Optics_Matrix_Prototype.py

1. Under the main function, we have defined an object "optics" under a class of the same name.

```
if __name__=='__main__':

    optics=OPTICS()

    optics.GenNumpyArray("../../Cfoil_elastic_p2_slim.root")
    optics.DefineSectors()
    optics.DrawHistAllSectors()
    optics.DrawScatterPlot(optics.sec1.gem1_x, optics.sec1.gem1_y)
    optics.SelectOneHole(optics.sec1)

    hole_id="12"
    filename="output/SieveHole_"+hole_id+".csv"
    optics.GenCSV(hole_id, filename)
```

Within a class we have methods and constructors. Methods are nothing but functions. Constructor is a special kind of method.

The first thing that we define when we create a class is a constructor.

The keyword "self" is used to assign the arguments passed to the method to the attributes. It also communicates among different functions defined within the class.

```
class OPTICS:

    def __init__(self):
        self.orig = pd.DataFrame()          # data before cut
        self.sec1 = pd.DataFrame()          # sector1 data before cut
        self.sec2 = pd.DataFrame()          # sector2 data before cut
        self.sec3 = pd.DataFrame()          # sector3 data before cut
        self.sec4 = pd.DataFrame()          # sector4 data before cut
        self.sec5 = pd.DataFrame()          # sector5 data before cut
        self.sec6 = pd.DataFrame()          # sector6 data before cut
        self.sec7 = pd.DataFrame()          # sector7 data before cut

        self.selected = pd.DataFrame()      # data of selected holes
```

we have defined 8 empty panda dataframes inside the constructor.

This is the first thing that will be executed when creating an instance of the class.

2. After initializing panda dataframes , we are filling the panda dataframe self.orig from data in the slim root files.

```
import uproot
import awkward
import pandas as pd
import math
import matplotlib.pyplot as plt

df=pd.DataFrame()

file = uproot.open("../../../Cfoil_elastic_pl.root")
print(file.classnames()) #This statement gives the objects within the file and what class these objects belong to
T = file['newT']
print(T.keys())             # We want to know what are the branches inside the tree.

branches = T.arrays()
print(branches['main_r'])  # prints out the data in the branch main_r for all the events
print(branches['main_r'][0]) #prints out the value of main_r for event 0

geo = T.arrays(["gem1_x", "gem1_y","gem1_r","gem1_ph","gem1_px","gem1_py","gem1_pz","tg_th","tg_ph","tg_vz","tg_p","rate"],library="pd")
geo.loc[geo['gem1_r'] > 300]
df=geo

print(df.columns)
```

First task would be to open the root file. For this purpose we are using the uproot library.

- The next task is to read the data from the branches. For this purpose we will use the T.arrays() method.

- Within the panda dataframe we only need to store information from a certain number of branches.

- The "loc" function filters the events according to the condition that "gem1_r > 300".

3. The next step is to fill the 7 empty panda dataframes one corresponding to each sector . For this purpose we need to have a definition of sectors That's what the DefineSectors() function does.

```
def DefineSectors(self):
    rot_angle=0        # put the GEM rotation angle here

    angle_lo=[]
    angle_up=[]

    for i in range(7):
        angle_lo.append(math.pi/14 + i*2*math.pi/7+rot_angle)
        angle_up.append(3*math.pi/14 + i*2*math.pi/7-rot_angle)
        if angle_lo[i]>math.pi:
            angle_lo[i]=angle_lo[i]-2*math.pi;
        if angle_up[i]>math.pi:
            angle_up[i]=angle_up[i]-2*math.pi;

    geo=self.orig

    self.sec1=geo.loc[(geo["gem1_ph"]<(angle_up[0])) & (geo["gem1_ph"]>angle_lo[0])]
    self.sec2=geo.loc[(geo["gem1_ph"]<(angle_up[1])) & (geo["gem1_ph"]>angle_lo[1])]
    self.sec3=geo.loc[(geo["gem1_ph"]<(angle_up[2])) & (geo["gem1_ph"]>angle_lo[2])]
    self.sec4=geo.loc[((geo["gem1_ph"]<(angle_up[3])) & (geo["gem1_ph"]>=-math.pi)) | ((geo["gem1_ph"]>angle_lo[3]) & (geo["gem1_ph"]<math.pi))]
    self.sec5=geo.loc[(geo["gem1_ph"]<(angle_up[4])) & (geo["gem1_ph"]>angle_lo[4])]
    self.sec6=geo.loc[(geo["gem1_ph"]<(angle_up[5])) & (geo["gem1_ph"]>angle_lo[5])]
    self.sec7=geo.loc[(geo["gem1_ph"]<(angle_up[6])) & (geo["gem1_ph"]>angle_lo[6])]
```

