Non-Linear Fits and Log Plots

Objective

We want to fit a non-linear function, such as

$$y(x) = a + \frac{b}{(x-c)^2},$$

to data that roughly follows this behavior. The parameters of the function are:

- $a =$ y-axis offset
- $b =$ amplitude of the $1/x^2$ curve
- $c =$ x-axis offset

Non-Linear Least squares fitting

Non-linear least squares fitting works the same way as a linear least squares fit: We want to find the parameters of the model for the curve that give the best description of the data. We define "best" to be the set of parameters that minimize χ^2, which is the sum of the squared y-distances between the data and the curve. The squared sum χ^2 is defined as

$$\chi^2 = \sum_i \frac{(y_i - y(x_i))^2}{\sigma_{y_i}^2}$$

where the sum over i is over all the data points. The variables are the following:

- y_i are the individual data points.
- σ_{y_i} are the errors on the individual data points.
- $y(x_i)$ is the value of the linear function at the data point with $x = x_i$.

Minimizing χ^2

We will use the "curve_fit" command in the **scipy** library to minimize χ^2. Conveniently, this command automatically returns the best fit values for m and b, as well as the errors on them. These are 1-σ errors, which means that if you repeat the process with new data , i.e., repeat experiment, then you will get m and b within this error bar range 67% of the time.

Quality of the fit

We evaluate the "quality" of the fit by looking at the "reduced χ^2", χ^2_ν, which is given by

$$\chi^2_\nu = \frac{\chi^2}{\text{degrees of freedom}}$$

The *degrees of freedom* are given by

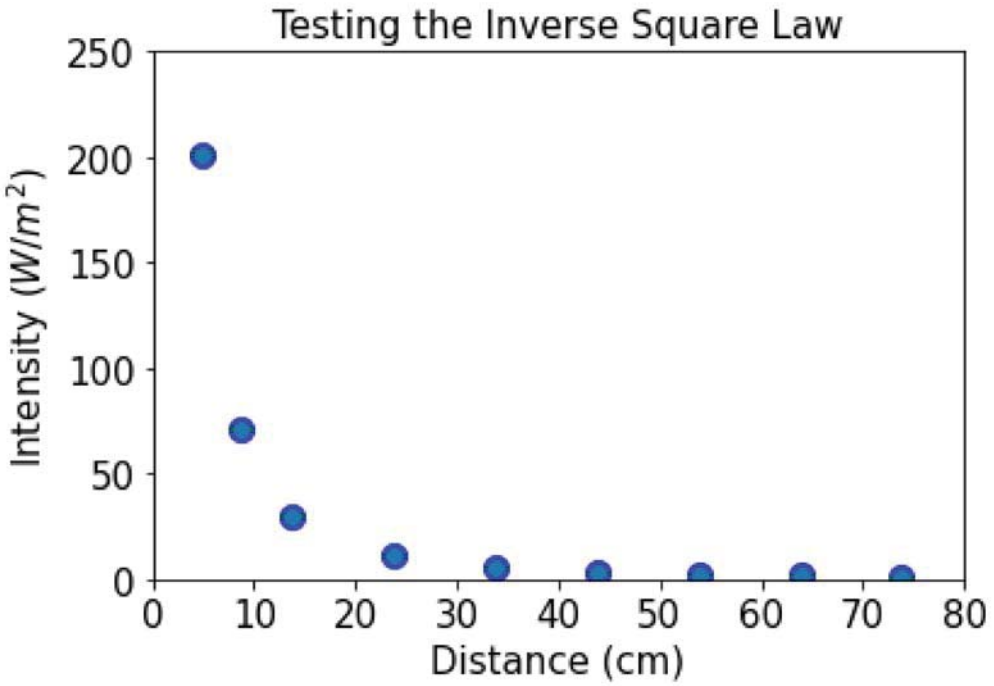
$$\nu = \text{degrees of freedom} = \#\text{ of data points} - \#\text{ of fit parameters}$$

In general, you want $\chi^2_\nu \simeq 1$. If this is not the case, then there are two main possibilities:

