Aufgabe11

June 8, 2020

1 Aufgabe 11: Simulationskette für Neutrinodetektor

Verwenden Sie für diese Aufgabe die Pythonbibliothek pandas. Füllen Sie die Ergebnisse der einzelnen Teilaufgaben, abgesehen von der Letzte, in ein DataFrame und speichern Sie dieses am Ende in einer hdf5-Datei NeutrinoMC.hdf5 mit dem Key Signal. Die Ergebnisse der letzten Teilaufgabe sollen in ein eigenes DataFrame geschrieben und in der selben hdf5-Datei unter dem Key Background gespeichert werden. Zum Schreiben einer hdf5-Datei besitzt das DataFrame die Methode to_hdf(). Wichtig: Die erstellte hdf5-Datei soll nicht mit abgegeben werden! Das fertige Programm muss ohne die bereits existierende hdf5-Datei funktionieren.

[1]:
```
import numpy as np
import pandas as pd
from project_c3.random import Generator
import matplotlib.pyplot as plt

rs = np.random.RandomState()
```

1.1 (a) Signal MC

Der Fluss der Neutrinos ist gegeben durch: $\Phi = \Phi\ 0\quad (\ \mathrm{E} -\ )$. TeV Hierbei ist der spektrale Index $= 2{,}7$, die untere Energiegrenze 1 TeV und die obere Energie- grenze unendlich. Simulieren Sie mit Hilfe der Transformationsmethode 10 5 Signalereignisse und speichern Sie diese in dem DataFrame unter dem Key Energy.

Transformationsmehtode: Normieren: $1 = \int_1^\infty \Phi_0(E)^{-\gamma} \rightarrow \Phi_0 = \gamma - 1$

Fläche bis Zufallsvariable: $A(E) = \int_1^E \Phi_0(E)^{-\gamma} = \frac{1}{1-\gamma}(E^{1-\gamma} - 1)$

Fläche normieren: $r(E) = A(E)\Phi_0 = 1 - E^{1-\gamma}$

Nach E umstellen: $E(r) = (1-r)^{1/(1-\gamma)}$

[2]:
```
#Konstanten
  = 2.7
Φ_0 =   -1


#Zufallszahlen
r = rs.uniform(size=10**5)
```

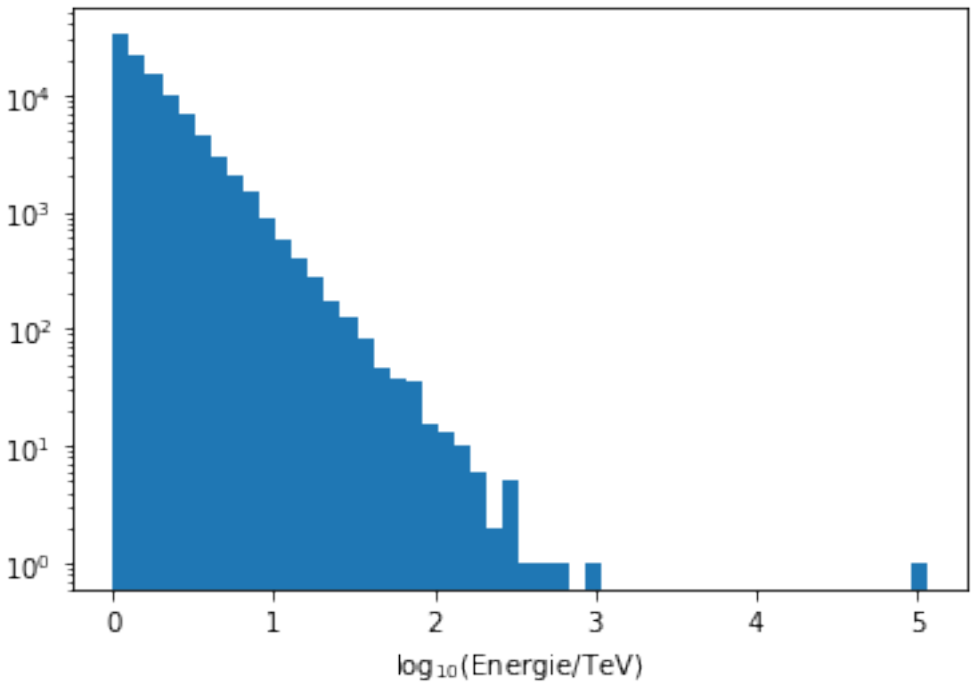
```
#Zufallszahl die der Verteilung folgt
energy = (1-r)**(1/(1- )) #energie in TeV
```