We are again using the "loc" function to fill out the panda dataframes for each sector from the overall self.orig dataframe that we filled in the last step.

```
12.857142857142858 38.57142857142857   sect1
12.857142857142858 38.57142857142857
54.28571428571429 90.0                 sect2
54.28571428571429 90.0
115.71428571428571 141.42857142857142  sect3
115.71428571428571 141.42857142857142
167.14285714285714 192.85714285714286  sect4
167.14285714285714 -167.14285714285714
218.57142857142858 244.28571428571428  sect5
-141.42857142857142 -115.71428571428571
270.0 295.7142857142857                sect6
-90.0 -64.28571428571429
321.42857142857144 347.14285714285717  sect7
-38.57142857142858 -12.857142857142858
(venv) ifarm1802.jlab.org>
```

→ } Definition of the seven sectors using the ϕ parameter before and after the $\pm\pi$ subtraction

4. The next task is to draw 2D Histograms for all the 7 sectors. For this purpose we are using the "Draw Hist All Sectors" function

```
def DrawHistAllSectors(self):
    fig, axs = plt.subplots(2, 4, figsize=(20,10))

    #fig,ax = plt.subplots(figsize =(10, 7))
    #plt.hist2d(npy_arr["main_x"], npy_arr["main_y"])
    #pts=ax.scatter(geo["geo1_x"], geo["geo1_y"])

    axs[0,0].hist2d(self.orig.geo1_x, self.orig.geo1_y,(200,200),cmap=plt.cm.jet, cmin=1)
    #axs[0,0].set_xlim(-1000,1000))
    #axs[0,0].set_ylim(-1000,1000))

    axs[0,1].hist2d(self.sec1.geo1_x,self.sec1.geo1_y,(100,100),cmap=plt.cm.jet, cmin=1)
    axs[0,1].set_title('sec1')
    axs[0,2].hist2d(self.sec2.geo1_x,self.sec2.geo1_y,(100,100),cmap=plt.cm.jet, cmin=1)
    axs[0,2].set_title('sec2')
    axs[0,3].hist2d(self.sec3.geo1_x,self.sec3.geo1_y,(100,100),cmap=plt.cm.jet, cmin=1)
    axs[0,3].set_title('sec3')
    axs[1,0].hist2d(self.sec4.geo1_x,self.sec4.geo1_y,(100,100),cmap=plt.cm.jet, cmin=1)
    axs[1,0].set_title('sec4')
    axs[1,1].hist2d(self.sec5.geo1_x,self.sec5.geo1_y,(100,100),cmap=plt.cm.jet, cmin=1)
    axs[1,1].set_title('sec5')
    axs[1,2].hist2d(self.sec6.geo1_x,self.sec6.geo1_y,(100,100),cmap=plt.cm.jet, cmin=1)
    axs[1,2].set_title('sec6')
    axs[1,3].hist2d(self.sec7.geo1_x,self.sec7.geo1_y,(100,100),cmap=plt.cm.jet, cmin=1)
    axs[1,3].set_title('sec7')

    plt.show()
```

We are using the "subplots" function defined inside the "matplotlib.pyplot" library.

```
fig, axs = plt.subplots(2,4)
```

↓ This is like an empty canvas

↓ We are dividing the canvas into 8 subparts.

```
axs[0,0].hist2d ( x , y , (nbinsX,nbinsY) , cmap, cmin)
```

↓ Defines the color map

↘ No. of bins with bin content less than this value will not be displayed.

After defining all the subplots we just need to call "plt.show()" function.

5. Now the next task is to select one particular hole.

```
def SelectOneHole(self, df):

    fig, ax = plt.subplots(figsize=(10,7))

    pts=ax.scatter(df.gem1_x,df.gem1_y)

    y_max=df.gem1_y.max()
    y_min=df.gem1_y.min()
    dy = (y_max-y_min)*0.1

    x_max=df.gem1_x.max()
    x_min=df.gem1_x.min()
    dx = (x_max-x_min)*0.1

    ax.set_ylim(y_min-dy, y_max+dy)
    ax.set_xlim(x_min-dx, x_max+dx)

    selector = SelectFromCollection(ax, pts)

    print("Select points in the figure by enclosing them within a polygon.")
    print("Press the 'esc' key to start a new polygon.")
    print("Try holding the 'shift' key to move all of the vertices.")
    print("Try holding the 'ctrl' key to move a single vertex.")

    plt.show()

    selector.disconnect()
    self.selected=df.loc[df.index[selector.ind]]
```

→ Producing a scatter plot

→ Setting the min and max values of the plot.