- If $\chi^2_\nu \gg 1$, then the errors bars are likely to be **underestimated**, *or* the model function **does not** describe the data well (statistically speaking).
- If $\chi^2_\nu \ll 1$, then the error bars are likely to be **overestimated**, meaning that the model function describe the data better than it should (statistically speaking).

```
# First, Let's plot some data that needs to be fitted

# import library modules needed for plotting and data analysis
import numpy as np
import matplotlib.pyplot as plt

# import library module for curve fitting
from scipy.optimize import curve_fit

# Data to be plotted and fitted
x_data=np.array([5.0, 8.8, 13.8, 23.8, 33.8, 43.8, 53.8, 63.8, 73.8])             # Distance in cm
y_data=np.array([201.4, 71.2, 30.2, 10.8, 5.6, 3.4, 2.3, 1.7, 1.3])              # Intensity in W/m^2

y_errorbars=np.array([2.0, 0.7, 0.3, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1])       # Error on y_data

# Plot the data on a new figure
fig1=plt.figure(1)                                              # create new figure for plotting
plt.plot(x_data, y_data, 'blue', linestyle='none', marker='o', markerfacecolor='blue', markersize=10)  #
plt.title('Testing the Inverse Square Law', fontsize=15)                          # Add the plot title
```

```
plt.ylabel('Intensity ($W/m^2$)', fontsize=15)                    # Add the y-axis label, "fontsize" is
plt.yticks(size=15)                                              # adjust the size of the y-axis tick numbe

plt.xlabel('Distance (cm)', fontsize=15)                          # Add the x-axis label, "fontsize" is
plt.xticks(size=15)                                              # adjust the size of the x-axis tick numbe

plt.xlim(0.0, 80.0)                                              # Set the range of the plot's x-axis
plt.ylim(0.0, 250.0)                                               # Set the range of the plot's y-axis

plt.errorbar(x_data, y_data, y_errorbars, fmt='o', ecolor='green', capsize=5)   # Add error bars

fig1.show()
```



Problem

The data points span several orders of magnitude, which makes it hard to distinguish the points close to zero from zero and from each other (on the y-axis).

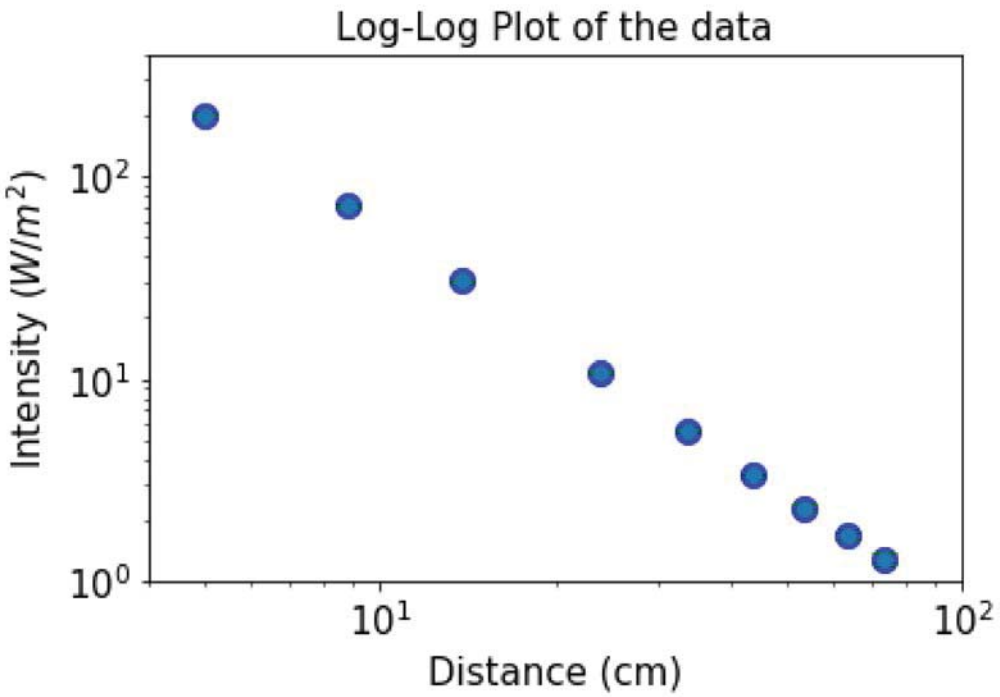
Solution: Log-Log plot

If we take the logarithm (base e or base 10) of the x-axis and y-axis values, then the data points are more spread out and easier to distinguish along both the x and y directions.

Note: A log-log plot cannot have negative values or zero values. The x-axis and y-axis scales must be entirely positive (otherwise you will get a divergence,i.e. $log(0) = -\infty$).

```
# Plot the data on a Log-Log plots (new figure)
fig2=plt.figure(2)                                          # create new figure for plotting
plt.loglog(x_data, y_data, 'blue', linestyle='none', marker='o', markerfacecolor='blue', markersize=10)
plt.title('Log-Log Plot of the data', fontsize=15)                              # Add the plot title

plt.ylabel('Intensity ($W/m^2$)', fontsize=15)                      # Add the y-axis label, "fontsize" is
plt.yticks(size=15)                                         # adjust the size of the y-axis tick numbe

plt.xlabel('Distance (cm)', fontsize=15)                         # Add the x-axis label, "fontsize" is
plt.xticks(size=15)                                         # adjust the size of the x-axis tick numbe

plt.xlim(4.0, 100.0)                                         # Set the range of the plot's x-axis
```

```
plt.ylim(1.0, 400.0)                                                    # Set the range of the plot's y-axis

plt.errorbar(x_data, y_data, y_errorbars, fmt='o', ecolor='green', capsize=5)   # Add error bars

fig2.show()
```



Note

If data follows a "linear" trend in a log-log plot, then it actually follows a power law: $y = Ax^n$

```
# Non-linear function with parameters a = y-offset, b = amplitude, c = x-offset
def yModel(x, a, b, c):
  return a + (b/(np.power(np.float_(x)-c,2)))
```

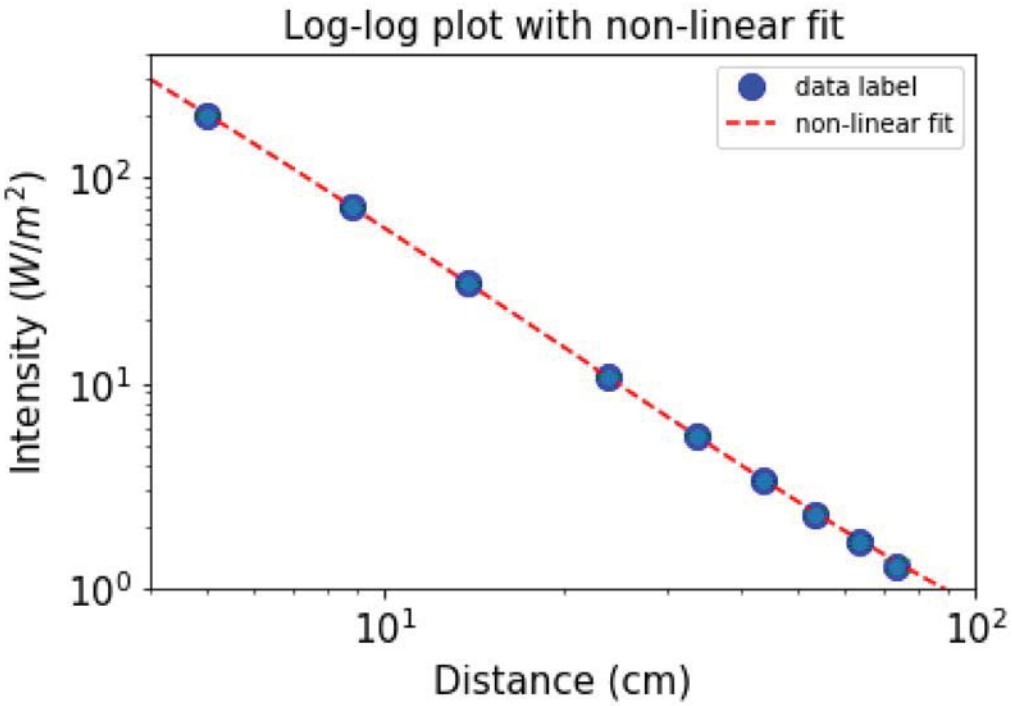
```
# Fit the data with the linear model
initialParameters = [0.5, 4000.0, 0.1]           # initial guess for fit parameters [a,b,c]
finalParameters, finalParameterErrors = curve_fit(yModel, x_data, y_data, initialParameters, y_errorbars,

# Extract Fit values and errors for a=y-offset, b=amplitude, and c=x-offset
a_fit = finalParameters[0]
a_error_squared = finalParameterErrors[0,0]
a_error = np.sqrt(a_error_squared)

b_fit = finalParameters[1]
b_error_squared = finalParameterErrors[1,1]
b_error = np.sqrt(b_error_squared)

c_fit = finalParameters[2]
c_error_squared = finalParameterErrors[2,2]
c_error = np.sqrt(c_error_squared)


a_b_covariance_error2 = finalParameterErrors[1,0]
a_c_covariance_error2 = finalParameterErrors[2,0]
b_c_covariance_error2 = finalParameterErrors[2,1]

# Evaluate the quality of the fit
yModel_i = yModel(x_data, a_fit, b_fit, c_fit)             # Calculate the Y-array (yModel_i) for the va
residuals_y = y_data - yModel_i                            # Calculate the difference between data and m
residuals_y_normalized = residuals_y/y_errorbars           # Normalize the residuals to the Y-error on e

Chi_squared = np.sum(residuals_y_normalized**2)            # Calculate the Chi^2 for the data and fit
DOF = len(y_data)-len(finalParameters)                     # Calculate the degrees of freedom: DOF = num
Reduced_Chi_squared = Chi_squared/DOF                      # Calculate the reduced Chi^2, which determir
```

```
# Output the fit parameter values and errors
print("Model parameters:")
print("a = ", a_fit, "+/-", a_error)
print("b = ", b_fit, "+/-", b_error)
print("c = ", c_fit, "+/-", c_error)
print("")
print("ab covariance error (squared) =", a_b_covariance_error2)
print("ac covariance error (squared) =", a_c_covariance_error2)
print("bc covariance error (squared) =", b_c_covariance_error2)
print("")
print("Fit quality")
print("Reduced Chi^2 =", Reduced_Chi_squared)
```

```
Model parameters:
a =  0.21499542825386686 +/- 0.05978265871535627
b =  6266.804475648378 +/- 82.56433095647407
c =  -0.5889564892086729 +/- 0.053849748650827124

ab covariance error (squared) = -3.563888091726095
ac covariance error (squared) = 0.0019047675219247967
bc covariance error (squared) = -4.032560225176399

Fit quality
Reduced Chi^2 = 0.39721453534226886
```

```
# Make a new plot with the original data and the non-linear fit

fig3=plt.figure(3)                                          # create new figure for plotting

# Original code for plotting data points with error bars, but with legend label added
```

```
#############################################################################################
plt.loglog(x_data, y_data, 'blue', linestyle='none', marker='o', markerfacecolor='blue', markersize=10, l
plt.title('Log-log plot with non-linear fit', fontsize=15)                                  # Add the plot title

plt.ylabel('Intensity ($W/m^2$)', fontsize=15)                        # Add the y-axis label, "fontsize" is
plt.yticks(size=15)                                                 # adjust the size of the y-axis tick numbe

plt.xlabel('Distance (cm)', fontsize=15)                               # Add the x-axis label, "fontsize" is
plt.xticks(size=15)                                                 # adjust the size of the x-axis tick numbe

plt.xlim(4.0, 100.0)                                                 # Set the range of the plot's x-axis
plt.ylim(1.0, 400.0)                                                  # Set the range of the plot's y-axis

plt.errorbar(x_data, y_data, y_errorbars, fmt='o', ecolor='green', capsize=5)   # Add error bars
#############################################################################################

# Generate the points for the fitting curve (linear function in this case)
x_fit = np.linspace(4.0,100.0,100)                    # Generate an ordered array of 100 x-values betweer
y_fit = yModel(x_fit, a_fit, b_fit, c_fit)                  # Generate the y-values for the fitting curve

plt.plot(x_fit,y_fit, 'r--', label='non-linear fit')                          # Plot the fit as a dashed r

# Add in legend
plt.legend()

plt.show()
```




```
# Save the plot to your computer
from google.colab import files                  # import the library for loading/saving files to/from C

figure_filename='Python_plot_withNonLinearFit_v1.pdf'     # string variable that contains the filenar
fig3.savefig(figure_filename)                   # Saves the fig2 figure to the Google colab environment
files.download(figure_filename)                 # Download the file "figure_filename" from the Google c
```

```
# Plot the normalized residuals from the non-linear fit

fig4=plt.figure(4)                                              # create new figure for plotting

plt.plot(x_data, residuals_y_normalized, 'blue', linestyle='none', marker='o', markerfacecolor='blue
```

```
plt.title('Normalized Residuals from Fit', fontsize=15)                          # Add the plot tit

plt.ylabel('Normalized Residuals', fontsize=15)                  # Add the y-axis label, "fontsize
plt.yticks(size=15)                                             # adjust the size of the y-axis tick m

plt.xlabel('Distance (cm)', fontsize=15)                         # Add the x-axis label, "fontsize
plt.xticks(size=15)                                             # adjust the size of the x-axis tick m

plt.xlim(4.0, 80.0)                                             # Set the range of the plot's x-axis
plt.ylim(-2.0, 2.0)                                              # Set the range of the plot's y-axis

#plt.errorbar(x_data, y_data, y_errorbars, fmt='o', ecolor='green', capsize=5)   # Add error bars
x_lines = np.array([4.0, 80.0])
y_lines = np.array([1.0, 1.0])
plt.plot(x_lines,0.0*y_lines, 'r-')
plt.plot(x_lines,y_lines, 'r--')
plt.plot(x_lines,-y_lines, 'r--')
```

```
[<matplotlib.lines.Line2D at 0x7f34e74c1e90>]
```