PyLab - Radioactive Decay

PHY224 Lab 2

Fredrik Dahl Bråten, Pankaj Patil

October 12, 2021

1 Exercise 2: Nonlinear fitting methods I

1.1 Abstract

In this exercise, we measure and model the rates of emission over time from the radioactive decay of Barium (Ba-137m). We have chosen two models, one where the emission rate $I(t) = ae^{bt}$, and one where $\ln(I(t)) = at + b$, where a and b are parameters of each model to be fitted to our data, t is time, and ln is the natural logarithm. After fitting these models to our data, we graphically plot and compare these model curves along with the theoretical curve of emission rate, and our data with error bars. To evaluate the quality of our models, we calculate and discuss the χ^2_{red} values of our models and data, and confirm that the theoretically predicted half-life of Ba-137m lies within the estimated half life with uncertainty of our two models. This analysis was done in Python by use of the numpy, scipy and matplotlib modules.

1.2 Introduction

In this exercise, we measure and model the rates of emission over time caused by radioactive decay from a sample of Barium (Ba-137m). The corresponding theoretically predicted relationship between emission rate (I) and time (t) is:

$$I(t) = I_0 e^{-t/\tau} = I_0 (\frac{1}{2})^{\frac{t}{t_{\frac{1}{2}}}}$$

Where I_0 is the initial emission rate, τ is the mean lifetime of the isotope, and $t_{\frac{1}{2}}$ is the half-life of the isotope. In our experiment, t is the independent variable, and I is our dependent variable.

We are modeling this relationship by using two models, one where the emission rate $I(t) = ae^{bt}$, and one where $\ln(I(t)) = at + b$, where a and b are parameters of each model to be fitted to our data, t is time, and ln is the natural logarithm.

1.3 Methods, Materials and Experimental Procedure

We successfully followed the procedures as described in the exercise2-NL-fit.pdf [1] document.

The points of data for which we based this analysis on, was downloaded from Quercus as instructed by the TAs. The uncertainties of the measured data were calculated as described in the exercise2-NL-fit.pdf [1] document.

1.4 Results

In Appendix Figure 1 and 2, we see our data from the experiment plotted as points with corresponding error bars. Furthermore, we see the theoretical curve as described in the introduction, along with the curves corresponding to our two models best fitted to our data.

The estimated optimal parameters with uncertainty by scipy optimize curve fit are:

[a,b] = [-0.00394 3.75] ± [0.00091 0.30] for the linear model, and

[a,b] = [4.4+01 -4.1e-03] ± [1.4e+01 1.0e-03] for the non linear model.

The χ^2_{red} values for the nonlinear and linear model, respectively, are 0.0069 and 0.0076.

The Half-life of the Barium, predicted by the non-linear model, is: 170.0 ± 42.0 seconds.

The Half-life of the Barium, predicted by the linear model, is: 176.0 ± 41.0 seconds.

1.5 Discussion

The χ^2_{red} values we found were very low. This means that our models fit our data very well. Thus, the Euclidian distance between the data points and our curves is in general very low. However, this is not necessarily a good sign. Our χ^2_{red} values should ideally both be equal to one. That we have extremely low χ^2_{red} values implies that we do not have enough data. It means that we are in risk of overfitting our models to our data.

The non-linear regression method gave a half-life closer to the expected half-life of 2.6 minutes = 156 seconds. 170 seconds is closer to 156 seconds, than 176 seconds, see half-life results in the Results section. Do however note that the theoretical half life falls within both of our models' estimates of the half-lives, with associated uncertainties.

Though it is hard to distinguish the two models in the non-linear plot, in the logarithmic, linear plot, you can more easily see that the non-linear model returns a curve closer to the theoretical curve than the curve returned by the linear model. Both models do however fit the theoretical curve quite well. Furthermore, both models are well within the uncertainties of our measurements, see Figure 1 and 2 in the Results section.

1.6 Conclusions

In this exercise we estimated the emission rate caused by radioactive decay of Barium (Ba-137m). We successfully followed the instructions for the experiment written in the exercise2.pdf document without issues. Though both of our models of emission rate over time returned quite similar curves, the non-linear model returned a curve closest to the theoretical curve. We have plotted our data with error bars, our models, and the theoretical curve in a normal plot, and a plot with logarithmic y-axis. Furthermore, we have calculated and discussed each models χ^2_{red} value, and confirmed that the theoretical half-life of Ba-137m lies within each of our models' predicted half-lives with uncertainties.

2 Exercise 5: Random number analysis

2.1 Abstract

The aim of this exercise is to analyze the random radioactive decay in Fiesta dinner plates, which contain low amount of Uranium Oxide. The emissions from Fiesta plates are random and the data for the rate of emission can be recorded using Geiger counter. The analysis on this data is done using the Python modules numpy, scipy and matplotlib. After the analysis, we concluded that the data collected for Fiesta plates obeys the rare event statistics given by Poisson distribution. We will discuss the analysis of the data using the python data analysis modules in this report.

2.2 Introduction

Poisson Distribution is a **discrete probability distribution** that expresses probability of a given number of events in fixed interval of time if the events are known to occur with constant mean rate [2]. The Poisson *probability mass function* is given by

$$P_\mu(n) = e^{-\mu}\frac{\mu^n}{\Gamma(n+1)}$$

Where μ is the expected average counts in measured counting interval and $\Gamma(n+1)$ is Gamma function.

