09/08/20 08:50:21 /home/ana/Documents/uni/PHS3000/code/betarays.py

```
# PHS3000
# Betarays - Radioactive decay of Cs - 137
# Ana Fabela, 08/09/2020
import monashspa.PHS3000 as spa
from scipy.interpolate import interp1d
import numpy as np
import pandas as pd
import pytz
import matplotlib
import matplotlib.pyplot as plt
from pprint import pprint
import scipy.optimize

plt.rcParams['figure.dpi'] = 150

# Globals
hbar = 1.0545718e-34 # [Js]
c = 299792458 # [m/s]
mass_e = 9.10938356e-31 # [kg]
eV = 1.602176634e-19 # [J]
MeV = 1e6 * eV
keV = 1e3 * eV
rel_energy_unit = mass_e * c**2 # to convert SI into relativistic or viceversa


# Desintegration energy
# Cs-137 disintegrates by beta minus emission to the excited state of Ba-137 (94.6 %)
theory_T = 0.5120 * MeV
theory_T_rel = theory_T / rel_energy_unit
theory_w_0_rel =  theory_T_rel + 1
p_0_rel = np.sqrt(theory_w_0_rel**2 - 1)

data = spa.betaray.read_data(r'beta-ray_data.csv')

def csv(data_file):
    # extracting valid data from csv file
    j = 0
    for row in data:
        # print(f'{j=}')
        if row[0] == pd.Timestamp('2020-08-18 10:26:00+10:00', tz=pytz.FixedOffset(360)):
            valid_data = data[j:]
            continue
        j+=1

    background_count_data = []
    count = []
    lens_current = []
    u_lens_current = []
    for row in valid_data:
        if row[3] == 'Closed':
            # print(row[3])
            background_count_data.append(row)
            continue
        count.append(row[5])
        lens_current.append(row[6])
        u_lens_current.append(row[7])

    # make lens current an np.array
    lens_current = np.array(lens_current)
    return background_count_data, count, lens_current, u_lens_current


def correct_count(background_count_data):
```

```python
    # correcting our data by removing avg background count and adjusting it for spectrometer resolution (3%)
    background_count = []
    for row in background_count_data:
        background_count.append(row[5])
    avg_background_count = np.mean(background_count)
    # print(f"We want to subtract the background count from our data {avg_background_count=}")
    # calculating fractional uncertainty in total background count (delta_t = 24 min)
    total_background = np.sum(background_count)
    u_avg_background_count = np.sqrt(total_background) / 4

    # uncertainty in the corrected count
    background_corrected_count = count - avg_background_count

    ##################################################################################
    # I chose the uncertainty in the count to be 15 counts
    u_background_corrected_count = np.sqrt(15**2 + u_avg_background_count**2)
    ##################################################################################

    # As per Siegbahn [9] correction for spectrometers resolution
    correct_count = background_corrected_count / lens_current
    u_correct_count = correct_count * np.sqrt((u_background_corrected_count / background_corrected_count)**2 + (u_lens_current / lens_current)**2)

    # print(f'\n{total_background=:.0f}')
    # print(f'{avg_background_count=:.0f}')
    # print(f'{u_avg_background_count=:.0f}')

    # print(f'\ncorrect counts:{correct_count[8:18]}')
    # print(f'uncertainty:{u_correct_count[8:18]}')

    return correct_count, u_correct_count

def compute_k(lens_current):
    # Finding constant of proportionality in p = kI
    # calibration peak (K) index of k peak is i=20
