PyLab - Radioactive Decay

PHY224 Lab 2

Fredrik Dahl Bråten, Pankaj Patil

October 5, 2021

Contents

List of Figures

List of Tables

1 Exercise 2: PyLab - Radioactive Decay

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 $I(t) = a * exp(b * t)$, and one where $ln(I(t)) = a * t + b$, 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 reduced Chi squared 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

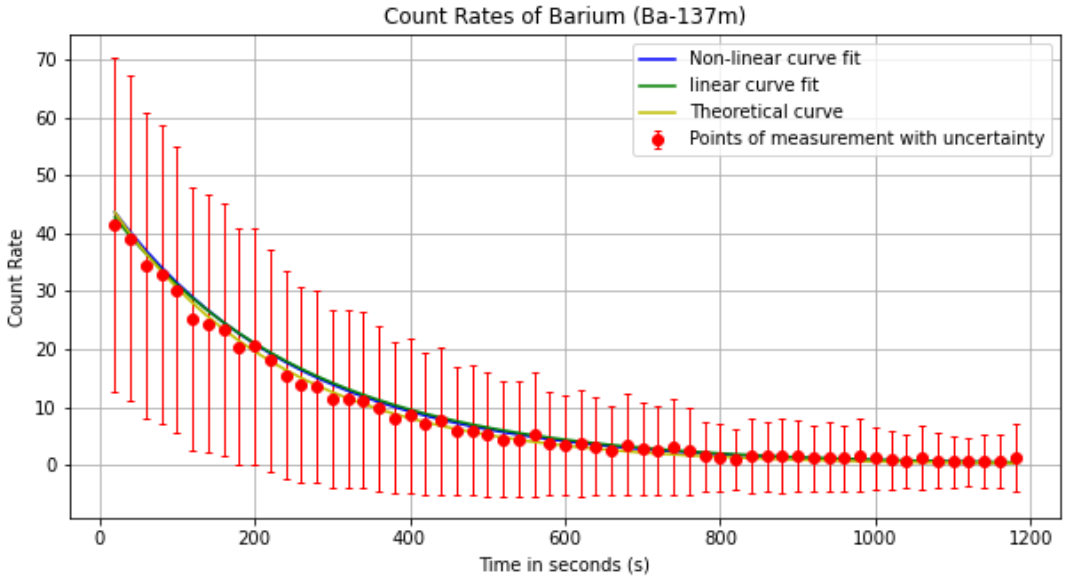
In this exercise, we measure and model the rates of emission over time caused by radioactive decay from a sample of Barium (Ba-137m). The corresponding theoretically predicted relationship between emission rate (I) and time (t) is: $I(t) = I_0 * exp(-t/\tau) = I_0 * (1/2)^(t/t_(1/2))$. Where I_0 is the initial emission rate, τ is the mean lifetime of the isotope, and $t_(1/2)$ 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 $I(t) = a * exp(b * t)$, and one where $ln(I(t)) = a * t + b$, 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 exercise3.pdf 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.pdf document.

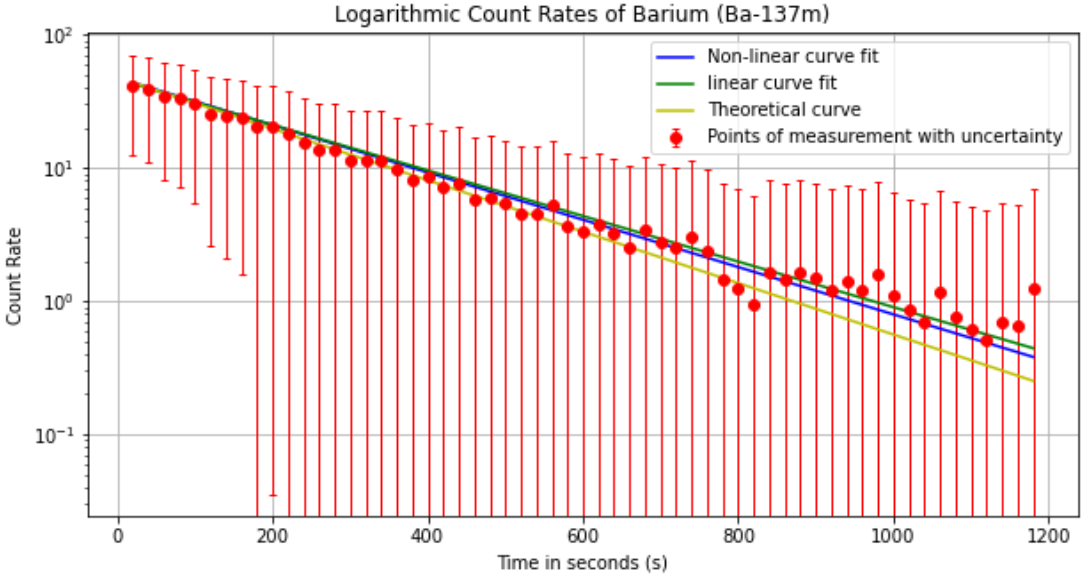
1.4 Results

Below in 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.