In this exercise, we analysis the radioactive counts from Fiesta dinner plates which contain Uranium Oxide in the glaze. We followed the commonly followed approach in Radioactive decay experiments, of using histograms to analyze the counts data.

For the Fiesta plates the mean count, μ, was found to be 126, and the same for background counts was 2.

2.3 Methods, Materials and Experimental Procedure

We successfully followed the procedures as described in the exercise5.pdf [2] document. The data was downloaded from Quercus for analysis.

2.4 Results

The data collected from Geiger counter was used to plot the histogram for the emission count. Figure 3 shows the histogram for data collected along with the qualitative fit of Poisson probability mass function. The histogram is plotted in normalized mode by setting $density = True$. Gaussian distribution is also added to the same plot.

The measured count for Fiesta plate is corrected by subtracting the mean background emission counts. Figure 3 shows the analysis done for Fiesta plates counts.

The expected average number of counts for Fiesta plate, per counting interval of 20 seconds, was found to be

$$\mu = 126 \quad \text{with} \quad \sigma = \sqrt{\mu} = 11.23$$

We note that the Poisson *pmf* function has value 0.035 around this count. Hence there is 3.5% probability that we will observe 126 counts in 20 second interval.

The expected average number of background counts, per counting interval of 20 seconds, was found to be

$$\mu = 2 \quad \text{with} \quad \sigma = \sqrt{\mu} = 1.41$$

Figure 4 shows the analysis done for background emission counts.

2.5 Discussion

For large data points the Poisson distribution is known to move towards Gaussian distribution, enabling us to treat the variable as continuous rather than discrete. As can be noticed from the plot Figure 3, for the Fiesta plate counts, we note that the Poisson distribution *probability mass function* is almost identical to Gaussian distribution or at least has moved really close. Hence we can say that there are enough data points to treat the mean count as continuous variable and treat this counts distribution as normal distribution.

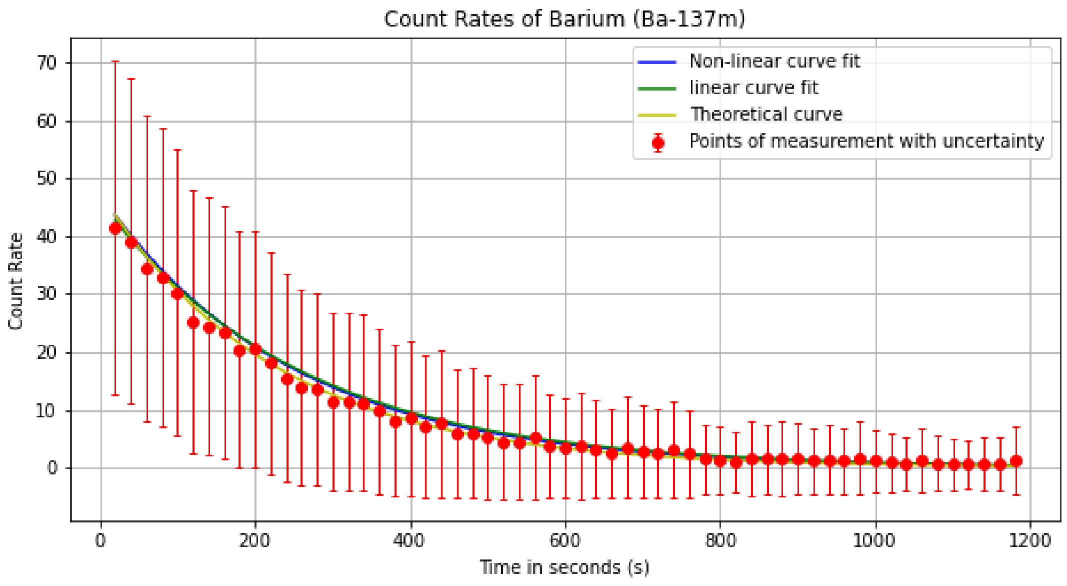
For background counts in plot Figure 4, the Poisson *pmf* is on the left of the Gaussian distribution function. Hence it is safe to conclude that we need more data points for the background counts.

2.6 Conclusions

In this exercise we successfully achieved the aim of the experiment by analyzing random emission counts data from Fiesta plate. We observed that the radioactive decay from Fiesta plates does obey the rare event statistics given by Poisson distribution. The Poisson distribution plotted using the collected data looks almost the same as Gaussian distribution, hence we conclude that we had enough data point for this experiment to treat the emission counts as continuous variable in Gaussian distribution. The same can not be said about the background emission counts.

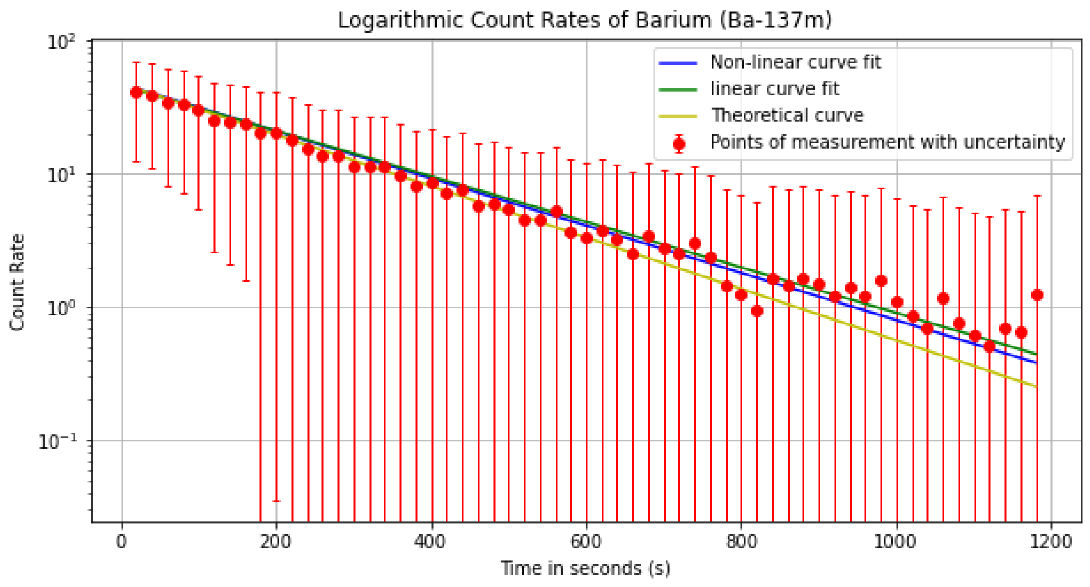
A Appendix

A.1 Plots For Exercise 2



Relationship between count rates and time. We have plotted the points of measurement with corresponding error bars, the curve of theoretical emission rate, and the two models of emission rate best fitted to our data.

Figure 1: Count Rates of Barium (Ba137-m)



Equal to Figure 1, though with a logarithmic y-axis

Figure 2: Logarithmic Count Rates of Barium (Ba137-m)

A.2 Plots For Exercise 5

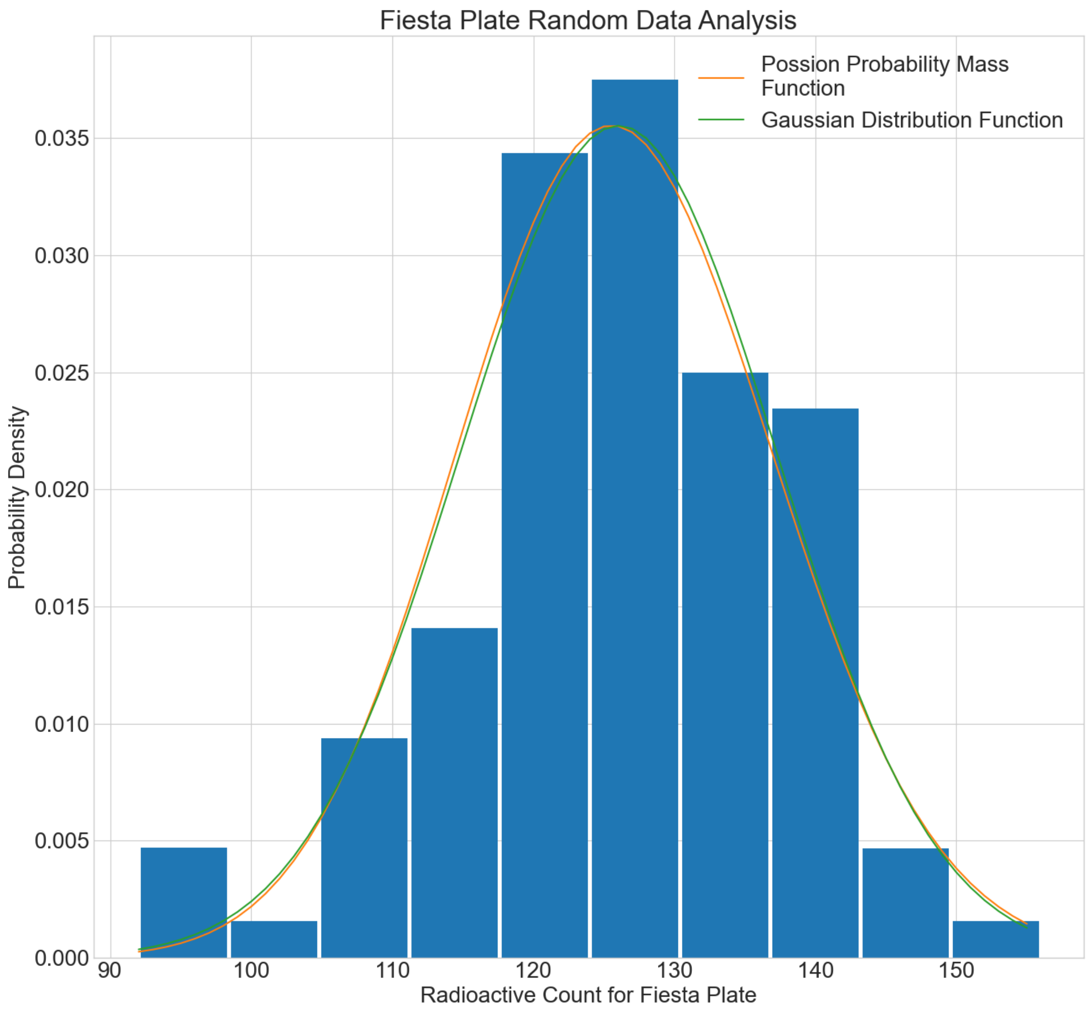


Figure 3: Random Data Analysis For Fiesta Plate Radioactive Counts

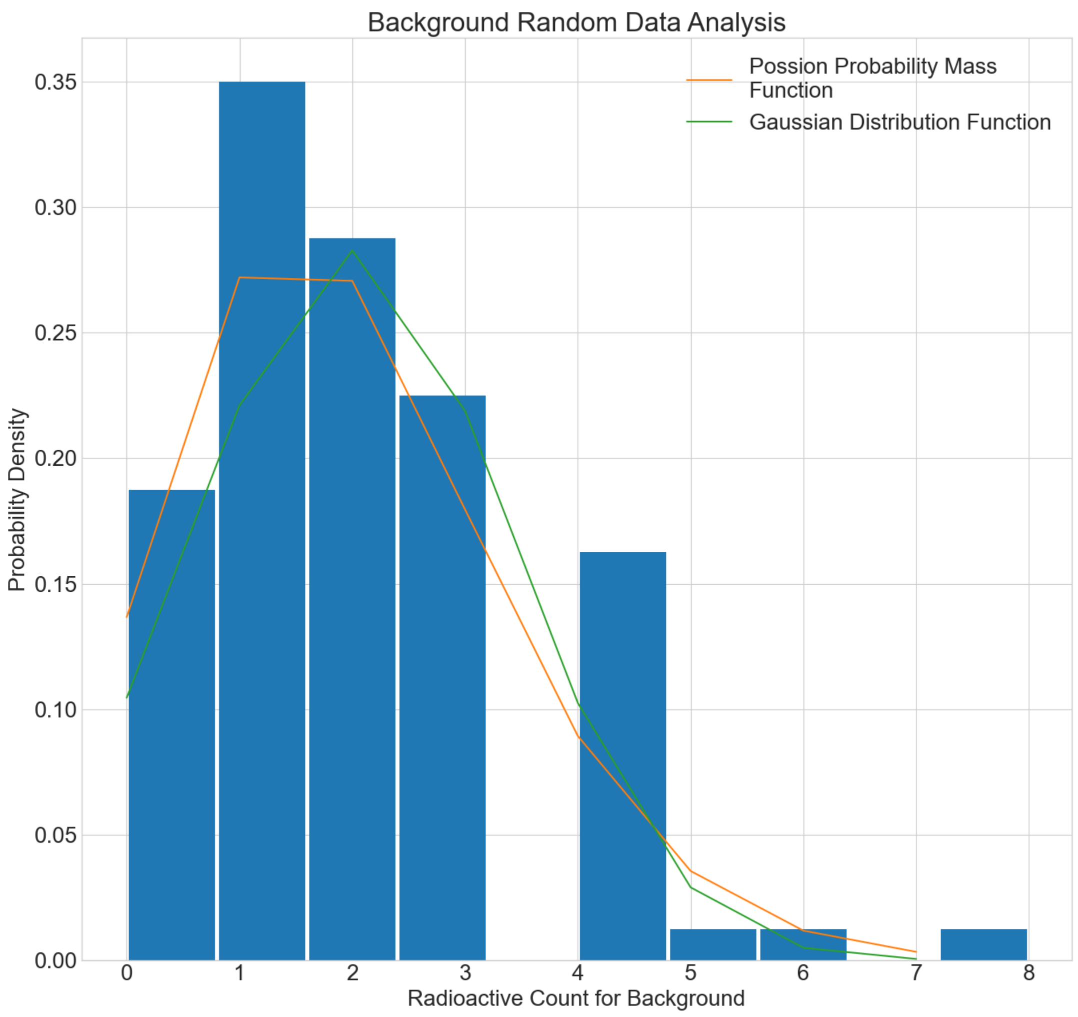


Figure 4: Random Data Analysis For Background Counts

A.3 Python Code: Exercise 2

The Python code for this exercise is divided into two files. Functions.py file contains utility methods which we will be frequently using in this course. Radioactive Decay.py file contains the code which analyzes the data.

A.3.1 Functions.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Sep 30 09:46:36 2021

@author: Fredrik
"""
import numpy as np
import scipy.optimize as optim
import matplotlib.pyplot as plt

#Defining the function for curve fitting and plotting
def curve_fit_and_plot(model,initial_guess,xdata,ydata,y_uncer,xunit,yunit,
                       plot_title):
    """
    This function uses the scipy curve_fit function to estimate the parameters
    of the model which will minimize the euclidian distance between our data
    points, and the model curve.
    We print these optimal model parameters along with their uncertainty, and
    plot the original data with error bars, along with the curve fit model.

    Parameters
    ----------
    model : function to be used as model
        model(x,a,b,c,...), where we are estimating a,b,c, etc.
    initial_guess : list of guesses for the parameters a,b,c, etc. eg. [2,4,254]
    xdata : list of input points for the model, eg. [2,4,5,7,9,28]
    ydata : list of output points for the model, eg [23,25,26,85,95,104]
    y_uncer : list of uncertaintes associated with the ydata, which the model
        shall output
    xunit : String describing the unit along the x-axis for label when plotting.
        eg. 'Voltage (V)'
    yunit : String describing the unit along the y-axis for label when plotting.
        eg. 'Current (A)'
    plot_title : String describing the title of the plot.
        eg. 'Current vs. Voltage'
    Returns None
    """
    #Using the scipy curve fit function to find our model parameters
    p_opt , p_cov = optim.curve_fit(model , xdata , ydata, p0 = initial_guess,
                                  sigma = y_uncer, absolute_sigma = True )
    p_std = np.sqrt( np.diag ( p_cov ))

    print("The optimal values for our curve fit model parameters, are:",np.round(p_opt,2))
    print("Their associated uncertainties  are:", np.round(p_std,2))

    #Now we create some data points on the model curve for plotting
    xvalues_for_plot = np.linspace(xdata[0],xdata[-1],1000)
    yvalues_for_plot = []
    for i in xvalues_for_plot:
        yvalues_for_plot.append(model(i,p_opt[0],p_opt[1]))

    #Now we plot the original data with error bars, along with the curve fit model
    plt.figure(figsize=(10,5))
    plt.errorbar(xdata,ydata,y_uncer,c='r', ls='', marker='o',lw=1,capsize=2,
                 label = 'Points of measurement with uncertainty')
    plt.plot(xvalues_for_plot,yvalues_for_plot, c='b',
             label = 'Scipy curve fit')
    plt.title(plot_title)
    plt.xlabel(xunit)
    plt.ylabel(yunit)
    plt.legend()
    plt.grid()
    plt.savefig(plot_title+'.png')
    plt.show()

    return None

def error_plot(model,p_opt,xdata,ydata,y_uncer,xunit,yunit,
                   plot_title):
    #Now we create some data points on the model curve for plotting
    xvalues_for_plot = np.linspace(xdata[0],xdata[-1],1000)
    yvalues_for_plot = []
    for i in xvalues_for_plot:
        yvalues_for_plot.append(model(i,p_opt[0],p_opt[1]))

    #Now we plot the original data with error bars, along with the curve fit model
    plt.figure(figsize=(10,5))
    plt.errorbar(xdata,ydata,y_uncer,c='r', ls='', marker='o',lw=1,capsize=2,
                 label = 'Points of measurement with uncertainty')
    plt.plot(xvalues_for_plot,yvalues_for_plot, c='b',
             label = 'Scipy curve fit')
    plt.title(plot_title)
    plt.xlabel(xunit)
    plt.ylabel(yunit)
    plt.legend()
    plt.grid()
    plt.savefig(plot_title+'.png')
    plt.show()
    return None

def chi2(y_measure,y_predict,errors):
    """Calculate the chi squared value given a measurement with errors and
    prediction"""
    return np.sum( np.power(y_measure - y_predict, 2) / np.power(errors, 2) )

def chi2reduced(y_measure, y_predict, errors, number_of_parameters):
    """Calculate the reduced chi squared value given a measurement with errors
    and prediction, and knowing the number of parameters in the model."""
    return chi2(y_measure, y_predict, errors)/ \
            (y_measure.size - number_of_parameters)

def read_data(filename, Del, skiprows, usecols=(0,1)):
    """Load give\n file as csv with given parameters,
    returns the unpacked values"""
    return np.loadtxt(filename,
                      skiprows=skiprows,
                      usecols=usecols,
                      delimiter=Del,
                      unpack=True)

def fit_data(model_func, xdata, ydata, yerrors, guess):
    """Utility function to call curve_fit given x and y data with errors"""
    popt, pcov = optim.curve_fit(model_func,
                                 xdata,
                                 ydata,
                                 absolute_sigma=True,
                                 sigma=yerrors,
                                 p0=guess)

    pstd = np.sqrt(np.diag(pcov))
    return popt, pstd
```

A.3.2 Radioactive Decay.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Sep 30 09:47:45 2021

@author: Fredrik
"""
#Importing modules
import numpy as np
import matplotlib.pyplot as plt
import Functions as F


#Defining Constants
Sample_time = 20 #seconds
Experiment_length = 20*60 #Seconds. = 20 min

#Specifying modelling function
def non_linear_model(x,a,b):
    return a*np.exp(b*x)

def linear_model(x,a,b):
    return a*x+b


#Importing data
Sample_number_barium, Number_of_counts_barium = F.read_data('Barium.txt',
                                                            None,2)
Sample_number_bar_background, Number_of_counts_bar_background = F.read_data(
    'Barium_Background.txt', None,2)

#However, background radiation is not measured at the same time as the
# radiation from the Barium. Thus I subtract the mean number of counts from
# the background radiation, from the number of counts for each Barium sample.
Number_of_counts_bar_background = np.mean(Number_of_counts_bar_background)

Number_of_counts = Number_of_counts_barium - Number_of_counts_bar_background

#Now we remove the latter part of the data, where the radiation from the
# barium is at the same level as the background radiation. When this happens
# the Number_of_counts value above may be negative.
list_of_negative_values = []
for i in range(len(Number_of_counts)):
    if Number_of_counts[i]<0:
        list_of_negative_values.append(i)
if len(list_of_negative_values)==0:
    list_of_negative_values.append(-1)

#Now that we know at what index this happen, we shorten all the arrays we will
# use later on
Number_of_counts = Number_of_counts[0:list_of_negative_values[0]]
Number_of_counts_barium = Number_of_counts_barium[0:list_of_negative_values[0]]
Sample_number_barium = Sample_number_barium[0:list_of_negative_values[0]]

#Standard deviation for each point, for derivation, see exercise 2 document.
Count_rate_uncertainty = np.sqrt(
    Number_of_counts_barium + Number_of_counts_bar_background)

Count_rate = Number_of_counts/Sample_time

#Now we fit the parameters of the two models to our data:
popt_linear, pstd_linear = F.fit_data(linear_model,
                                      Sample_number_barium*Sample_time,
                                      np.log(Count_rate),
                                      Count_rate_uncertainty/Count_rate,
                                      [-0.00393908,  3.75408883])



popt_non_linear, pstd_non_linear = F.fit_data(non_linear_model,
                                      Sample_number_barium*Sample_time,
                                      Count_rate,
                                      Count_rate_uncertainty,
                                      [4.36112953e+01, -4.08767785e-03])
print("The estimated optimal parameters with uncertainty by scipy optimize",
      "curve fit are:",popt_linear, "+-",pstd_linear," for the linear, and:",
      popt_non_linear, "+-", pstd_non_linear, "for the non linear model.")

#Calculating predicted y-values of models:
Count_rate_predicted_non_linear = np.zeros(len(Sample_number_barium))
Count_rate_predicted_linear = np.zeros(len(Sample_number_barium))
#And now we also calculate the y-values predicted by the linear model,
# for the non logarithmic scale. I will use this later on in the plot
Count_rate_predicted_linear_non_linear = np.zeros(len(Sample_number_barium))

for i in range(len(Sample_number_barium)):
    Count_rate_predicted_non_linear[i] = popt_non_linear[0]*np.exp(
        popt_non_linear[1]*i*Sample_time)
    Count_rate_predicted_linear[i]= popt_linear[0]*i*Sample_time+popt_linear[1]
    Count_rate_predicted_linear_non_linear[i] = np.exp(popt_linear[1])*np.exp(
        popt_linear[0]*i*Sample_time)
```

```
#The Chi squared values for these models are:
chi2_non_linear = F.chi2reduced(Count_rate,Count_rate_predicted_non_linear,
                         Count_rate_uncertainty,2)
chi2_linear = F.chi2reduced(np.log(Count_rate),Count_rate_predicted_linear,
                         Count_rate_uncertainty/Count_rate,2)
print("\nThe reduced Chi squared values for the non linear and linear",
      "model respectivly",
      "are", np.round(chi2_non_linear,4), "and", np.round(chi2_linear,4))

print("These values are very low. This means that our models fit our data",
      "very well. Thus, the euclidian distance between the data points and our",
      "curves are in general low.")
print("However, this is not necesserely a good sign. Our reduced Chi squared",
      "values should ideally both be equal to one. That we have extreamly low",
      "reduced Chi squared values implies that we have to little data.",
      "That we are in risk of overfitting our models to our data.")

#Now we calculate the estimate of the halflife, for our two models:
# We know half life = (mean lifetime)*ln(2)
# Furthermore, we know that the parameter b in the non-linear model, is equal
# to -1/mean_lifetime. Thus, for the non-linear model, we have:
b_non_linear = popt_non_linear[1]
mean_lifetime_non_linear = -1/b_non_linear
Half_life_non_linear = mean_lifetime_non_linear*np.log(2)
#Now we calculate the uncertainty of the halflife. multiplying the quantity
# by scalars, means that we must also multiply the uncertainties by these
# scalars. However, when we take the quantity 1/a, then the uncertainty of
# 1/a is equal to the uncertainty of a divided by a^2:
Half_life_non_linear_uncertainty = -np.log(2) * (
    pstd_non_linear[1]/popt_non_linear[1]**2)

print("\nThe Halflife of the Barium, predicted by the non-linear model, is:",
      np.round(Half_life_non_linear,0), "+-", np.round(
          Half_life_non_linear_uncertainty,0))
#For the linear model, we have:
a_linear = popt_linear[0]
mean_lifetime_linear = -1/a_linear
Half_life_linear = mean_lifetime_linear*np.log(2)
Half_life_linear_uncertainty = -np.log(2) * (pstd_linear[0]/popt_linear[0]**2)


print("The Halflife of the Barium, predicted by the linear model, is:",
      np.round(Half_life_linear,0), "+-", np.round(
          Half_life_linear_uncertainty,0))
print("The non-linear model gave a half-life closer to the expected",
      "half-life of 2.6 minutes/156 seconds.")
```

```
print("\nThe non-linear regression method gave a half-life closer to the",
      "expected half-life of 2.6 minutes. 170 seconds is closer to 156",
      "seconds, than 176 seconds. However, note that the theoretical half",
      "life falls within both of our estimates of the half lifes with",
      "assosiated uncertaintees.")

#Now we plot these models, the data, and the theoretical curve:
#Now we create some data points on the model curve for plotting

#We calculate the predicted count rates by the theory:
# Note, theoretical halflife is 2.6 min
Count_rate_theory_predicted = np.zeros(len(Sample_number_barium))
for i in range(len(Sample_number_barium)):
    Count_rate_theory_predicted[i] = popt_non_linear[0]*np.exp(
        -i*Sample_time/(2.6*60)*np.log(2))

#Now we plot the original data with error bars, along with the curve fit model
plt.figure(figsize=(10,5))
plt.errorbar(Sample_number_barium*Sample_time,
             Count_rate,Count_rate_uncertainty ,c='r', ls='',
             marker='o',lw=1,capsize=2,
             label = 'Points of measurement with uncertainty')

plt.plot(Sample_number_barium*Sample_time,
         Count_rate_predicted_non_linear, c='b',
         label = 'Non-linear curve fit')
plt.plot(Sample_number_barium*Sample_time,
         Count_rate_predicted_linear_non_linear, c='g',
         label = 'linear curve fit')
plt.plot(Sample_number_barium*Sample_time,Count_rate_theory_predicted, c='y',
         label = 'Theoretical curve')

plt.title("Count Rates of Barium (Ba-137m)")
plt.xlabel("Time in seconds (s)")
plt.ylabel("Count Rate")
plt.legend()
plt.grid()
plt.savefig("Count Rates of Barium (Ba-137m)"+'.png')
plt.show()


#Now we plot the logarithmic version of this:
plt.figure(figsize=(10,5))
plt.errorbar(Sample_number_barium*Sample_time,
             Count_rate,Count_rate_uncertainty ,c='r', ls='',
             marker='o',lw=1,capsize=2,
             label = 'Points of measurement with uncertainty')

plt.plot(Sample_number_barium*Sample_time,
         Count_rate_predicted_non_linear, c='b',
         label = 'Non-linear curve fit')
plt.plot(Sample_number_barium*Sample_time,
         Count_rate_predicted_linear_non_linear, c='g',
         label = 'linear curve fit')
plt.plot(Sample_number_barium*Sample_time,Count_rate_theory_predicted, c='y',
         label = 'Theoretical curve')

plt.title("Logarithmic Count Rates of Barium (Ba-137m)")
plt.xlabel("Time in seconds (s)")
plt.ylabel("Count Rate")
plt.legend()
plt.grid()
plt.yscale('log')
plt.savefig("Logarithmic Count Rates of Barium (Ba-137m)"+'.png')
plt.show()

print("\nThough it is hard to distinguish the two models in the non-linear",
      "plot, in the logaritmic, linear plot, you can more easily see that",
      "the non-linear model returns a curve closer to the theoretical curve,",
      "than the curve returned by the linear model.")
print("Both models does however fit the theoretical curve quite good.",
      "Furthermore, both models are well within the uncertainty of our",
      "measurements, see plots above.")
```

A.4 Python Code: Exercise 5

The Python code for this exercise is divided into two files. statslab.py file contains utility methods which we will be frequently using in this course. lab_2_ex_5_code.py file contains the code which analyzes the data for this experiment.

A.4.1 statslab.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
@author: Pankaj Patil
"""

import numpy as np
import scipy.optimize as optim
import matplotlib.pyplot as plt


################################################################################
# Utility Methods Library
#
# This file contains some utility method which are common to our data analysis.
# This library also contains customized plotting methods.
################################################################################

# use bigger font size for plots
plt.rcParams.update({'font.size': 20})

def chi2(y_measure,y_predict,errors):
    """Calculate the chi squared value given a measurement with errors and
    prediction"""
    return np.sum( np.power(y_measure - y_predict, 2) / np.power(errors, 2) )

def chi2reduced(y_measure, y_predict, errors, number_of_parameters):
    """Calculate the reduced chi squared value given a measurement with errors
    and prediction, and knowing the number of parameters in the model."""
    return chi2(y_measure, y_predict, errors)/ \
            (y_measure.size - number_of_parameters)

def read_data(filename, skiprows=1, usecols=(0,1), delimiter=","):
    """Load give\n file as csv with given parameters,
    returns the unpacked values"""
    return np.loadtxt(filename,
                      skiprows=skiprows,
                      usecols=usecols,
                      delimiter=delimiter,
                      unpack=True)

def fit_data(model_func, xdata, ydata, yerrors, guess=None):
    """Utility function to call curve_fit given x and y data with errors"""
    popt, pcov = optim.curve_fit(model_func,
                                xdata,
                                ydata,
                                absolute_sigma=True,
                                sigma=yerrors,
                                p0=guess)

    pstd = np.sqrt(np.diag(pcov))
    return popt, pstd

# y = ax+b
def linear_regression(xdata, ydata):
    """Simple linear regression model"""
    x_bar = np.average(xdata)
    y_bar = np.average(ydata)
    a_hat = np.sum( (xdata - x_bar) * (ydata - y_bar) ) / \
            np.sum( np.power((xdata - x_bar), 2) )
    b_hat = y_bar - a_hat  * x_bar
    return a_hat, b_hat

