ABLA Reader

Kenny Haak

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics and used resources of the Facility for Rare Isotope Beams (FRIB) Operations, which is a DOE Office of Science User Facility under Award Number DE-SC0023633, and by the US National Science Foundation under Grants No. PHY-20-12040 and 23-10078 “Windows on the Universe: Open Quantum Systems in Atomic Nuclei at FRIB”.

Outline

Initialization – Slide 3

Event File Processing – Slide 8

Dump File Processing – Slide 14

Initialization

Required Package Versions

- Ensuring the same environment is important, this code functions with:
  - Python version 3.9.10
  - Pandas version 1.3.5
  - Numpy version 1.21.6
  - Matplotlib version 3.5.3

```
import matplotlib as mpl
import pandas as pd
import numpy as np
import sys

print(mpl.__version__)
print(pd.__version__)
print(np.__version__)
sys.version
```

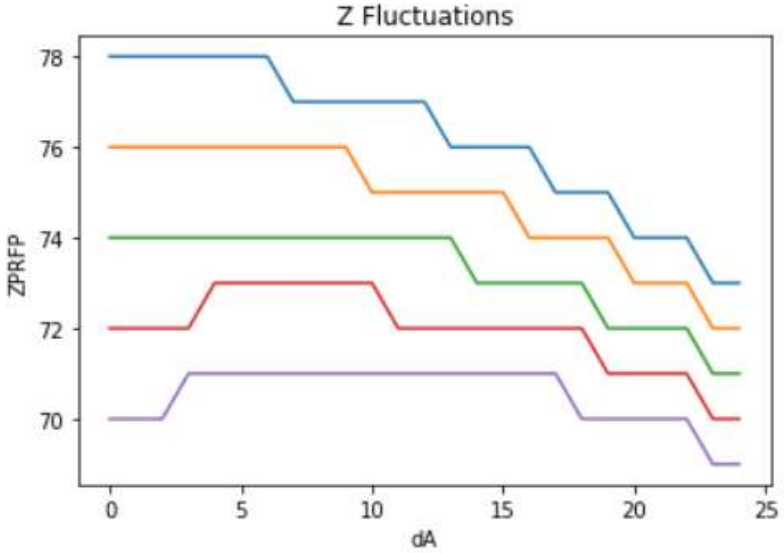
```
3.5.3
1.3.5
1.21.6

'3.9.10 (tags/v3.9.10:f2f3f53,
```

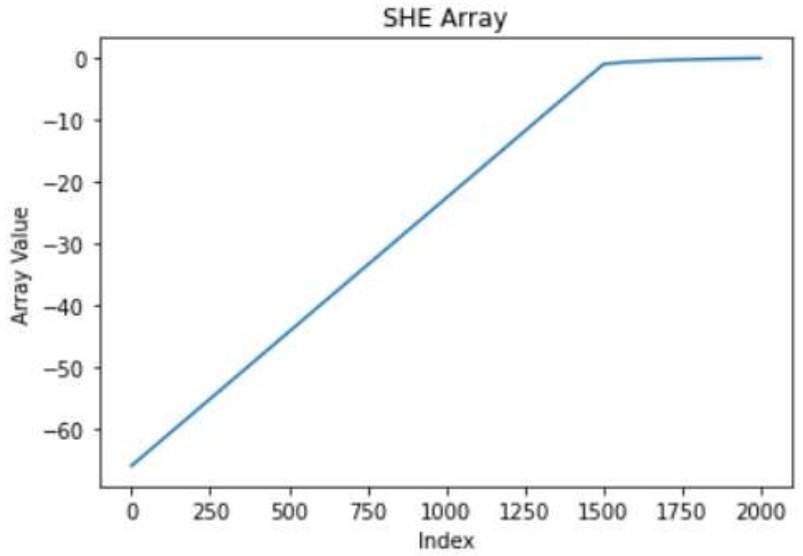
Quick Foreword

- You should be able to make this code usable for future projects if you understand how to use matplotlib and pandas well
- The ABRABLA program provides two mainly useful files to analyze
  - Event File
  - Dump File
- The Event file contains A LOT of information so it is usually much smaller in number than dump file (you can't leave on the event switch and run 5 million events without crashing the computer)
  - Useful for diagnostics and understanding underlying physics
- The dump file is for when you want to calculate cross sections
  - Useful for making rate predictions for future experiments

