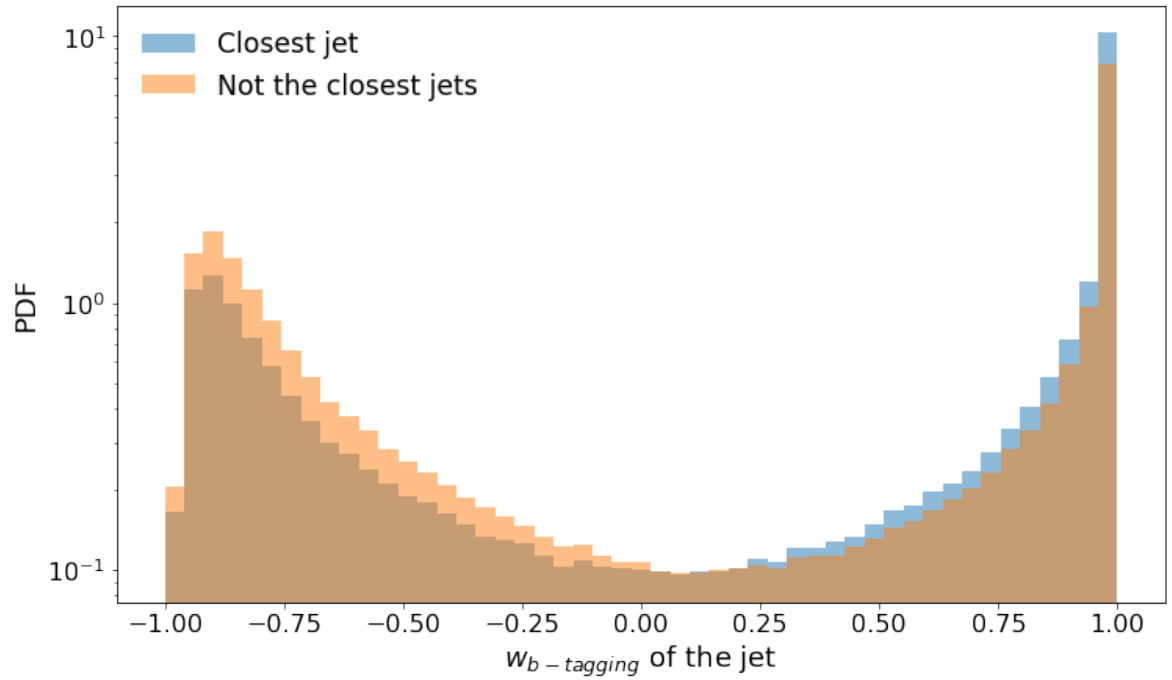
```
jet_btag_w = npu.replace_nan(jets_elec_pairs[:, :, 0, 3], value=999)
```

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 jet index for each event. Then one can use the functions `get_indexed_value()` and `get_all_but_indexed_value()` from `nu_utils.py` to get either the btag weight of the minimal *dR* or all the others. We remind that `dRej` was already computed earlier.

```
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)
```