1.2 (b) Akzeptanz

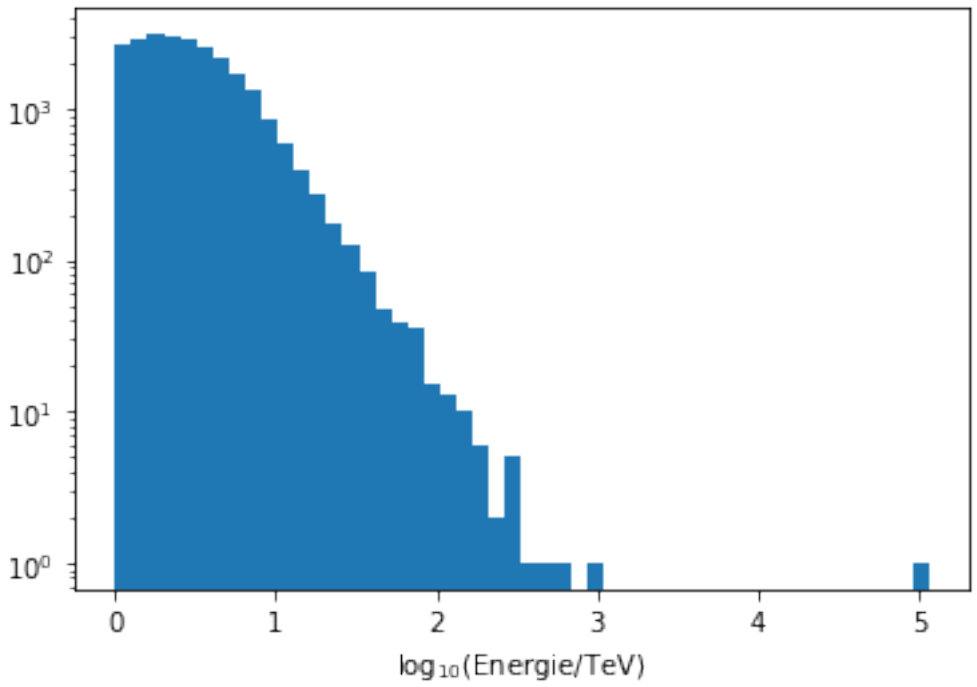
Die Wahrscheinlichkeit, ein Ereignis zu detektieren, ist energieabhängig. Dies muss bei der Simulation mit berücksichtigt werden. Die Detektionswahrscheinlichkeit lässt sich durch folgende Gleichung beschreiben: P (E) = (1 − e −E/2) 3 . Nutzen Sie das Neumann'sche Rückweisungsverfahren, um die Detektorakzeptanz für die in Teil a) simulierten Signal-Ereignisse zu berücksichtigen. Speichern Sie die Ergebnisse in Form einer Maske (True-False-Folge) unter dem Key Acceptance-Mask in dem DataFrame. Stellen Sie das Ergebnis von a) und b) in einem Plot dar. Hinweis: Nutzen Sie eine log-log Darstellung.