    T_K = 624.21 * keV / rel_energy_unit
    I_k = lens_current[20]
    k = np.sqrt((T_K + 1)**2 - 1) / I_k
    # defining appropriate uncertainty for our k peak
    u_I_k = 0.1 / 2
    u_k = k * (u_I_k / lens_current[20])
    # print(f'{T_K=:.3f} mc^2')
    # print(f'{k=:.3f}')
    # print(f'{u_I_k=:.3f}')
    # print(f'{u_k=:.3f}')
    return k, u_k

def compute_p_rel(lens_current, k, u_k):
    # The momentum spectrum (relativistic units)
    p_rel = k * lens_current
    u_p_rel = p_rel * np.sqrt((u_k / k)**2 + (0.0005 / lens_current)**2)
    dp_rel = p_rel[1]-p_rel[0]
    # print(f'\np :{p_rel[8:18]}')
    # print(f'uncertainty:{u_p_rel[8:18]}')
    # print(f'dp :{dp_rel}')
    return p_rel, u_p_rel, dp_rel

def interpolated_fermi(p_rel):
    # G = (p_rel * F(z=55, ,w_rel)) / w_rel
    fermi_data = spa.betaray.modified_fermi_function_data
    return interp1d(fermi_data[:,0], fermi_data[:,1], kind='cubic')(p_rel)

def compute_w(p_rel):
    # KURIE/Fermi PLOT
```

```python
    w_rel = np.sqrt(p_rel**2 + 1) # relativistic energy units
    u_w_rel = u_p_rel[8:18]
    # print(f'\n{w_rel=}')
    # print(f'{u_w_rel=}')
    return w_rel, u_w_rel


def f(x, m, c):
    # linear model for optimize.curve_fit()
    return m * x + c

def compute_S_n(x, opt_w_n, u_opt_w_n):
    # shape factor from Siegbahn
    S_n = x**2 - 1 + (opt_w_n - x)**2
    u_S_n = np.sqrt((2 * u_x * x)**2 + (2 * np.sqrt(u_opt_w_n**2 + u_x**2) * (opt_w_n -
x))**2)
    return S_n, u_S_n

def LHS(S_n, u_S_n):
    # left hand side of our linearised relation
    y = np.sqrt(correct_count[8:18] / (p_rel[8:18] * x *
interpolated_fermi(p_rel[8:18]) * S_n))
    u_y = (y / 2) * np.sqrt((u_correct_count[8:18] / correct_count[8:18])**2 + (2 *
(u_p_rel[8:18] / p_rel[8:18])**2) + (u_interpolated_fermi /
interpolated_fermi(p_rel[8:18]))**2 + (u_S_n / S_n)**2)
    return y, u_y

def optimal_fit(f, x, y, u_y):
    # linear fit
    # unpack into popt, pcov
    popt, pcov = scipy.optimize.curve_fit(f, x, y, sigma=u_y, absolute_sigma=False)
    # To compute one standard deviation errors on the parameters use
    perr = np.sqrt(np.diag(pcov))

    # optimal parameters
    opt_K_2, opt_intercept = popt
    u_opt_K_2, u_opt_intercept = perr
    # print(f"\noptimised gradient {opt_K_2:.3f} ± {u_opt_K_2:.3f}")
    # print(f"optimised intercept {opt_intercept:.3f} ± {u_opt_intercept:.3f}")

    optimised_fit = f(x, opt_K_2, opt_intercept)
    # uncertainty in linear model f given optimal fit
    u_f = np.sqrt((x * u_opt_K_2)**2 + (u_opt_intercept)**2)
    # return optimal parameters
    return opt_K_2, opt_intercept, u_opt_K_2, u_opt_intercept, optimised_fit, u_f

def iterative_solve(x, w_n, u_w_n):
    # using our results to find T
    T = (w_n - 1) * rel_energy_unit

    # print("\nHenlo, this is the start of the while loop")
    while True:
        old_T = T
        S_n, u_S_n = compute_S_n(x, w_n, u_w_n)
        yn, u_yn = LHS(S_n, u_S_n)
        K_2, intercept, u_K_2, u_intercept, optimised_fit, u_f = optimal_fit(f, x, yn,
u_yn)

        # using our results to find new w_n
        w_n = intercept / - K_2
        u_w_n = np.sqrt((u_K_2 / K_2)**2 + (u_intercept / intercept)**2) * w_n

        # new T in SI units
        T = (w_n - 1) * rel_energy_unit

        # print(f"T = {T / MeV} MeV")
```

```python
        # print(f"old_T = {old_T / MeV} MeV\n")

        if abs(T - old_T) < 1e-10 * MeV:
            break
    # print("\nthis is the end of the while loop, yay bai.")

    u_T = (w_n - 1) * u_w_n / w_n * rel_energy_unit
    return T, u_T, yn, u_yn, optimised_fit, u_f

def compare(T, u_T):
    # comparison to theory
    diff = 0.512 * MeV - T
    how_many_sigmas = diff / u_T
    print(f"\nEXPECTED RESULT T = {theory_T / MeV :.3f} MeV")
    print(f"(optimised) T = {T / MeV:.3f} ± {u_T / MeV:.2f} MeV")
    # print(f"difference {diff:.3f}")
    print(f"number of σ away from true result: {abs(how_many_sigmas):.3f}")




########################### Calling our functions ###########################

# open, read and dissect data file
background_count_data, count, lens_current, u_lens_current = csv(data)

# find constant k
k, u_k = compute_k(lens_current)

# correct background count (accounting for background and resolution (3%))
correct_count, u_correct_count = correct_count(background_count_data)

# find momentum spectrum
p_rel, u_p_rel, dp_rel = compute_p_rel(lens_current, k, u_k)




# our sliced data linearised
x, u_x = compute_w(p_rel[8:18])

# uncertainty in interpolated fermi
u_interpolated_fermi = np.sqrt((u_p_rel[8:18] / p_rel[8:18])**2 + (u_x / x)**2) * interpolated_fermi(p_rel[8:18])

# LINEARISED KURIE WITH RESOLUTION CORRECTION
y = np.sqrt(correct_count[8:18] / (p_rel[8:18] * x * interpolated_fermi(p_rel[8:18])))
u_y = (y / 2) * np.sqrt((u_correct_count[8:18] / correct_count[8:18].clip(min=1))**2 + (2 * (u_p_rel[8:18] / p_rel[8:18])**2) + (u_interpolated_fermi / interpolated_fermi(p_rel[8:18]))**2)

# first order fit
opt_K_2, opt_intercept, u_opt_K_2, u_opt_intercept, optimised_fit, u_f = optimal_fit(f, x, y, u_y)
# using our parameters to find opt_w_0
opt_w_0 = opt_intercept / - opt_K_2
u_opt_w_0 = np.sqrt((u_opt_K_2 / opt_K_2)**2 + (u_opt_intercept / opt_intercept)**2) * opt_w_0
print(f" w_0 = {opt_w_0 * rel_energy_unit / MeV:.3f} MeV")
print(f"u(w_0) = {u_opt_w_0 * rel_energy_unit / MeV:.3f} MeV")

# ITERATIVE ANALYSIS using Shape factor (higher order fits)
T, u_T, yn, u_yn, optimised_fit, u_f = iterative_solve(x, opt_w_0, u_opt_w_0)

# final comparison to theoretical value T = 0.512 MeV
compare(T, u_T)
```

```python
############################ plots ############################

# OPTIMISED FIT PLOT and residuals plot
plt.figure()
plt.errorbar(
            x, yn, xerr=u_p_rel[8:18], yerr=u_yn,
            marker="None", linestyle="None", ecolor="m",
            label=r"$y = (\frac{n}{p w G})^{\frac{1}{2}}$", color="g", barsabove=True
)
plt.plot(
        x, optimised_fit, marker="None",
        linestyle="-",
        label="linear fit"
)
plt.fill_between(
                x, optimised_fit - u_f,
                optimised_fit + u_f,
                alpha=0.5,
                label="uncertainty in linear fit"
)
plt.title("linear fit for Kurie data")
plt.xlabel(r"$w [mc^{2}]$")
plt.ylabel(r"$\left ( \frac{n}{p w G} \right )^{\frac{1}{2}}$", rotation=0, labelpad=18)
plt.legend()
spa.savefig('Kurie_linear_data_plot.png')
# plt.show()

residuals = optimised_fit - yn
plt.figure()
plt.errorbar(
            x, residuals, xerr=u_p_rel[8:18], yerr=u_f,
            marker="o", ecolor="m", linestyle="None",
            label="Residuals (linearised data)"
)
plt.plot([x[0], x[-1]], [0,0], color="k")
plt.title("Residuals: linear fit for Kurie data")
plt.xlabel(r"$w [mc^{2}]$")
plt.ylabel(r"$\left ( \frac{n}{p w G} \right )^{\frac{1}{2}}$", rotation=0, labelpad=18)
plt.legend()
spa.savefig('linear_residuals_Kurie_linear_data.png')
# plt.show()
############################ plots ############################
```