# ### Code for the construction of the distribution of the energy lost by the ion

#

# ##### Intellectual Property (IP) Statement:

# ##### This document are the result  of the work conducted by the undersigned author. All intellectual property rights associated with the content herein remain with with the author and the Institution MedAustron. Any use, reproduction, or distribution must be authorized by the owners of the property rights."

# ##### Giulio Magrin,  5 September 2025

# In[1]:

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

import os

import datetime

import statistics

import random

from scipy import signal

from scipy.interpolate import interp1d

from scipy.optimize import fsolve

# #### Functions and settings

# In[2]:

# Functions

#Function to check if "element" is not zero

def is\_not\_zero(element):

    return element != 0

#Function to compact a long list (data) in a shorter list (reduced)

# whose elements are the averages of successive groups of n elements (n=compactor) of the old list

def reduce\_list\_by\_average(data, n):

    if n <= 0:

        raise ValueError("Factor 'n' must be greater than 0.")

    if len(data) < n:

        raise ValueError("List is too short for the given factor.")

    reduced = [

        sum(data[i:i+n]) / n

        for i in range(0, len(data) - n + 1, n)

    ]

    return reduced

# #Function to compact a long list (old\_list) in a shorter list (new\_list)

# # whose elements are the averages of successive groups of n elements (n=compactor) of the old list

# def compact\_of\_list\_elements (old\_list,compactor):

#     new\_list = []

#     for i in range(0, len(old\_list), compactor):

#         compacted\_elements = old\_list[i:i+compactor]

#         average = sum(compacted\_elements) / len(compacted\_elements)

#         new\_list.append(average)

#     return new\_list

# Output files

e = datetime.datetime.now()

fn1=str(e.strftime("%Y-%b-%d"))     #filename

# INITALIZING THE SPECTRA REPOSITORY

allFinalConvComp=[]

allFinalConvComp3=[]              #Construction of the distribution of the energy in the individual collision

                                  #by the ion and the electron

#Working directory

directory = os.getcwd()

# print(directory)

# #### INPUTS

# In[3]:

#Physics constants

e\_ele=1.6022e-19    #this is a CONSTANT: charge of electron in C

c\_l=299792458       #this is a CONSTANT: speed of light in m/s

m\_ele=0.5109906     #this is a CONSTANT: mass of electron in MeV

m\_ele\_MKS=m\_ele\*1e6\*e\_ele/c\_l\*\*2

epsilon\_0=55.263    #this is a CONSTANT: epsilon0 in e^2 GeV-1 fm-1

N\_A=6.022E+23       #Avogadro's number

k\_eps=1/(epsilon\_0\*np.pi\*4)   #value of 1/(4π·є0)

k\_eps\_MKS=1/(epsilon\_0\*np.pi\*4)/e\_ele\*1E-6

# Numerical precision and collision probability

prec1=17            #this provides the number of decimal digits used in the calculation of the convolution

f0=np.exp(-1/1024)  # Probability of no collision in chosen thickness

                    # which is 1/1024 of the thickness corresponding on average to 1 collision.

f01=1-f0            # Probability of collision

#Ion type and energy

projectile="carbon-ion"

# projectile="proton"

if projectile=="proton":

    rest\_mass=938.27

    z\_ion=1

if projectile=="carbon-ion":

    rest\_mass=11193.41

    z\_ion=6

Ek\_u=25       # Kinetic energy in MeV/u

# #### Projectile parameter

# In[ ]:

# #### Target parameters

#

# In[4]:

#### Target

target="silicon"

# target="diamond"

# target="water"

# thick=0.010            #thickness of the target in micron

# diameter=0.010           #diameter of the target in micron

thick=250      #thickness of the target in micron

diameter=1  #diameter of the target in micron

if target=="silicon":