[3]:
```
def detektionswahrschl(E):
    return (1-np.exp(-(E)/2))**3

r1 = energy
r2 = rs.uniform(0, 1, size=10**5)

acceptancemask = r2 < detektionswahrschl(r1)

#im folgenden werden die berechnungen für alle ereignisse durchgeführt
#um nur die detektierten zu erhalten muss jeweils die maske angewendet werden␣
 ↪(siehe zweiter plot)
```

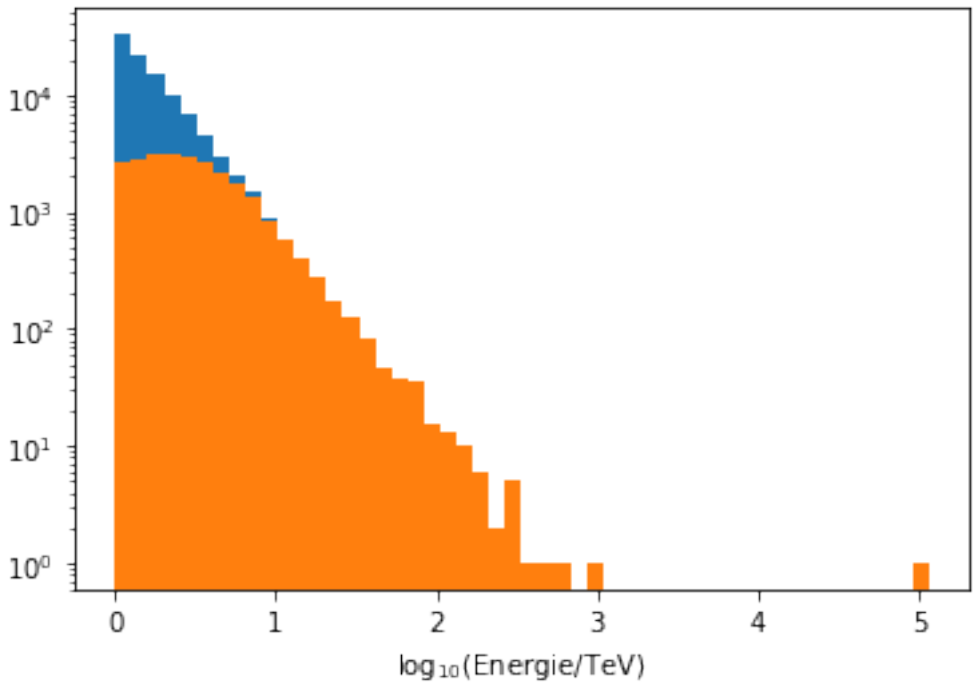
[4]:
```
plt.hist(np.log10(energy), log=True, bins=50) #alle Ereignisse
plt.xlabel(r'$\log_{10}{(\mathrm{Energie/TeV})}$')

None
```



[5]:
```
plt.hist(np.log10(energy)[acceptancemask==True], log=True, bins=50)␣
 ↪#detektierte Ereignisse
plt.xlabel(r'$\log_{10}{(\mathrm{Energie/TeV})}$')

None
```



[6]:
```
plt.hist(np.log10(energy), log=True, bins=50)
plt.hist(np.log10(energy)[acceptancemask==True], log=True, bins=50,)
plt.xlabel(r'$\log_{10}{(\mathrm{Energie/TeV})}$')
#detektierte über allen Ereignissen, um es besser vergleichen zu können
None
```



2 c) Polarmethode

Die Implementation ist in random.py. Es wurde die Polar-Methode aus der Vorlesung angewendet.

Polarmethode:

- Erzeuge gleichverteilte u1, u2
- Umformung $v1 = 2u1 - 1$, $v2 = 2u2 - 1$
- Berechne $s = v1^2 + v2^2$
- Verwerfe, wenn $s \geq 1$
- Berechne $x1 = v1\sqrt{-\frac{2}{s}\ln s}, x2 = v2\sqrt{-\frac{2}{s}\ln s}$