Points of measurement with corresponding error bars, theoretical power laws for tungsten and a blackbody, along with our two power law models best fitted to our data points

Figure 1: Power law, Current vs. Voltage in a Circuit with a Lightbulb



Points of measurement with corresponding error bars, theoretical power laws for tungsten and a blackbody, along with our two power law models best fitted to our data points

Figure 2: Power law, Current vs. Voltage in a Circuit with a Lightbulb

The estimated optimal parameters with uncertainty by scipy optimize curve fit are: [a,b] = [-0.00393908 3.75408883] +- [0.00090733 0.29589009] for the linear model, and [a,b] = [4.36112915e+01 -4.08767747e-03] +- [1.35411958e+01 1.00398911e-03] for the non linear model.

The reduced Chi squared 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 reduced Chi squared 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 reduced Chi squared values should ideally both be equal to one. That we have extremely low reduced Chi squared 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 reduced Chi squared 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

2.2 Introduction

2.3 Methods, Materials and Experimental Procedure

2.4 Results

2.5 Discussion

2.6 Conclusions

A Appendix

A.1 Data Tables for Exercise 2

Radioactive Decay Experiment Sep 30, 2021 9:38 AM

Sample Number	Number of Counts
1	834
2	785
3	691
4	663
5	609
6	508
7	491
8	472
9	411
10	413
11	365
12	314
13	280
14	275
15	232
16	231
17	228
18	199
19	165
20	174
21	145
22	155
23	121
24	124
25	111
26	93
27	93
28	109
29	77
30	71
31	79
32	68
33	55
34	73
35	59
36	54
37	64
38	52
39	33
40	29
41	23
42	37
43	33
44	37
45	34

Radioactive Decay Experiment Sep 30, 2021 9:12 AM

Sample Number	Number of Counts
1	2
2	4
3	5
4	3
5	6
6	3
7	5
8	5
9	3
10	6
11	3
12	0
13	4
14	4
15	2
16	1
17	6
18	4
19	3
20	2
21	4
22	6
23	7
24	5
25	3
26	5
27	3
28	3
29	1
30	3
31	4
32	5
33	2
34	7
35	5
36	1
37	3
38	8
39	1
40	8
41	6
42	7
43	3
44	3
45	2
46	7
47	4
48	3
49	7

A.2 Data Tables for Exercise 3

Radioactive Decay Experiment Sep 30, 2021 9:53 AM

Sample Number	Number of Counts
1	122
2	138
3	125
4	119
5	141
6	130
7	120
8	140
9	109
10	130
11	131
12	120
13	138
14	148
15	114
16	137
17	131
18	126
19	114
20	120
21	128
22	141
23	116
24	131
25	120
26	125
27	110
28	121
29	142
30	125
31	114
32	137
33	98
34	115
35	126
36	136
37	134
38	132
39	129
40	140
41	94
42	124
43	121
44	134
45	137
46	158
47	110

Radioactive Decay Experiment Sep 30, 2021 10:08 AM

Sample Number	Number of Counts
1	2
2	0
3	0
4	2
5	3
6	0
7	4
8	0
9	1
10	1
11	4
12	1
13	5
14	4
15	0
16	4
17	1
18	2
19	2
20	1
21	1
22	1
23	0
24	0
25	2
26	2
27	3
28	1
29	2
30	1
31	3
32	4
33	2
34	0
35	1
36	3
37	3
38	1
39	1
40	1
41	2
42	2
43	1
44	1
45	0
46	3
47	2
48	1
49	3

A.3 Python Code: Exercise 1

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.3.3 Program Output

A.4 Python Code: Exercise 3

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. exercise_3.py file contains the code which analyzes the data for this experiment.

A.4.1 Functions.py

A.4.2 exercise_3.py

References

[1] Lab Manual - PyLab - Ohm and Power laws - Exercise 2.

[2] Lab Manual - PyLab - Ohm and Power laws - Exercise 5.