    Z\_target= 14        #atomic number of silicon target

    A\_target= 28        #mass number of silicon target

    W\_value=3.62        #w-value in eV per pair of silicon

    ro\_target=2.32      #density of target in g/cm^3

    ro\_t\_MKS=ro\_target\*1000

    band\_gap=1.12       #this is an INPUT: material band gap eV

    band\_gap\_MKS=band\_gap\*e\_ele

elif target=="diamond":

    Z\_target= 6                 #atomic number of diamond target

    A\_target= 12                #mass number of diamond target

    W\_value=13.4                #w-value in eV per pair of diamond

    ro\_target=3.52              #density of target in g/cm^3

    ro\_t\_MKS=ro\_target\*1000

    band\_gap=5.5               #this is an INPUT: material band gap eV

    band\_gap\_MKS=band\_gap\*e\_ele

elif target=="water":

    Z\_target= 3.33                 #atomic number of water target

    A\_target= 6                #mass number of water target

    W\_value=34.6                #w-value in eV per pair of water

    ro\_target=1              #density of target in g/cm^3

    ro\_t\_MKS=ro\_target\*1000

#     band\_gap=78               #this is an INPUT: material ionization potential

    band\_gap=6.9             #changed 7 Sept 2023, This is an experimental value, 7.3 is calculated see REFERENCE:

                             #Fang C, Li WF, Koster RS, Klimeš J, Van Blaaderen A, Van Huis MA.

                             #The accurate calculation of the band gap of liquid water by means of GW corrections applied to plane-wave density functional theory molecular dynamics simulations.

                             #Physical Chemistry Chemical Physics. 2015;17(1):365-75.

                             #Using instead ionization potential would strongly enlarge the final distributions

    band\_gap\_MKS=band\_gap\*e\_ele

print ("Material: {}; Z={}; A={}; W-value={}eV; Density={}g/cm³; Band gap={}eV; Thickness: {}µm".format(target, Z\_target, A\_target, W\_value,ro\_target,band\_gap,thick))

# #### Electronic Stopping Power (calculated from the tables)

# In[5]:

# Load stopping power data

column\_names = ['Energy',"siliconR","siliconSP","diamondR","diamondSP","waterR","waterSP"]

StPow=pd.read\_csv(r"..\inputFiles\StPowRange\_"+projectile+".csv", names=column\_names, sep=",") #colling the stopping power table

# Extract energy and stopping power for the selected target

# (for diamond, the values from ICRU are converted from graphite and using for diamond the density 3.52)

ref\_ene=StPow['Energy']

ref\_ene0=ref\_ene.tolist()

ref\_StPow=StPow[target+'SP'] # "siliconSP" refers to the silicon stopping power

ref\_StPow1=ref\_StPow.tolist()

# Interpolate stopping power at desired energy

y\_interp = interp1d(ref\_ene0, ref\_StPow1)

StPow=y\_interp(Ek\_u\*1000)

print("The {} stopping power (real density) in {} at energy {} MeV/u is {} keV/µm".format(projectile,

                                                                                          target,Ek\_u,np.around(StPow,2)))

# #### Construction and optimization of the single collision distribution

# In[6]:

#CONSTANTS and PARAMETERS CORRELATED

# Relativistic parameters

Ek=Ek\_u\*12                                       #the kinetic energy of the carbon ion in MeV

Et=(Ek+rest\_mass)                                #total energy, relativistic

momentum=np.sqrt(Et\*\*2-rest\_mass\*\*2)             #momentum, relativistic

gamma=Et/rest\_mass                               #gamma

beta=np.sqrt(1-(1/gamma)\*\*2)                     #beta

# Impact parameter limits

b\_max=k\_eps\_MKS\*z\_ion\*(e\_ele\*\*2)/(c\_l\*beta)\*np.sqrt(2/(m\_ele\_MKS\*band\_gap\_MKS))

b\_min=k\_eps\_MKS\*z\_ion\*(e\_ele\*\*2)/(c\_l\*\*2\*beta\*\*2\*gamma\*m\_ele\_MKS)

# Maximum energy transfer in a collision

emaxM=(2\*m\_ele\*beta\*\*2\*gamma\*\*2)/(1+2\*gamma\*m\_ele/rest\_mass+(m\_ele/rest\_mass)\*\*2)

emax=emaxM\*1000000

# Collision density

n\_col=np.pi\*N\_A\*1000\*Z\_target/A\_target\*ro\_t\_MKS\*(b\_max\*\*2-b\_min\*\*2)

                                    #this is the mean number of primary collisions in 1 meter

n\_col\_um=round(n\_col/1000000)       #this is the mean number of  primary collisions in 1µm

# Mean energy per primary collision

delta1=StPow/n\_col\_um\*1000          # in eV

#########################################################

#Procedure to find the displacement "dev", within the bin, where the distribution w(e) should be calculated

#for a correct eveluation of the first momentum of the energy, delta1;

#"k10" is the new normalization factor of the collision distribution calculated considering the displacement'dev'

#'k10' and 'dev' are the solutions of the system of two equations

#The first equation imposes that the collision distribution is still normalized to 1

#

#The second equation imposes that the first moment coincedes with the calculated one based on the theory but

# this is done considering a very low probability of collision (f01) considering a thickness 1024 times smaller

# than the thickness (let us call it d1) corresponding to one average  collision).

# This ensure that the there are no distorsion in the distribution w(e)=k10\*1/e^2 due to overlaps of two collisions.

# Therefore this second equation is constructed considering the thickness of d1/1024,

# where the distribution k10\*1/e^2 is reduced by the probability f01 and the expected energy is delta1/1024

# Solve system to find normalization factor k0 and minimum energy emin

# in the collision distribution.

def f(x):

    f0=x[0]\*(np.log(emax)-np.log(x[1]))-delta1   #analytical condition: delta1 is the first moment of the distribution (Eq. 20 in the paper)

    f1=x[0]\*(emax-x[1])-emax\*x[1]                #analytical condition: k0 to obtain a normalized collision distribution: k0(1/emax-1/emin)=1

    return np.array([f0,f1])

x0 = np.array([1,1])

x = fsolve(f,x0)

k0 = x[0]                                          # k0 is a first normalization factor

emin=x[1]                                        # minimum energy used in the collision distribution

                                                 # the values 'emin' and 'k0' are similar as can be seen from 'f1'

# Estimate number of delta rays (per real thickness in microns) and convolutions

n\_pair\_um=round(StPow\*1000/W\_value)

n\_reit=np.log2(n\_col\_um\*thick)

reiter1=int(10+np.floor(n\_reit))

n\_bins=int(emax/emin)

print("Minimum energy value considered in the collision distribution (eV)=",round(emin,4),

      "\nMaximum energy value in the collision distribution (eV)=",round(emax,2),

      "\nNumber of delta rays created per micron at the density of the material =",int(n\_pair\_um),

      "\nNumber of delta rays created in primary collisions per micron at the density of the material =",int(n\_col\_um),

      "\nNumber of convolutions needed: 10 (initial) +",int(round(n\_reit,0)))

# reiter1=10+math.floor(n\_reit)

                        #number of bins of the collision distribution

######################################################

# System of two equations;

# the solutions of the two equations provide the displacement and the new normalization:

def g(y):

    g0=sum([y[0]\*emin/(emin\*i+y[1])\*\*2 for i in range(1,n\_bins+1)])-1                        #normalization to 1 of discrete collision distribution

    g1=sum((y[0]\*emin\*\*2\*i\*f01)/(emin\*i+y[1])\*\*2 for i in range(1,n\_bins+1))-delta1/1024  #forcing delta1 to the first moment of the discrete distribution

    return np.array([g0,g1])

y0=np.array([0.0,0.5])

y=fsolve(g,y0)

k10=y[0]\*1             #new normalization

dev=y[1]\*1             #relative displacement where the distribution w(e) is calculated

print("\nNumber of bins in the collision distribution =",n\_bins,

      "\nNormalization coefficient used before introducing the bin displacement=",round(k0,4),

      "\nBin displacement =",round(dev,4),

      "\nNormalization coefficient used after introducing the bin displacement =",round(k10,4))

# Construction of the initial COLLISION DISTRIBUTION of a thickness corresponding to 1/1024th of the single-collision thickness

# Two tests are performed, the first to check that the distribution if normalized,

# the second to check that the first moment coincides with the value calculated from the input parameters

w\_e=[k10\*f01\*emin/(emin\*i+dev)\*\*2 for i in range(1,n\_bins+1)]

w\_e.insert(0,f0)   #Adding the element f0 which represents the probability at energy zero.

# Validate normalization and first moment

s\_w\_e=sum(w\_e)

e\_w\_e=[emin\*i\*j for i,j in zip(range(0,len(w\_e)),w\_e[0:])] #This is the integrand for calculating delta1, the increment "de" is already present in w\_e.

s\_e\_w\_e=sum(e\_w\_e)  #This sum is delta1/1024

print(f"Check normalization (expected 1) = {s\_w\_e}: {'PASSED' if round(s\_w\_e, 8) == 1 else 'FAILED'}")

print(f"Check first moment (expected {round(delta1,8)}) = {round(s\_e\_w\_e \* 1024,8)}: {'PASSED' if abs(delta1 - s\_e\_w\_e \* 1024) < 1e-7 else 'FAILED'}")

# In[ ]:

# In[9]:

#########################################################

#Procedure to find the displacement "dev", within the bin, where the distribution w(e) should be calculated

#for a correct eveluation of the first momentum of the energy, delta1;

#"k10" is the new normalization factor of the collision distribution calculated considering the displacement'dev'

#'k10' and 'dev' are the solutions of the system of two equations

#The first equation imposes that the collision distribution is still normalized to 1

#

#The second equation imposes that the first moment coincedes with the calculated one based on the theory but

# this is done considering a very low probability of collision (f01) considering a thickness 1024 times smaller

# than the thickness (let us call it d1) corresponding to one average  collision).

# This ensure that the there are no distorsion in the distribution w(e)=k10\*1/e^2 due to overlaps of two collisions.

# Therefore this second equation is constructed considering the thickness of d1/1024,

# where the distribution k10\*1/e^2 is reduced by the probability f01 and the expected energy is delta1/1024

# Solve system to find normalization factor k0 and minimum energy emin

# in the collision distribution.

def f(x):

    f0=x[0]\*(np.log(emax)-np.log(x[1]))-delta1   #analytical condition: delta1 is the first moment of the distribution (Eq. 20 in the paper)

    f1=x[0]\*(emax-x[1])-emax\*x[1]                #analytical condition: k0 to obtain a normalized collision distribution: k0(1/emax-1/emin)=1

    return np.array([f0,f1])

x0 = np.array([1,1])

x = fsolve(f,x0)

k0 = x[0]                                          # k0 is a first normalization factor

emin=x[1]                                        # minimum energy used in the collision distribution

                                                 # the values 'emin' and 'k0' are similar as can be seen from 'f1'

# Estimate number of delta rays (per real thickness in microns) and convolutions

n\_pair\_um=round(StPow\*1000/W\_value)

n\_reit=np.log2(n\_col\_um\*thick)

reiter1=int(10+np.floor(n\_reit))

n\_bins=int(emax/emin)

print("Minimum energy value considered in the collision distribution (eV)=",round(emin,4),

      "\nMaximum energy value in the collision distribution (eV)=",round(emax,2),

      "\nNumber of delta rays created per micron at the density of the material =",int(n\_pair\_um),

      "\nNumber of delta rays created in primary collisions per micron at the density of the material =",int(n\_col\_um),

      "\nNumber of convolutions needed: 10 (initial) +",int(round(n\_reit,0)))

# reiter1=10+math.floor(n\_reit)

                        #number of bins of the collision distribution

######################################################

# System of two equations;

# the solutions of the two equations provide the displacement and the new normalization:

def g(y):

    g0=sum([y[0]\*emin/(emin\*i+y[1])\*\*2 for i in range(1,n\_bins+1)])-1                        #normalization to 1 of discrete collision distribution

    g1=sum((y[0]\*emin\*\*2\*i\*f01)/(emin\*i+y[1])\*\*2 for i in range(1,n\_bins+1))-delta1/1024  #forcing delta1 to the first moment of the discrete distribution

    return np.array([g0,g1])

y0=np.array([0.0,0.5])

y=fsolve(g,y0)

k10=y[0]\*1             #new normalization

dev=y[1]\*1             #relative displacement where the distribution w(e) is calculated

print("\nNumber of bins in the collision distribution =",n\_bins,

      "\nNormalization coefficient used before introducing the bin displacement=",round(k0,4),

      "\nBin displacement =",round(dev,4),

      "\nNormalization coefficient used after introducing the bin displacement =",round(k10,4))

# Construction of the initial COLLISION DISTRIBUTION of a thickness corresponding to 1/1024th of the single-collision thickness

# Two tests are performed, the first to check that the distribution if normalized,

# the second to check that the first moment coincides with the value calculated from the input parameters

w\_e=[k10\*f01\*emin/(emin\*i+dev)\*\*2 for i in range(1,n\_bins+1)]

w\_e.insert(0,f0)   #Adding the element f0 which represents the probability at energy zero.

# Validate normalization and first moment

s\_w\_e=sum(w\_e)

e\_w\_e=[emin\*i\*j for i,j in zip(range(0,len(w\_e)),w\_e[0:])] #This is the integrand for calculating delta1, the increment "de" is already present in w\_e.

s\_e\_w\_e=sum(e\_w\_e)  #This sum is delta1/1024

print(f"Check normalization (expected 1) = {s\_w\_e}: {'PASSED' if round(s\_w\_e, 8) == 1 else 'FAILED'}")

print(f"Check first moment (expected {round(delta1,8)}) = {round(s\_e\_w\_e \* 1024,8)}: {'PASSED' if abs(delta1 - s\_e\_w\_e \* 1024) < 1e-7 else 'FAILED'}")

# In[ ]:

# #### EXECUTE ALL THE SELF-CONVOLUTIONS - NO Threshold

# In[26]:

# Initialize convolution list

#(EXECUTE ALL  SELF\_CONVOLUTIONS foressen to match the tickness)

listofconvolutions2=[]

#convolution weights and energy weights

c\_w\_e = list(w\_e)

e\_w\_e = [(emin\*i)\*j for i,j in zip(range(1,len(c\_w\_e)),c\_w\_e[1:])]

print(("Un-convoluted:  number of bins used = {}; mean energy = {}").format(len(c\_w\_e),np.around(sum(e\_w\_e),6)))

doubled\_energy = 2 \* sum(e\_w\_e)

# Perform iterative convolutions

for i in range(reiter1):

    start\_time=datetime.datetime.now()

    c\_w\_e=signal.fftconvolve(c\_w\_e, c\_w\_e, mode='full').tolist() #added fft, deleted , method='direct'

#     c\_w\_e=c\_w\_e.tolist()

    # Substitute near-zero values with zeros and double the lenght with zeros

    max\_p\_bef=c\_w\_e.index(max(c\_w\_e))

    a=np.around(c\_w\_e, decimals=prec1)

    c\_w\_e=list(filter(is\_not\_zero, a))

    max\_p\_aft=c\_w\_e.index(max(c\_w\_e))  #Index of the maximum value after the removal

#     c\_w\_e=[0 for x in range(max\_p\_bef-max\_p\_aft)]+c\_w\_e #replacing the zero terms at the left side of the list

#     c\_w\_e=c\_w\_e+[0 for x in range(len(c\_w\_e))]  #so cf0x is double the lenght of convxx

    c\_w\_e=[0] \* (max\_p\_bef-max\_p\_aft) + c\_w\_e

    c\_w\_e += [0] \* len(c\_w\_e)  #so cf0x is double the lenght of convxx

    # Recalculate energy

    e\_w\_e=[(emin\*i)\*j for i,j in zip(range(1,len(c\_w\_e)),c\_w\_e[1:])]

    listofconvolutions2.append(list(c\_w\_e))

    s\_e\_w\_e=sum(e\_w\_e)

    end\_time=datetime.datetime.now()

    elapsed=end\_time-start\_time

    print(f"Convolution #{i+1}: bins = {len(c\_w\_e)}; mean energy = {np.around(s\_e\_w\_e, 6)}; "

          f"δ1·#collisions = {np.around(doubled\_energy, 6)} (computed in {np.around(elapsed.total\_seconds(), 2)} s)")

    doubled\_energy=2\*doubled\_energy

#Test of good chooce of precision comparing energies

# At each convolution, the value of the mean energy loss in collisions must double.

# Globally, the mean energy loss calculated from the new convoluted distribution must equals the energy calculated

# from the mean energy δ1 of a single collision multiplied by the total number of collisions

# If the two values coincide, the presision "prec1" is acceptable,

# otherwise the precision "prec1" is not high enough and must be increased.

# #### Determination of which of the solutions above must be combined to represent the energy lost in the site

#

# In[25]:

# Calculate total number of collisions in the target thickness

n\_col\_thick = n\_col\_um \* thick

n\_col\_thick\_bin = []

ctr = int(0)

tempV1 = n\_col\_thick\*1

# Convert to binary representation for convolution selection

while(tempV1 > 0):

    n\_col\_thick\_bin.append(int(tempV1 % 2))    #the symble %  provides the reminder of the modulo 2

#     print(n\_col\_thick\_bin)

    tempV1 = int(tempV1/2)

    ctr += 1

print(f"The single collision distribution will be convoluted {int(n\_col\_um\*thick)} times")

saaa=[i\*(j+10) for i,j in zip(n\_col\_thick\_bin,range(len(n\_col\_thick\_bin)))]

saaa2=[i\*2\*\*j for i,j in zip(n\_col\_thick\_bin,range(len(n\_col\_thick\_bin)))]

# print (saaa)

# print ("sum",saaa2,"=",sum(saaa2))

print("The distribution for the total thickness is reached convoluting {w} elements out of the last {z} convolutions \n(in total {v} self convolutions were calculated), \naccording to the sequence \n{y}\n{xx} = {xy}\nto obtain the energy of {x} keV"

      .format(w=sum(n\_col\_thick\_bin),x=round(StPow\*thick,2),y=n\_col\_thick\_bin,z=reiter1-9,v=reiter1,xx=saaa2,xy=sum(saaa2)))

##################################################################################################################

# Select and combine convolutions based on binary representation

# The result of the computation is the frequency

# is the distribution of energy TRANSFER

finalConv=[]

select\_conv=[]

convDist=list(listofconvolutions2[0])

for i in range(len(n\_col\_thick\_bin)):

    #convDist=convDist+[0 for x in range(len(listofconvolutions2[i])-len(convDist))]  #so cf0x is double the lenght of convxx

    if n\_col\_thick\_bin[i]==1 :

        start\_time=datetime.datetime.now()

        conv\_conv=signal.fftconvolve(convDist,listofconvolutions2[9+i], mode='full') #signal.fftconvolve(c\_w\_i, c\_w\_i, mode='full')

        convDist=list(conv\_conv)

        select\_conv.append(convDist)

        e\_w\_e=[x\*y\*emin for x,y, in zip(convDist,range(0,len(convDist)+1))]

        s\_e\_w\_e=sum(e\_w\_e) ###this is equivalent to y\_F

        end\_time=datetime.datetime.now()

        elapsed=end\_time-start\_time

        print(f"Including convolution #{10+i}: bins = {len(convDist)}; mean energy (eV) = {np.around(s\_e\_w\_e, 2)} (computed in {np.around(elapsed.total\_seconds(), 2)} s)")

finalConv=list(convDist)

# #### Spectral representation of the results

# In[29]:

# Prepare thickness labels

coll\_thick = 1 / n\_col\_um

thicknesses = [coll\_thick \* 2\*\*i for i in range(len(listofconvolutions2))]

thickness\_labels = []

for t in thicknesses:

    if t < 0.1:

        thickness\_labels.append(f"{round(t \* 1000, 1)} nm")

    elif t < 1:

        thickness\_labels.append(f"{int(t \* 1000)} nm")

    elif t < 100:

        thickness\_labels.append(f"{round(t, 1)} µm")

    elif t < 1000:

        thickness\_labels.append(f"{int(t)} µm")

    else:

        thickness\_labels.append(f"{round(t / 1000, 1)} mm")

thickness\_labels = [""] \* 9 + thickness\_labels

# Plot distributions

fig, ax=plt.subplots(figsize=(13,6.5))

plt.margins(x=0, y=0)

jump= 19                # transition to the compaced representation

compact\_plot=100        # factor used to compact the representation

ref\_conv=10              # first distribution to represent

    #This code speeds up the plot separating it in 2 differentparts parts using "jump" as separator

    #one, before the "jump" value, for the first distributions with density

    #the other compacted by a factor "compact\_plot"

for i in range(ref\_conv,jump):

    mlc=1/max(listofconvolutions2[i])

    x\_vals = [j \* emin / (2\*\*(i - 9)) \* n\_col\_um / 1000 for j in range(len(listofconvolutions2[i]))]

    y\_vals = [k \* mlc for k in listofconvolutions2[i]]

    plt.plot(x\_vals, y\_vals, label=thickness\_labels[i])

for i in range(jump, len(listofconvolutions2)):

    density\_reduced = reduce\_list\_by\_average(listofconvolutions2[i], compact\_plot)

    max\_val = max(density\_reduced)

    step\_reduced = [j \* emin/ (2\*\*(i - 9)) \* n\_col\_um / 1000 \* compact\_plot for j in range(len(density\_reduced))]

    normalized = [x / max\_val for x in density\_reduced]

    plt.plot(step\_reduced,normalized,label=thick\_label[i])

plt.legend(loc="upper right")

plt.xscale('log')

plt.xlim([10,1000])

plt.ylim([1E-5, 1.1])

plt.xlabel ("keV·µmˉ¹")

plt.ylabel("Density distributions normalized to unity")

plt.show()

# #### Rossi's style representation

# In[32]:

#THE FIRST SET CORRESPONDS TO THE LONG DISTRIBUTIONS CALCULATED VIA CONVOLUTIONS,

#WE ASSUME THAT THE DISTRIBUTION CORRESPONDING TO THE LAST CONVOLUTIONS IS GAUSSIAN

#THE SECOND PART THEN IS SIMPLY CALCULATED AS GAUSSIANS WHOSE STANDARD DEVIATIONS

#DECREASES BY A FACTOR sqrt(2) AT EACH DOUBLING

#THE PLOTS ARE SEPARATED TO SPEED UP THE ADJUSTMENT

# fig, ax=plt.subplots(figsize=(6,12))

# plt.margins(x=0, y=0)

rand\_dist=[]

rand\_x=[]      # It is created to have identical x-coordinate for each set. The number of sets is given by repetitions\_rand

rand\_y=[]

repetitions\_rand=200    # It defines how many random values are chosen to represent each distribution

n\_rand\_y=[]

for i in range(9,40):                 #listofconvolutions2[i])

    ele\_x = [i-9]\*repetitions\_rand

    rand\_x=rand\_x+ele\_x

        # n\_distribution=[k\*(2\*\*(i)) for k in distribution]

    if i < len(listofconvolutions2):

        sum\_distr=sum(listofconvolutions2[i])

        #step\_reduced=[j\*emin/(2\*\*(i-9))\*n\_col\_um/1000\*compact\_plot for j in range(len(density\_reduced))]

        elements = [j\*emin/(2\*\*(i-9))\*n\_col\_um/1000 for j in range(len(listofconvolutions2[i]))]       #adjust the length of the distributions represent the specific energy dividing by 2 at each step

#         elements = [j/(2\*\*i) for j in range(len(listofconvolutions2[i]))]       #adjust the length of the distributions represent the specific energy dividing by 2 at each step

        distribution = [abs(k/sum\_distr) for k in listofconvolutions2[i]]

    #     distribution = [abs(k\*mlc) for k in listofconvolutions3[i]]

        rand\_val=np.random.choice(elements, repetitions\_rand, p=distribution)   #array of n random values (n:repetitions\_rand) in each distribution

        rand\_val=rand\_val.tolist()

        # rand\_dist.append(rand\_val)              #the lists representing each distributions are put together in a single list

        add\_y=[j\*(2\*\*(i)) for j in(rand\_val)]   #

        rand\_y=rand\_y+add\_y

        print("convoluted ",i,end='\r')

        #############

        rand\_val=np.random.choice(elements, repetitions\_rand, p=distribution)  #array of n random values (n:repetitions\_rand) in each distribution

    #     n\_rand\_val=[k\*(2\*\*i) for k in rand\_val]

# For visualization purposes, a randon value random.random() is added to separate values in the graph

# which would otherwise overlap

        n\_rand\_val=[k+random.random() for k in rand\_val]

#         n\_rand\_val=[k\*emin for k in rand\_val]

        # n\_rand\_y=n\_rand\_y+n\_rand\_val

        n\_rand\_y=n\_rand\_y+n\_rand\_val

        sigma = statistics.stdev(n\_rand\_val)

        # plt.plot(rand\_x,n\_rand\_y, c='k', marker=0,markersize=20,fillstyle='full',linestyle='None')

#THE SECOND PARTS ASSUMES THAT THE CUMULATIVE CONVOLUTIONS ARE FINALLY WELL REPRESENTED BY A GAUSSIAN. THIS

# SIMPLIFIES THE REPRESENTATION OF THE DESTRIBUTION WHEN THE THICKESS DOUBLES:

# IT IS A GAUSSIAN WITH FWHM WHICH IS RELATIVELY DECREASED BY A FACTOR sqrt(2)

    else:

# mean and standard deviation

        mean\_rand\_y=np.mean(n\_rand\_val)

        sigma = statistics.stdev(n\_rand\_val)/(2\*\*(i+1-len(listofconvolutions2)))

        s = np.random.normal(mean\_rand\_y, sigma, repetitions\_rand)

        s=s.tolist()

        n\_rand\_y=n\_rand\_y+s

        print("estimated ",i,"   ", end='\r')

# #### Rossi's style spectra

# In[34]:

plot\_max=repetitions\_rand\*reiter1

fig, ax=plt.subplots(figsize=(6,12))

plt.margins(x=0, y=0)

plt.plot(rand\_x[:plot\_max],n\_rand\_y[:plot\_max], c='k', marker=0, markersize=11, fillstyle='full',linestyle='None')  #

# plt.xscale('linear')

# plt.yscale('linear')

# plt.xlim([0.001,0.3])

plt.ylim([0, 1000])

plt.xlabel('log₂ m/m₀', size=15)

plt.ylabel('E/m (a.u.)', size=15)

plt.show()