Initial Cells

- The first couple cells aim to elucidate the nature of a couple key variables in the ABRABLA program
- Reference the ABLA documentation in I:\projects\lisedev\Projects\Abrabla for more





Mass Table

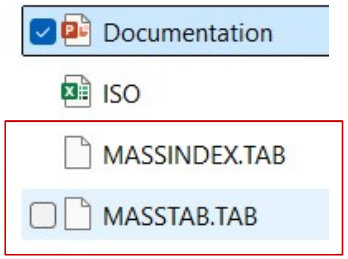
- This cell allows you to look up the Mass Excess for any nuclide given A,Z

```
#Mass TABLE interpretor

with open(f"MASSINDEX.tab") as f:
    dump = f.readlines()
```

```
    ITAB = I1 + N - I2
    print(f"ITAB index for {A}{PT[Z]}: {int(ITAB)}")

    return(TAB.loc[ITAB][0])

print('Mass Excess: ',look_up(186,72))
```

```
ITAB index for 186Hf: 2525
Mass Excess:  -1486.485
```

- These are said to be calculated with Liquid Drop Model, but there are inconsistencies between LISE LDM and the ABLA LDM
- With this code you can interpret the built in mass table of ABLA (files MASSINDEX and MASSTAB)
- If you look at the source code of this cell you can also learn how the data structure works so you can make your own mass table and replace the file for ABLA to update it mass excess values



Event File Processing

Event File Processing

- With the event switch turned on (see documentation @ I:\projects\lisedev\Projects\Abrabla), the LMD extension file will be generated
- The fourth cell in the ABLA_reader notebook will convert this file to a DataFrame object and **write to a CSV file for you**
- Several acronyms are explained here in the comments
- You need to set the path and the filename to provide the correct file for processing

```
#Event file processing (lmd extension)

# FTYPE = Reaction type
# PRFP = Pre-Fragment Particle
# IMODE = Fission type
# LCP = Light charged particle
# IMF = Intermediate mass fragment
# FP = Final Particle
```

```
# path = "I:\\projects\\lisedev\\Projects\\Abrabla\\ABRABLA07_Intel\\out\\"
file = 'A198Z78_Ni.lmd'
```

Count DataFrame & Parameter Histograms

- This cell is useful for just quickly looking at the number of counts of each product for a sanity check

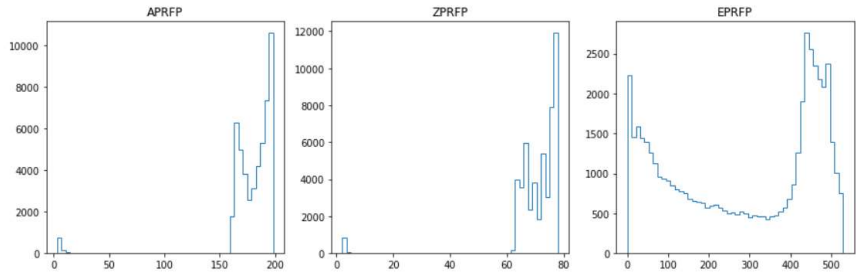
In [7]:
```
#Quick organization to show a couple of nearest fragments to beam (Cold Frag

index = sorted([Z for Z in set(df['ZFP1'])])
index.reverse()
name = sorted([N for N in set(df['N'])])

cDF = pd.DataFrame(columns=name,index=index)

for Z in index:
    for N in name:
        cDF.loc[Z,N] = df[(df['ZFP1']==Z) & (df['N']==N)].shape[0]

cDF[:10]
```