Das Skalieren hat leider nicht funktioniert –> siehe Test Datei

2.1 (d) Energiemessung

Ein realistischer Detektor besitzt nur eine endliche Energieauflösung. Zudem wird die Energie nicht direkt gemessen, sondern mit Hilfe energiekorrelierter Observablen rekonstruiert. Eine sol- che Observable ist bespielsweise die Anzahl der Photomultiplier, die angesprochen haben (im Folgenden als Hits bezeichnet). Die Anzahl der Hits lässt sich aus einer Normalverteilung mit folgenden Eigenschaften ziehen: N(10 E/TeV, 2 E/TeV). Hierbei bezeichnet N(10E/TeV, 2E/TeV) die

Normalverteilung mit Mittelwert 10E und Stan- dardabweichung 2E (E in TeV). Wandeln Sie die Zahl der Hits in eine ganze Zahl um, da nur ganze Anzahlen an Hits auftreten können. Achten Sie ebenfalls darauf, dass für die Anzahl der Hits N nur Werte oberhalb von Null in Frage kommen. Ziehen Sie gegebenenfalls eine neue Zufallszahl, falls diese Bedingung verletzt wird. Simulieren Sie die Anzahl der Hits in Abhänig- keit der zuvor simulierten Energie und speichern Sie die Anzahl unter dem Key NumberOfHits. Nutzen Sie hierzu die von Ihnen implementierte Polarmethode, um die Normalverteilung zu rea- lisieren.

[7]:
```
numberofhits = np.zeros(10**5)
for index, value in enumerate(energy):
    numberofhits[index] = np.random.normal(loc=10*value, scale=2*value)
    while numberofhits[index] < 0:
        numberofhits[index] = np.random.normal(loc=10*value, scale=2*value)

numberofhits = np.round(numberofhits)
```