class plot_details:
    """Utility class to store information about plots"""
    def __init__(self, title):
        self.title = title
        self.x_log_scale = False
        self.y_log_scale = False

    def errorbar_legend(self, v):
        self.errorbar_legend = v
    def fitted_curve_legend(self, v):
        self.fitted_curve_legend = v
    def x_axis_label(self, v):
        self.x_axis_label = v
    def y_axis_label(self, v):
        self.y_axis_label = v
    def xdata(self, x):
        self.xdata = x
    def ydata(self, y):
        self.ydata = y
    def yerrors(self, y):
        self.yerrors = y
    def xdata_for_prediction(self, x):
        self.xdata_for_prediction = x
    def ydata_predicted(self, y):
        self.ydata_predicted = y
    def legend_position(self, p):
        self.legend_loc = p
    def chi2_reduced(self, c):
        self.chi2_reduced = c
    def set_x_log_scale(self, c):
        self.x_log_scale = c
    def set_y_log_scale(self, c):
        self.y_log_scale = c

def plot(plot_details, new_figure=True, error_plot=True):
    """Utility method to plot errorbar and line chart together,
    with given arguments"""
    if new_figure:
        plt.figure(figsize=(16, 10))
        plt.style.use("seaborn-whitegrid")

    # plot the error bar chart
    if error_plot:
        plt.errorbar(plot_details.xdata,
                     plot_details.ydata,
                     yerr=plot_details.yerrors,
                     marker="o",
                     label=plot_details.errorbar_legend,
                     capsize=2,
                     ls="")

    # plot the fitted curve
    plt.plot(plot_details.xdata_for_prediction,
             plot_details.ydata_predicted,
             label=plot_details.fitted_curve_legend)

    # legend and title
    plt.title(plot_details.title)
    plt.xlabel(plot_details.x_axis_label)
    plt.ylabel(plot_details.y_axis_label)

    if plot_details.x_log_scale:
        plt.xscale("log")

    if plot_details.y_log_scale:
        plt.yscale("log")
    legend_pos = "upper left"
    if hasattr(plot_details, "legend_loc"):
        legend_pos = plot_details.legend_loc

    plt.legend(loc=legend_pos)

def plot_histogram(count_data, new_figure=True,
                   title="", xlabel="", ylabel="", legend_pos='upper left'):
    """Utility method to plot histogram, with  densiity=True"""
    if new_figure:
        plt.figure(figsize=(16, 15))
        plt.style.use("seaborn-whitegrid")

    # plot the data
    count, bins, _ = plt.hist(count_data, bins=10, rwidth=0.95, density=True)

    # legend and title
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)

    # set legend position
    plt.legend(loc=legend_pos)
    return count, bins
```

A.4.2 lab_2_ex_5_code.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
@author: Pankaj Patil
"""

import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats

# import our utility method library from statslab.py
import statslab as utils

# load the radioactive count from a Fiesta plate.
measured_count = utils.read_data("Fiesta_30092021.txt",
                                                usecols=(1),
                                                skiprows=2,
                                                delimiter=None)

# load the background count data
bg_measured_count = utils.read_data("Fiesta_Background_30092021.txt",
                                       usecols=(1),
                                       skiprows=2,
                                       delimiter=None)

# compute the mean background count
mean_background_count = np.mean(bg_measured_count)

# correct the measured count by reducing from it mean background count
measured_count_corrected =  (measured_count - mean_background_count)

# function to analyze given counts data
def analyze_data(data_type, counts):
    """
    Function to analyze given counts data, and plot histogram, along
    with Poisson Mass Function and Gaussian Distribution
    """

    # create new figure for this analysis
    plt.figure(figsize=(16, 10))
    plt.style.use("seaborn-whitegrid")

    # plot the histogram for the measured (corrected) counts
    utils.plot_histogram(counts)

    # the most appropriate value for mu = average value of measured (corrected)
    # count data
    mu = np.average(counts)
    print("%s mu ="%data_type, mu)

    # standard daviation for Gaussian Distribution
    std = np.sqrt(mu)
    print("%s std = %.2f"% (data_type, std))

    # get the x range for our  data as input to distrribution functions
    x = np.arange(np.min(counts), np.max(counts))

    # plot the Possion Probability Mass Function
    pmf_data = stats.poisson.pmf(x.astype(int), mu)
    plt.plot(x, pmf_data, ms=1, label="Possion Probability Mass \nFunction")

    # plot the Gaussian Distrribution
    plt.plot(x,
             stats.norm.pdf(x, loc=mu, scale=std),
             label="Gaussian Distribution Function")

    # add legend
    plt.xlabel("Radioactive Count for %s" % data_type)
    plt.ylabel("Probability Density")
    plt.title("%s Random Data Analysis" % data_type)
    plt.legend()

    # save the plot
    plt.savefig("%s.png" % data_type)

# analyze the Data from Fiesta Plate
analyze_data("Fiesta Plate", measured_count_corrected)

# analyze the Data from Background
analyze_data("Background", bg_measured_count)
```

References

[1] Lab Manual - Nonlinear fitting methods I - Exercise 2.

[2] Lab Manual - Random number analysis - Exercise 5.