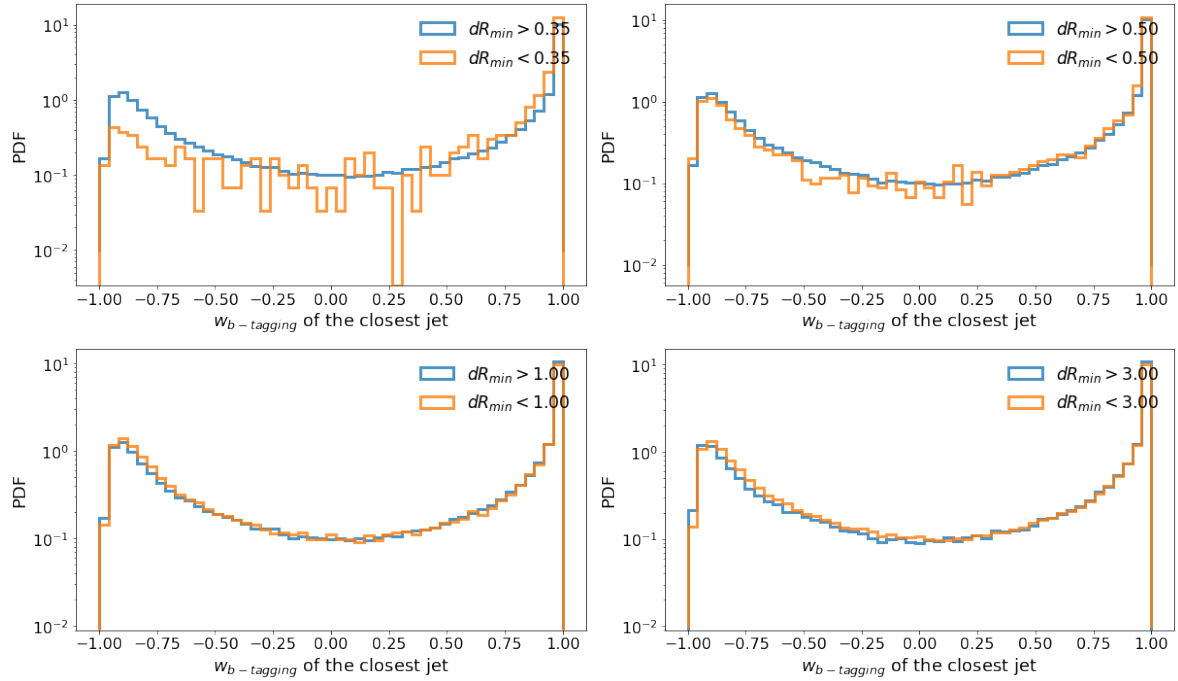
One then can plot the distribution of the b-tagging weight for both the closest jet and all the other, to see if this jet observable is dependent from its environement.

```
plt.figure(figsize=(12,7))
plt.hist(jet_btag_w_dRmin, bins=np.linspace(-1, 1, 50),
              alpha=0.5, density=True, log=True, label='Closest jet')
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')
plt.xlabel('$w_{b-tagging}$ of the jet')
plt.ylabel('PDF')
plt.legend();
```



We can also do the same game but for different region in `dRmin`, in order to see if there are not small effects if the object are *really* close or not too close.

```
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
    plt.hist(dRgt_btag,
                  label='$dR_{min}>'+'{:.2f}$'.format(cut), **style)
    plt.hist(dRlt_btag,
                  label='$dR_{min}<'+'{:.2f}$'.format(cut), **style)
    plt.xlabel('$w_{b-tagging}$ of the closest jet')
    plt.ylabel('PDF')
    plt.legend()
plt.tight_layout();
```