Out[7]:

	1	2	4	5	6	8	62	63	64	65	...	111	112	113	114	115	116	117	118	119	120
78	0	0	0	0	0	0	0	0	0	0	...	167	326	340	514	497	606	460	635	667	0
77	0	0	0	0	0	0	0	0	0	0	...	288	362	385	495	348	461	362	374	261	377
76	0	0	0	0	0	0	0	0	0	0	...	196	236	160	153	104	102	57	41	21	16
75	0	0	0	0	0	0	0	0	0	0	...	76	75	42	42	16	14	13	3	2	0
74	0	0	0	0	0	0	0	0	0	0	...	11	10	6	5	1	1	1	0	0	0
73	0	0	0	0	0	0	0	0	0	0	...	4	1	1	0	0	0	0	0	0	0
72	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
71	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
70	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
69	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

- This cell will give a quick idea of the distribution of certain parameters

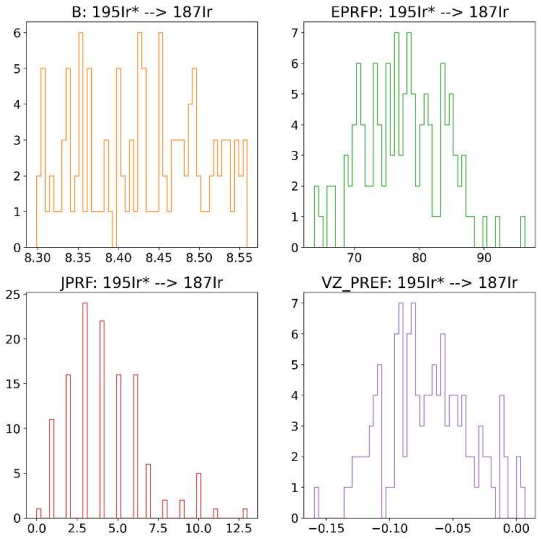
In [8]:
```
# A quick look at histograms of chosen parameters

BINS=50

k=1
plt.figure(figsize=(15,15))
for F in ['APRFP','ZPRFP','EPRFP','JPRF','VZ_PREF','VX_ABRA','VY_ABRA','VZ_ABRA']:
    counts,bins = np.histogram(df[F],bins=BINS)

    plt.subplot(3,3,k)
    plt.title(F)
#     plt.yscale('log')

    plt.stairs(counts,bins)
    k+=1
```



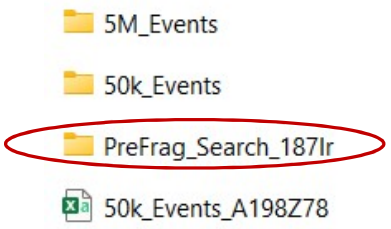
Pre-Fragment Search

```
# Pre-fragment Search and Display Parameters (B, EE*, Angular Momentum, Z-direction Velocity)
plt.rcParams.update({'font.size': 16})

BINS=50
Z=77
N=110
```

Adjust your prefragment and # of bins here



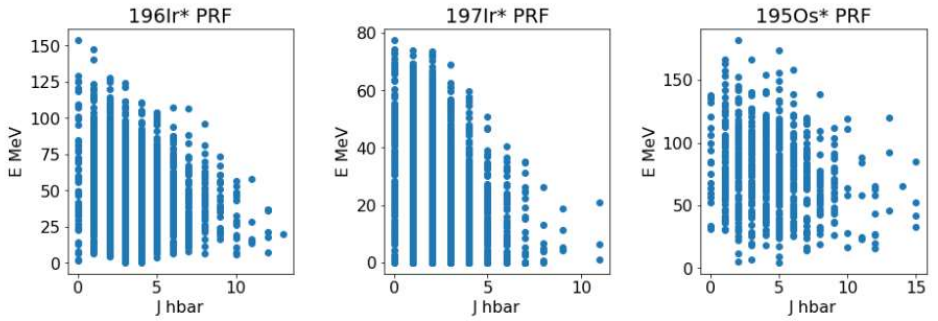
- The next cell allows you to quickly search for all prefragments for a given final fragment and plot the histograms of chosen parameters
- The default parameters are
  - Impact parameter
  - Excitation energy
  - Angular momentum
  - Z-direction velocity
- After this cell is done running, check your running directory for a file that says PreFrag_Search