This is the function we are using to select one particular hole

we are using

from polygone_selector_demo import SelectFromCollection

Again using the loc function to fill the self.selected data frame.

6. The next step is to generate a csv file from the selected hole dataframe.

```
def GenCSV(self, hole_id, filename):

    df=self.selected
    df["gem1_rp"]=(df.gem1_x*df.gem1_px+df.gem1_y*df.gem1_py)/(df.gem1_r*df.gem1_pz)      # gem 1 r'
    df["gem1_php"]=(-df.gem1_y*df.gem1_px+df.gem1_x*df.gem1_py)/(df.gem1_r*df.gem1_pz)    # gem 1 phi'

    header=["tg_th","tg_ph","tg_vz","tg_p","gem1_r","gem1_rp","gem1_ph","gem1_php"]
    df.to_csv(filename,columns=header)
```

There are a lot of columns in the panda dataframe. We do not need to load all of those in the csv file.

For this reason we are passing the argument "columns = header" inside df.to_csv()

7. Now before we can perform a polynomial regression we need to understand how python implements Machine Learning methods in polynomial regression.

- We first divide the dataset into a training set and a testing set. Usually it's an 80:20 ratio but we can change the relative sizes of the two sets.
- If we assume that z depends upon two variables x & y. Machine Learning methods also studies the dependence of z on the correlations of x and y.

$z = f(x,y)$ and we want upto second degree polynomial Regression.

$x^0, x^1, x^2, y^0, y^1, y^2, xy$] we need information on all these.

we use the "sklearn" module to perform the polynomial regression.

```
def PolynomialRegression(self, X, y, degree):  # X is the GEM variables in numpy array, y is the target variable

    X_train,X_test,y_train,y_test=train_test_split(X, y, test_size=0.33, random_state=42) # split the data into traning set and test set


    poly = PolynomialFeatures(degree)
    X_train_new=poly.fit_transform(X_train)   # transform columns into polynomial columns
    X_test_new=poly.fit_transform(X_test)

    #print(X_train_new)
    regression = linear_model.LinearRegression()

    model = regression.fit(X_train_new, y_train)
    y_pred = regression.predict(X_test_new)

    params = model.coef_
    intercept = model.intercept_

#   ax = plt.axes(projection='3d')
#   ax.scatter3D(X_test[:,0], X_test[:,1], y_test, c=y_test, cmap='Greens')
#   ax.scatter3D(X_test[:,0], X_test[:,1], y_pred, c=y_pred, cmap='Reds')
    plt.show()

    ax = plt.axes()
    #ax.scatter(X_test[:,1:], y_test, c=y_test, cmap='Greens')
    ax.scatter(X_test[:,0], y_test, cmap='Greens')
    #ax.scatter(X_test[:,1:], y_pred, c=y_pred, cmap='Reds')
    ax.scatter(X_test[:,0], y_pred, cmap='Reds')
    plt.show()


    #self.TestFit(X_test,y_test,params[0],intercept)

    score = model.score(X_test_new, y_test)
    print("score: ", score)
```

The first module that we need is

from sklearn.model_selection import train_test_split.

train_test_split() is used to split the dataset into a training and a testing set.

Another feature of train_test_split function is that it also randomizes the data set.

Then we need to take into account all the terms in the fit function.

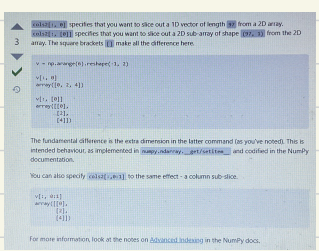
$$\theta_{tg} = f(r, r')$$

$$= r^0 \quad r' \quad r^2$$
$$r'^0 \quad r'' \quad r'^2$$
$$rr'$$

Within the csv file we already have columns for r and r' Now we need to compute all the other columns.

from sklearn.preprocessing import PolynomialFeatures

we define the model using the training set and we test the efficiency of our model using the testing set.



→ Important Information when we will show the accuracy of our model using the testing set.