# Curve Fitting

February 14, 2022

## 1 The Least Squares Method

### 1.1 Linear Regression

We can consider two linearly related variables and try to find the straight line (curve) that best represents the relationship between them. The functional relationship between the dependent and the independent variable is given by,

$$y = a + bx.$$

The goal is to find the optimal values of $a$ and $b$. It is impossible to exactly fit a straightline to experimentally measured data like the one shown in the figure below. Because there are $N$ points while for a straightline one only needs two points. Moreover there are flucuations in the data. This is a case of an *overdetermined* problem.

The data consists of pairs of measurements $(x_i, y_i)$. Attributing all the experimental uncertainty to the dependent variable, we want to find the values of $a$ and $b$ that minimize the discrepency between the measured values $y_i$ and the predicted values $y(x_i) = a + bx_i$ of the dependent variable,

$$\begin{aligned} d_i &= y(x_i) - y_i \\ &= a + bx_i - y_i. \end{aligned}$$

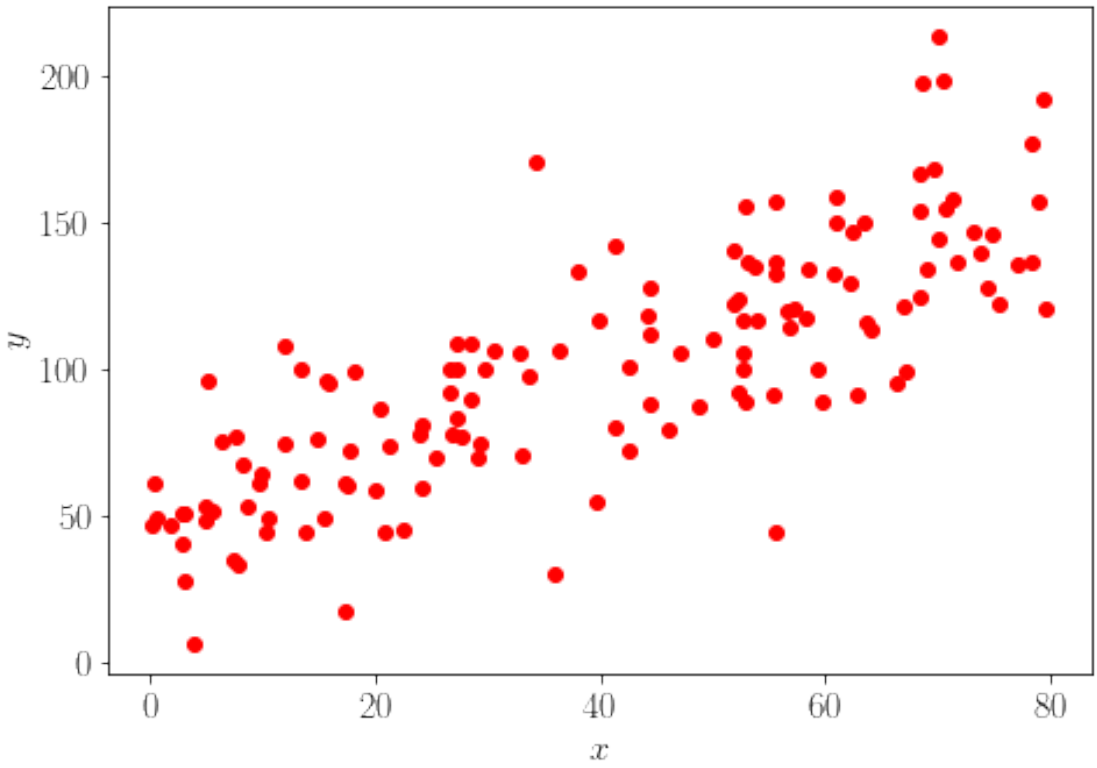
It is sum of these deviations, also referred as residuals, that should be minimized to take into account all the data points. However, the sum is not a good measure of the best fit since the large positive fluctuations can cancel the large negative fluctuations still producing small $\sum_i d_i$. Although summing the absolute values $\sum_i |d_i|$ would work but is cumbersome to deal with analytically. Therefore, it is sum of the squares that is widely used as a measure of the best fit and thus minimized. The quantity that is actually minimized is $\chi^2$ and given as,

$$\begin{aligned} \chi^2 &= \sum_i \left( \frac{y(x_i) - y_i}{\alpha_i} \right)^2, \\ &= \sum_i \left( \frac{a + bx_i - y_i}{\alpha_i} \right)^2, \end{aligned}$$

where $\alpha_i$ is the standard error in the $i^{th}$ data point. For analytical justification of $\chi^2$ or the sum of the square of normalized residuals see Measurements and their Uncertainties by Hughes and Hase.