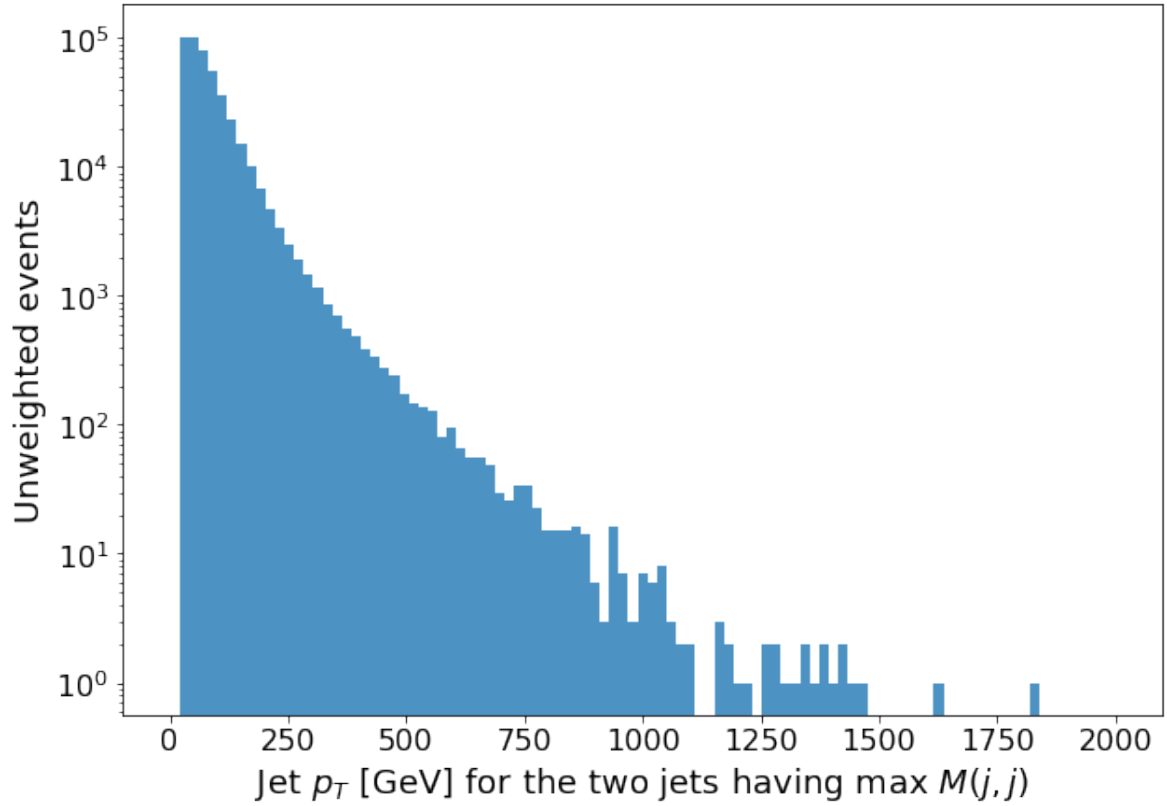


3.5.2.2 E.g. 2: p_T distribution for jets forming the highest $M(j,j)$

```
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,
}
plt.hist(npu.replace_nan(pt_mjj_max, -999), **style)
plt.xlabel('Jet $p_T$ [GeV] for the two jets having max $M(j,j)$')
plt.ylabel('Unweighted events');
```



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**

```
# 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
```

I didn't find any example of how to save numpy arrays as TTree using uproot (to be followed up). But it's definitely possible with root_numpy (but wich directly needs ROOT installed):

```
# Possible with root_numpy
from root_numpy import array2root
array2root(events, 'ttbar_jets.root', 'tree_jets', mode='recreate')
```

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 developpement to manage jagged array as numpy array without the "squaring" I used. This tool is called awkward-array and is directly implemented into uproot.

Conclusion and perspectives

It is clear that NumPy allows to perform very sophisticated calculation on multi-dimensional data, in a very consise piece of code and rather efficiently. These are clearly strong points in favour of using of NumPy, together with the large collection of tools which come along. All these tools form a very handy and powerful ecosystem for a lot of studies that can be performed in data analysis and numerical computation fields.

However, in my opinion, this ecosystem is not always scalable to large data samples on which event-by-event complex calculations have to be executed. The fundamental reason is of course that python is not compiled but the implicit comparison here is done with C++ and more specifically ROOT software, fully designed for particle physics (while NumPy is not, so the fairness of the comparison is debatable even if conclusions remain relevant). In order to illustrate these limitations with concrete cases taken from these notes, one could focus on the three following points.

- **Code readability.** Manipulating multi-dimensional NumPy arrays using slicing and indexing can become quite difficult to read at some point. Indeed, one needs to exactly remember how the data are stored, *i.e.* the meaning of each axis. This can be clear when one writes the code, but can become slightly convoluted when re-reading its own code few months later. The typical following lines of code might illustrate this difficulty:

```
j1, j2 = jet_pairs[:, :, 0, :], jet_pairs[:, :, 1, :]
mjj = j1[..., 0]*j2[..., 0]
      * (np.cosh(j1[..., 1]-j2[..., 1])-np.cos(j1[..., 2]-j2[..., 2])))
```

- **Computation methodology.** The strength of NumPy clearly relies on the vectorized computations, which requires to eradicate explicit *for* loops. If this is sometimes nicer to not have loops (shorter/clearer code), it might make life difficult for any arbitrary type of computations. The example discussed in these note of making every possible pairs of objects per event is probably a good illustration. Indeed, one had to really think how to implement the pair combinatorics in a vectorized way without *for* loops, and the solution might not always be straightforward. This effort might be redone every time there is a new complex computation that comes up.

- **Memory.** The final point which limits this ecosystem for large and complex computations is the memory management. This wasn't mentioned at all in these note but it can quickly become a limitation *if we use the existing tools out-of-the-box*. When data are loaded, the RAM of the computer is used and if the initial data sample is large *and* complex calculations are performed (creation of intermediate arrays), the saturation of the RAM arrives rather quickly. It's one of the reason why the notebooks of these notes cannot all be ran into `mybinder` (issue of "dead kernel").