Parameter Comparisons

- The next cell shows a 2d scatter plot comparison of two parameters of interest (histogram would be better but I didn't make that)
- You can change the Z and N you are looking at
- And you can change the parameter

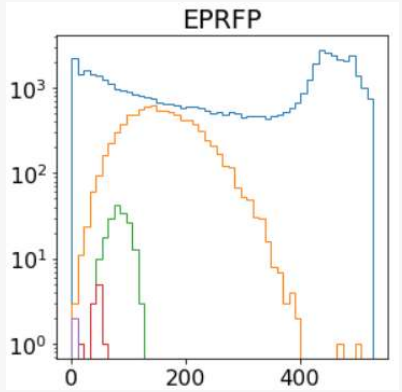
```
for Z in [77,76,75,74]:
    for N in [119,120]:

        df_FILTER = df[(df['ZPRFP']==Z) & (df['APRFP']==(Z+N))]

        plt.subplot(2,4,k)

        plt.scatter(df_FILTER['JPRF'],df_FILTER['EPRFP'])
        plt.title(f"{N+Z}{PT[Z]}* PRF")
        plt.xlabel('J hbar')
        plt.ylabel('E MeV')

        k+=1

# plt.savefig('J_vs_E.png')
```



Gating

- This cell is documented in: I:\projects\lisedev\Projects\Abrabla)
- This cell is for visualizing gates within gates within gates
- You can set gate boundaries manually
- Or they can be calculated
- You can also choose between gating on specific final fragments vs parameter values
- The cell below has various calculation methods for assigning optimal gate values

```
BINS=50

#Combo, lower B bound from A, higher B bound from Z
bmin = df[(df['AFP1'] > 170)]['B'].min()
bmax = df[(df['ZFP1'] > 70)]['B'].max()
#~~~~~~~~~~
bmin = 7.2
bmax = 8.2
EEmin = 3
EEmax = 7

# N=120
# df_FILTER = df[(df['ZFP1']==76) & (df['N']==N)]
# df_FILTER3 = df[(df['ZFP1']==75) & (df['N']==N)]
# df_FILTER4 = df[(df['ZFP1']==74) & (df['N']==N)]
# df_FILTER5 = df[(df['ZFP1']==73) & (df['N']==N)]

df_FILTER = df[(df['B'] > bmin) & (df['B'] < bmax)]
df_FILTER3 = df[(df['ZFP1'] < 75)  & (df['AFP1'] > 181)]
df_FILTER4 = df[(df['ZFP1'] < 75)  & (df['AFP1'] > 181) & (df['N-2Z'] > -35)]
df_FILTER5 = df[(df['B'] > bmin) & (df['B'] < bmax) & (df['EPRFP'] < EEmax) & (df['EPRFP'] > EEmin)]
```




```
#Gate on b as function of mass (A)
bmin = df[(df['AFP1'] > 181)]['B'].min()
```

```
#Combo, lower bound from A, higher bound from Z
bmin = df[(df['AFP1'] > 181)]['B'].min()
```

```
#Gate on b as function of charge (Z)
bmin = df[(df['ZFP1'] > 69)]['B'].min()
```

```
#Min Max from Manual Selection of Isotopes along the 'Edge'
bmin = bmin #Actual was 6.8463, not helpful
```