The choice of the analytical function $y(x)$ is arbitrary and should be informed by the expected trends in the data. Therefore, taking a careful look at the data before hand is instructive.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import time

# For tex fonts and math-mode in figures
plt.rcParams['text.latex.preamble']=r'\usepackage{lmodern}'
params = {'text.usetex':True,'font.size':16,'font.family':
          'serif','figure.autolayout': True}
plt.rcParams.update(params)

# read data from file
data = np.loadtxt('data-01.txt',comments='#')

# plot data
fig1 = plt.figure(figsize=(7.0,5.0))
fig1 = plt.plot(data[:,0],data[:,1], 'ro')
fig1 = plt.xlabel('$x$')
fig1 = plt.ylabel('$y$')
```




After choosing an analytical function to fit to the data, there are a number of ways to numerically

minimize the $\chi^2$. One of those methods is known as Gradient Descent.

### 1.2 Gradient Descent

Gradient descent like other minimization methods is iterative. A trial solution is assumed and $\chi^2$ is calculated. Then the values of the fitting parameters are updated to improve the trial solution or minimize $\chi^2$. In gradient descent the vector $\nabla\chi^2$ is calculated which points towards the minima within the parameter space. The fitting parameters are updated to step towards the minima along the line of **steepest descent**. The update rule for a straightline would be,

$$a_{s+1} = a_s - \beta \frac{\partial \chi^2(a,b)}{\partial a}\bigg|_{a=a_s, b=b_s} = a_s - 2\frac{\beta}{\alpha}\sum_i (a + bx_i - y_i)$$

$$b_{s+1} = b_s - \beta \frac{\partial \chi^2(a,b)}{\partial b}\bigg|_{a=a_s, b=b_s} = b_s - 2\frac{\beta}{\alpha}\sum_i (a + bx_i - y_i)\, x_i.$$

We have assumed that the standard error is constant for all points and $\beta$ is chosen to control the rate of descent. The update rules can be recast in vector form to take advantage of optimal vector operations. The fitting parameters can be represented as a column vector of $2 \times 1$. The independent variable can be reprsented by a $N \times 2$ matrix where the first column contains ones, to represent the intercept, and the second contains $x_i$. Finally the dependent variable can be respresented by an $N \times 1$ vector,

$$\mathbf{a} = \mathbf{a} - 2\frac{\beta}{\alpha}\mathbf{X}^T\left(\mathbf{X}\mathbf{a} - \mathbf{y}\right).$$

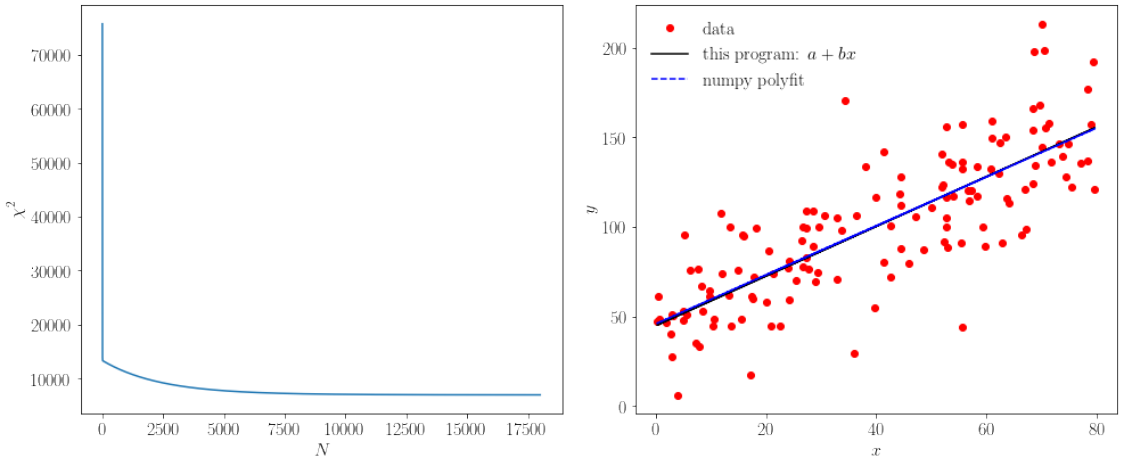
[2]:
```
N = len(data[:,0])          # size of the data set.
X = data[:,0].reshape(N,1) # to get a neat column vector

intCept = np.ones((N, 1))
X = np.hstack((intCept,X)) # to add a column of ones to the vector X

y = data[:,1].reshape(N,1) # get a neat column vector y

fittingParam = np.zeros((2,1)) # initialize the fitting parameters to zero

numbIter = 18000 # number of iteration of the gradient descent
beta = 0.00001    # rate of descent
alpha = np.std(y)/np.sqrt(N-1)  # standard error
alphaS = alpha**2

xPrime = np.transpose(X)   # to avoid calculating the transpose inside the loop

chiSHistory = np.zeros((numbIter, 1)) # to record how chi**2 changes
                                      # with iterations
paraHistory = np.zeros((2,numbIter+1))

def computeChiS(X,y, fittingParam):
```

```
    chiS = 0;
    auxVar = X@fittingParam - y
    chiS = (1/alphaS)*(auxVar.T@auxVar)
    return chiS

def gradDesc(X, y, fittingParam, beta, numbIter):
    for iter in range(numbIter):
        temp_param = fittingParam - (2*beta/alpha)*xPrime@(X@fittingParam - y)
        fittingParam = temp_param
        chiSHistory[iter,:] = computeChiS(X, y, fittingParam)
        paraHistory[:,iter+1] = fittingParam[:,0]
    return fittingParam
```

[3]:
```
print(fittingParam)
curveParam = gradDesc(X,y,fittingParam,beta,numbIter)
print('The best fit parameters calculated by this program:\n a = '
      , curveParam[0,0], '\n b = ', curveParam[1,0])

numpyParam = np.polyfit(data[:,0], data[:,1], 1)
numpyParam = numpyParam.reshape(2,1)
numpyParam = np.flip(numpyParam)
print('The best fit parameters from the built in numpy polyfit:\n a = '
      , numpyParam[0,0], '\n b = ', numpyParam[1,0])
```

```
[[0.]
 [0.]]
The best fit parameters calculated by this program:
 a =  44.84352970665802
 b =  1.3853441280883674
The best fit parameters from the built in numpy polyfit:
 a =  45.86844040055611
 b =  1.3663885635469724
```

[4]:
```
fig2 = plt.figure(figsize=(14.0,6.0))
ax1 = fig2.add_subplot(1,2,1)
ax2 = fig2.add_subplot(1,2,2)

ax1.plot(chiSHistory)
ax1.set_xlabel('$N$')
ax1.set_ylabel(r'$\chi^2$')

ax2.plot(data[:,0],data[:,1], 'ro', label= 'data')
ax2.plot(data[:,0], X@curveParam, 'k-', label= r'this program: $a + b x$')
ax2.plot(data[:,0], X@numpyParam, 'b--', label= r'numpy polyfit')
ax2.set_xlabel('$x$')
ax2.set_ylabel('$y$')
fig2 = ax2.legend(loc='upper left',fontsize=16, fancybox=True, framealpha=0.0)
```




### 1.3 Error Surface

The following graph shows a contour plot of the error surface and a few respresentative steps along the gradient descent path. Although the step size is the same, the gradient is too steep in the beginning. That is why it roles up and down and up again in the start but eventualy turns right towards the valley.

[5]:
```
M = 1000

a1min = curveParam[0,0]-2*curveParam[0,0]
a1max = curveParam[0,0]+2*curveParam[0,0]

a2min = curveParam[1,0]-2*curveParam[1,0]
a2max = curveParam[1,0]+2*curveParam[1,0]

a1 = np.linspace(a1min,a1max,M)
a2 = np.linspace(a2min,a2max,M)
A, B = np.meshgrid(a1, a2)

def computeChiSonGrid(X,y,A,B):
    chiAux = 0
    for it in range(N):
        chiAux += (B*X[it,1] + A - y[it])**2
    chiSonGrid = (1/alphaS)*chiAux
    return chiSonGrid

chiSurfS = computeChiSonGrid(X,y,A, B)
chiSurfS = chiSurfS/(np.max(chiSurfS))


fig3, ax3 = plt.subplots()
```

```
cntr1 = ax3.contourf(A, B, chiSurfS, levels=60,cmap="RdBu_r")
ax3.set_xlabel('$a$')
ax3.set_ylabel('$b$')
cbar = plt.colorbar(cntr1)
cbar.set_label('$\chi^2$')
scat2 = ax3.scatter(paraHistory[0,0:10:1],paraHistory[1,0:10:1],
                    c=np.nonzero(paraHistory[1,0:11:1]), cmap="bwr",
                    edgecolors=None)
scat2 = ax3.scatter(paraHistory[0,200:18000:1000],paraHistory[1,200:18000:1000],
                    c=np.nonzero(paraHistory[1,200:18000:1000]), cmap="RdYlGn",
                    edgecolors=None)
```