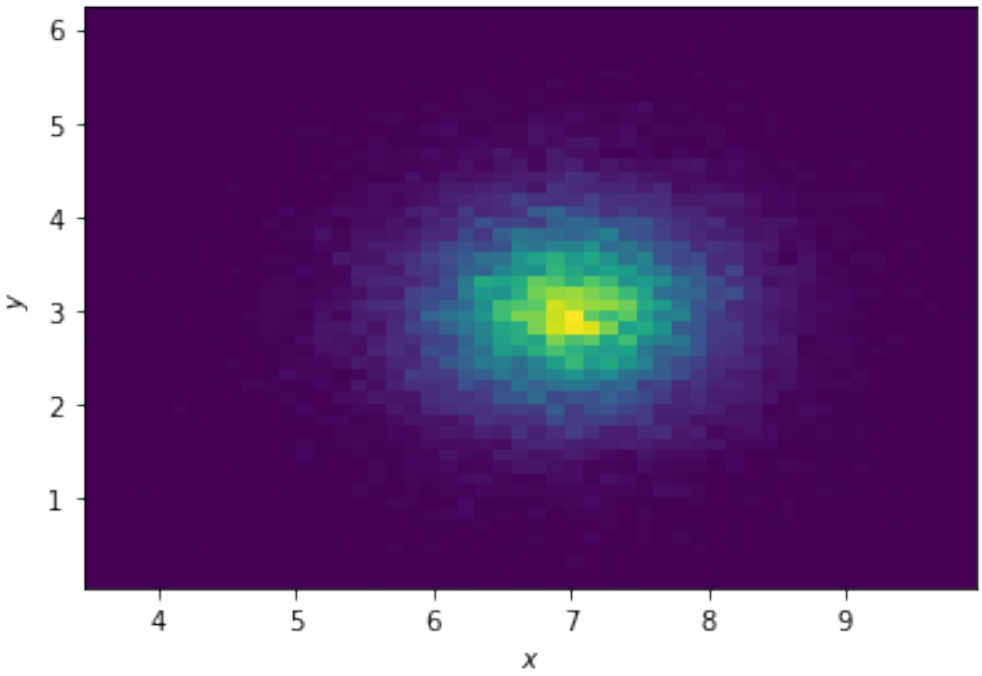
2.2 (e) Ortsmessung

Betrachten Sie im Folgenden einen quadratischen Flächendetektor mit der Kantenlänge 10 Längeneinheiten. Das Signal trifft am Punkt (7,3) auf den Detekor. Die Ortsauflösung ist wiederum energieabhängig. Simulieren Sie die Orte zu den zuvor erzeugten Ereignissen, indem Sie sowohl für die x- als auch für die y-Richtung eine Normalverteilung annehmen. Hierbei ist energieab- hängig und gegeben durch = 1 , log 10 (N + 1) wobei N die zuvor bestimmte Anzahl der Hits ist. Achten Sie hier wiederum darauf, dass die gezogenen Ereignisse innerhalb des Detektors liegen (ggf. neue Zufallszahlen ziehen). Speichern Sie die Koordinaten der Orte unter den Keys x und y. Stellen Sie die erhaltenen Orte in einem zweidimensionalen Histogramm dar.

[8]:
```
x = np.zeros(10**5)
y = np.zeros(10**5)
  = 1/np.log10(numberofhits+1)

for index, value in enumerate(energy):
    x[index] = np.random.normal(loc=7, scale= [index])
    while x[index] < 0 or x[index] > 10:
        x[index] = np.random.normal(loc=7, scale= [index])
    y[index] = np.random.normal(loc=3, scale= [index])
    while y[index] < 0 or y[index] > 10:
        y[index] = np.random.normal(loc=3, scale= [index])

#weil die selbstgeschriebene Methode nicht ganz funktioniert, wurde die numpy␣
 ↪funktion verwendet
#code für eigene funktion:
#gen = Generator(seed=0)
#for index, value in enumerate(energy):
#    x[index] = gen.normal(loc=7, scale= [index])[0]
#    while x[index] < 0 or x[index] > 10:
#        x[index] = gen.normal(loc=7, scale= [index])[0]
#    y[index] = gen.normal(loc=3, scale= [index])[1]
```

```
#     while y[index] < 0 or y[index] > 10:
#         y[index] = gen.normal(loc=3, scale= [index])[1]
```

[9]:
```
plt.hist2d(x[acceptancemask.nonzero()],y[acceptancemask.nonzero()], bins = 50)
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')

None
```



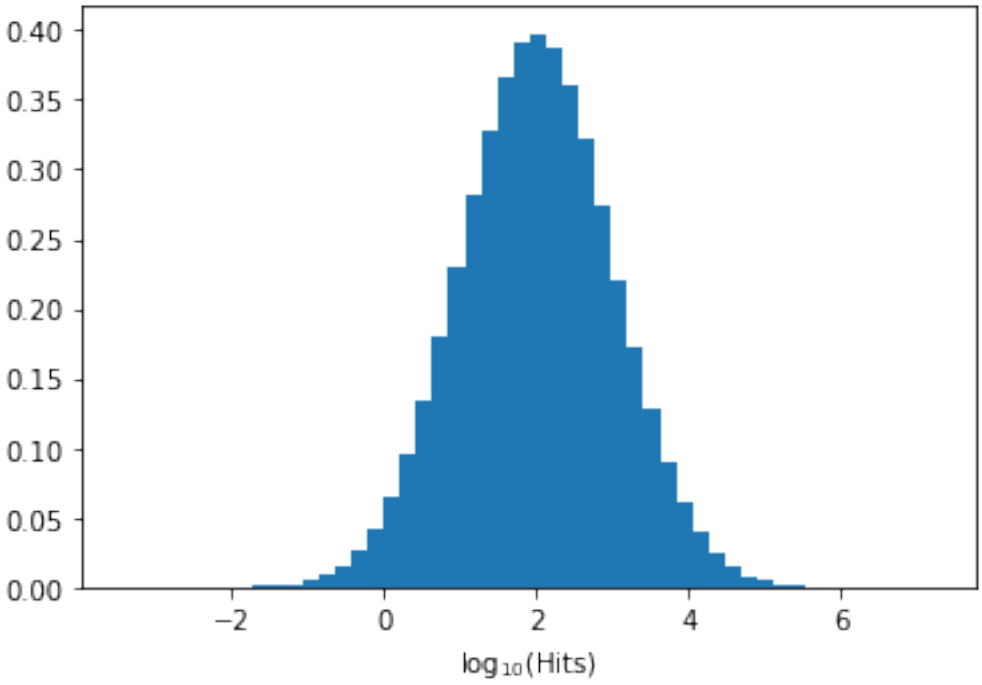
[10]:
```
data = {"Energy": energy, "EcceptanceMask": acceptancemask, "NumberOfHits":␣
 ↪numberofhits, "x": x, "y": y}
signal = pd.DataFrame(data = data)
signal.to_hdf('NeutrinoMC.hdf5', key='Signal', mode='w')
```