Dump File Processing

Initial Processing

- This first cell will read in the dump file and organize it into arrays/lists
- You can set the folder and file path here
- There is a list of acronyms

```
#Initialize Read-in of Cross Section Results file (The 'dump' file)

folder = "./5M_Events/"
file = 'ALL_A198Z78.dmp'
PATH = folder+file
```

```
# BU => Break-up
# IMF => Intermediate Mass (Fragment)
# LP => Light Particle
# MF => Multi-Fragment?
```

- The output helps you understand how many different decay channels were used (Cumulative Group = 7). And you can also see how much of the data file was empty by comparing iso groups to non-empty iso groups

```
Cumulative Groups: 7
Isotope Groups: 780
Non-Empty Iso-Groups: 448
```

Data Formatting

- The data must then be formatted and sorted to be entered into an excel sheet
- There is two excel sheets made
  - Cumulative
  - Isotope
- Cumulative is just the sum of all XS for a given element for a given channel
- Isotope has all the individual production cross sections

```
#####################################  CUM CALC ############
dfLI = []
for i,c in enumerate(cumul):
    name = c[0].split(':')[0]
    col = [c[2].strip(' '),c[3].replace(' ','')]
    df = pd.DataFrame(np.zeros([1,len(col)]),columns=col)
```

```
#####################################  ISO CALC ####
li_DF=[]
for i,xt in enumerate(XS_Types):
    df = pd.DataFrame(np.zeros([1,3]),columns=['Z',
```

Additional Calculations – MIN CALC

- The dump file only gives you a raw output value for each isotope produced, but this number doesn't tell you how many events of that isotope were made
- You can back calculate by dividing the raw output by the minimum value (it is hopefully equal to 1 count)
- However, I noticed it is possible for the smallest value to be a multiple of the minimum increment, and there for the MIC CALC section was made to look for smallest common denominator
- This code works for the sample data provided but had issues with other ABLA results, so it is currently commented out

- If you want to have
  - **Number of events**
  - **Probability**
  - **"Corrected" cross section value (Probability X Total CS)**
- You can uncomment the MIN CALC section and this **one specific line in the ISO CALC**

```
################################  ISO CALC
li_DF=[]
for i,xt in enumerate(XS_Types):
    df = pd.DataFrame(np.zeros([1,3]),columns=

    for I in ISO:
#         print(I,'\n\n\n\n')
        if xt in I[0]:
            data = I[5].strip('\n').strip(' ')

            for d in data:
                df.loc[df.shape[0]] = WS_conver

    if 0 in df['Raw_Out']:
        df = df[df['Raw_Out'] != 0]
#     df_format(df,MIN=MIN)
    li_DF.append(df)
    li_DF[i].name = xt.strip('(') #Store ident
```

```
Avaliable increments to choose from:
0.1380484101078819
0.32188193387792596
10.461817082605517
0.0013084631458452275
0.0006542315729226138
0.007850778875071365
3558.2621565375744
```

Additional Calculations – ERR CALC

- Another calculation that can be performed is on the intrinsic error that comes from round the lowest common denominator
- The worst-case scenario is 50% relative error, but usually it is much less than this
- This code will check all decay channels and find the max error for each and then show the closest value to the 0.5 rounding error

```
#####################################  ERR CALC ##########################################
ERR=[]
for df in li_DF:
    remain = df['Raw_Out']/MIN % 1 #Excess after integer
    err = np.abs(np.abs(remain - 0.5).min() - 0.5) #closest to middle
    ERR.append(err)
print('Error check...\nLargest remainders per group:')
for E in ERR:
    print(E)
print(f'Closest to 0.5: {round(max(ERR),5)}')
print(f"Fragmentation events: {li_DF[5]['events'].sum()}")
#########################################################################################
```

```
Error check...
Largest remainders per group:
0.0007575827139305602
0.0007632613378518727
0.0007567078681987027
0.0007592895931486797
0.000761956942596953
0.0007560513040516526
0.0007637066340748788
0.0007079234087690178
0.000761956942596953
0.0007602072510621838
Closest to 0.5: 0.00076
Fragmentation events: 16072288
```