3 f) Untergrund MC

Die Zahl der erwarteten Untergrund-Ereignisse ist groß im Verhältnis zum erwarteten Signal. Erzeugen Sie einen neues DataFrame mit den Keys NumberOfHits, x und y. Simulieren Sie 10^7 Untergrund-Ereignisse mit folgenden Eigenschaften: • Der Zehner-Logarithmus der Anzahl der Hits folgt einer Normalverteilung mit =2und =1. • Die - bzw. -Koordinaten der Ereignisse sind um den Mittelpunkt des Detektors normalverteilt. Hierbei ist =3. • Zwischen der - und der -Koordinate besteht eine Korrelation von =0.5. Stellen Sie die Orte der Untergrundereignisse in einem zweidimensionalen und den Logarithmus der Anzahl der Hits in einem eindimensionalen Histogramm dar. Würfelt man standardnormalverteilte Zufallszahlen bzw. , so ergeben sich die

normalverteilten Zufallszahlen bzw. mit beliebigem , und Korrelationskoeffizientem aus der Transformation: =√1− 2 + +

[12]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

#np.random.seed(44) #seed macht das Ganze reprduzierbar
size = int(1e7)

mu = 2
sig = 1

log10hits = np.random.normal(mu, sig, size)
hits = [ int(10**x) for x in log10hits ]

dfBackHits = pd.DataFrame({'NumberOfHits': hits}) #Hits der Untergrundereignisse
%matplotlib inline
plt.hist(log10hits, bins=50, density=True, label=r'Untergrund')
plt.xlabel(r'$\log_{10}{(\mathrm{Hits})}$')
#plt.legend()
#plt.tight_layout()
plt.savefig('f_hits.pdf')

None
```



[13]:
```
def ort(mu, sig, rho, size):
    x=[] #leere Liste wird erzeugt
    y=[]
    while len(x)<size:
        xx = np.random.normal(0, 1)
        yy = np.random.normal(0, 1)

        xx = np.sqrt(1-rho**2)*sig*xx+rho*sig*yy+mu
        yy = sig*yy+mu

        if 0 <= xx <= 10 and 0 <= yy <= 10:
            x.append(xx) #xx wird an die leere Liste angehängt
            y.append(yy)

    return x, y

mu = 5
sig = 3
rho = 0.5

x, y = ort(mu, sig, rho, size)
%matplotlib inline
plt.grid()
plt.hist2d(x,y, bins=[100,100], range=[[0,10],[0,10]], cmap='inferno')
plt.colorbar()
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')

plt.savefig('f_detektor.pdf')
plt.clf()

#Der Code braucht Ewigkeiten um durchzulaufen!! Woran liegt das?
```

```
<Figure size 432x288 with 0 Axes>
```

[14]:
```
dfBackX = pd.DataFrame({'x': x})
dfBackY = pd.DataFrame({'y': y})

dfBackground = pd.concat([dfBackHits, dfBackX, dfBackY], axis=1)
dfBackground = dfBackground.to_hdf('NeutrinoMC.hdf5', key='Background',mode='a'␣
↪)
```

Der zweite Teil des Codes aus f) braucht Ewigkeiten um Durchzulaufen! (Woran liegt das?) Deswegen wurde der fertige Plot extra eingefügt

[ ]: