Least Squares Data Fitting

Mohd Harith Akmal Zulfaizal Fadillah, Bahari Idrus,
Mohammad Khatim Hasan

Centre for Artificial Intelligence (CAIT),
Faculty of Information Science & Technology,
Universiti Kebangsaan Malaysia, 43600 Bangi, Selangor

Abstract

The least squares method is an approach for determining the best fit for a series of data points by minimizing the sum of the squared differences between the actual values and the approximated values from the plotted curve. The least squares method is able to approximate functions such as linear, quadratic and cubic functions as well as functions of higher degree and complexity. This report will model, fit and plot the curves for linear least squares of linear, quadratic and cubic models, multiple linear regression (MLR), least squares function approximation and nonlinear least squares using Python.

1 Introduction

The least squares method is a statistical model commonly used in regression analysis in an attempt to estimate the relationships between the dependent variable and one or more independent variables. It is used to approximate the solutions to a system of equations that minimize the sum of squared residuals which is essentially the summation of the squared difference between the y-values from the data set and the approximated model [1].

Its' applications concern mainly of regression, predictions and estimations in a wide range of fields such as finance and economy, image processing, medicine, agriculture among others. The least squares approximation is applicable whenever the dependent variable is assumed to have a causal relationship with the independent variables. Furthermore, the least squares solutions can be easily obtained from matrix linear algebra which will be discussed in the next section. However, since it is inherent in the method to minimize the sum of squared residuals, the existence of outliers can affect the solutions and line of best fit.

In the following sections, we are going to study the method of least squares for some of the general models such as linear, quadratic and cubic. We are also going to look at multiple linear regression (MLR), least squares function approximation and nonlinear least squares. After studying the mathematical foundations of those methods, we are going to model, fit and plot the approximations to relevant applicable data sets using Python and observe the difference in the different ways and modules used in fitting the data.

2 The Method of Least Squares

2.1 Linear Least Squares & Regression

A set of data points consisting of independent and dependent variables, (x_i, y_i) is plotted. A model function of some sort is chosen to best fit the data set by adjusting the parameters available. For example, we can choose the simplest linear model, $y = a_0 + a_1 x$ to be approximated

to the plotted data set. The fit of the model chosen can be measured by minimizing the sum of its squared residuals which is the difference between the actual data points and the prediction from the fit approximated by the model [1],

$$F(a_i) = \sum_{i=1}^{n}(y_i - (a_0 + a_1 x_i))^2$$

To minimize the squared residuals, the partial derivatives of $F(a, b)$ is set to zero,

$$\frac{\partial F(a_i)}{\partial a_2} = -2\sum_{i=1}^{n}(y_i - (a_0 + a_1 x_i)) = 0$$

$$\frac{\partial F(a_i)}{\partial a_1} = -2\sum_{i=1}^{n}(y_i - (a_0 + a_1 x_i))x_i = 0$$

which can be simplified to a similar system of linear equations form, $A\mathbf{x} = \mathbf{b}$,

$$a_1\sum_{i=1}^{n} x_i + na_0 = \sum_{i=1}^{n} y_i$$

$$a_1\sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n} x_i a_0 = \sum_{i=1}^{n} x_i y_i$$

or in its matrix form,

$$\begin{bmatrix} n & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix}\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum y_i x_i \end{bmatrix}$$

Solving for $\mathbf{x} = A^{-1}\mathbf{b}$, the values for a_0 and a_0 can be obtained.

This method to obtain the best fit can also work for any polynomial models by simply changing the model used in the squared residuals. Consider a quadratic model, $y = a_0 + a_1 x + a_2 x^2$,

$$F(a) = \sum_{i=1}^{n}(y_i - (a_0 + a_1 x + a_2 x^2))^2$$

By minimizing the squared residuals, a system of linear equations can be obtained to be solved for the parameters a_0, a_1, a_2.

$$\begin{bmatrix} n & \sum x_i & \sum x_i^2 \\ \sum x_i & \sum x_i^2 & \sum x_i^3 \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 \end{bmatrix}\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum y_i x_i \\ \sum y_i x_i^2 \end{bmatrix}$$

From the obtained system of equations from both linear and quadratic models, it is observed that there is a pattern to the matrices A and $\mathbf{b}$. The system of equations to measure the parameters can be generalized to,

$$\begin{bmatrix} n & \sum x_i & \sum x_i^2 & \dots & \sum x_i^m \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \dots & \sum x_i^{m+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x_i^m & \sum x_i^{m+1} & \sum x_i^{m+2} & \dots & \sum x_i^{m+m} \end{bmatrix}\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{bmatrix} = \begin{bmatrix} \sum y_i x_i^0 \\ \sum y_i x_i^1 \\ \vdots \\ \sum y_i x_i^m \end{bmatrix} \tag{1}$$

It is also possible to measure the paramater from its system of equations, $A\mathbf{x} = \mathbf{b}$ by plugging in the values of x and y into the chosen model such that,

$$\text{Eg: Cubic model} \qquad y = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_m & x_m^2 & x_m^3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \tag{2}$$

where m is the number of rows or the number of equations available from the dataset. Since the number of rows is greater than the number of columns, it often happens that there is no exact solution for $A\mathbf{x} = \mathbf{b}$ as $\mathbf{b}$ is not in the column space of A. Another vector is chosen such that $\mathbf{e} = \mathbf{b} - \mathbf{p}$, which is the distance between $\mathbf{b}$ and the chosen vector is minimized while $\mathbf{e}$ is perpendicular to the column space. The best possible solution happens to be the projection of $\mathbf{b}$ on $C(A)$ which is $p = A\hat{x}$ where $\hat{x}$ is the least squares solution [2]. By minimizing the squared error, $\mathbf{e}$,

$$\begin{aligned} F(\mathbf{x}) &= \|\mathbf{e}\|^2 = \mathbf{e}^T\mathbf{e} \\ &= (\mathbf{b} - A\mathbf{x})^T(\mathbf{b} - \mathbf{x}) \\ &= \mathbf{b}^T\mathbf{b} - 2\mathbf{b}^T A\mathbf{x}^T + A^T A\mathbf{x} \end{aligned}$$

and setting the gradient to zero,

$$\frac{\partial F(\mathbf{x})}{\partial \mathbf{x}} = -2A^T\mathbf{b} + 2A^T A\mathbf{x} = 0$$

$$A^T A\mathbf{x} = A^T\mathbf{b} \tag{3}$$

the least squares solution can be solved for $\hat{x} = (A^T A)^{-1}A^T\mathbf{b}$ where $A^T A$ is a much more dense invertible square matrix with nonzero entities where the dot product between the Vandermonde matrix A and its transpose A^T will result in the generalized matrix A from Equation 2,

$$A^T A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 \\ x_1^2 & x_2^2 & x_3^2 & x_4^2 \\ x_1^3 & x_2^3 & x_3^3 & x_4^3 \end{bmatrix} \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \\ 1 & x_4 & x_4^2 & x_4^3 \end{bmatrix} = \begin{bmatrix} n & \sum x_i & \sum x_i^2 & \sum x_i^3 \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \sum x_i^4 \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \sum x_i^5 \\ \sum x_i^3 & \sum x_i^4 & \sum x_i^5 & \sum x_i^6 \end{bmatrix}$$

2.2 Multiple Linear Regression (MLR)

Previously, we have looked into linear least squares regression consisting of only singular independent variable. MLR concerns itself in obtaining a best fit for a dataset with multiple independent variables, $(x_{i1}, x_{i2}, ...)$ with respect to the dependent variable, y_i given that the independent variables are not too highly correlated with each other.

A system of equations consisting of multiple independent variables can be solved by using Equation 3. Consider three independent variables of the following function,

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2^2 + \beta_3 x_1 x_2$$

The system of equations, $A\mathbf{x} = \mathbf{b}$ given that A is simply the Vandermonde matrix from Equation 2 [3],

$$\begin{bmatrix} 1 & x_{11} & x_{12}^2 & x_{11}x_{12} \\ 1 & x_{21} & x_{22}^2 & x_{21}x_{22} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{m1} & x_{m2}^2 & x_{m1}x_{m2} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

where m is the number of equations and the parameters β_i of $\mathbf{x}$ can be solved for $\mathbf{x} = (A^T A)^{-1}A^T\mathbf{b}$.

3 Least Squares Function Approximation

Given a function, $f(x)$, it can be approximated to a weighted sum of functions usually chosen to be of orthogonal functions, $\phi(x)$ where the inner product of orthogonal functions are $\langle \phi_i, \phi_j \rangle = 0$ when $i \neq j$ due to its orthogonality properties.

$$f(x) \approx \sum_{i=1}^{n} c_i \phi_i(x) = c_1\phi_1(x) + c_2\phi_2(x) + ... + c_n\phi_n(x) = \hat{f}(x)$$

The approximation function, $\hat{f}(x)$ can best fit the original function by determining the c_i coefficients such that $\|f - \hat{f}\|^2$ is minimum.

$$\left\|f - \hat{f}\right\|^2 = \int_a^b (f(x) - \sum_{i=1}^{n} c_i\phi_i(x))^2 \, dx$$

There are generally two ways of minimizing the squared error. Firstly, by using calculus and taking the derivatives of the squared residuals with respect to c_i to get the minimum points. However, we can take advantage of the orthogonality of $\phi_i(x)$ for the minimization via linear algebra where [1],

$$\begin{aligned}
\left\|f - \hat{f}\right\|^2 &= \langle f - \hat{f}, f - \hat{f} \rangle \\
&= \langle f, f \rangle - 2\langle f, \hat{f} \rangle + \langle \hat{f}, \hat{f} \rangle \\
\langle \hat{f}, \hat{f} \rangle &= \langle \sum_{i=1}^{n} c_i\phi_i, \sum_{j=1}^{n} \phi_j \rangle \\
&= c_1^2\langle \phi_1, \phi_1 \rangle + c_2^2\langle \phi_2, \phi_2 \rangle + ... + c_n^2\langle \phi_n, \phi_n \rangle \\
&= \sum_{i=1}^{n} c_i^2 \|\phi_i\|^2 \\
\langle f, \hat{f} \rangle &= \langle f, \sum_{i=1}^{n} \phi_i \rangle \\
&= \langle f, c_1\phi_1 + c_2\phi_2 + ... + c_n\phi_n \rangle \\
&= c_1\langle f, \phi_1 \rangle + c_2\langle f, \phi_2 \rangle + ... + c_n\langle f, \phi_n \rangle \\
&= \sum_{i=1}^{n} c_i\langle f, \phi_i \rangle
\end{aligned}$$

The critical value of the squared error, $E(c_i)$ can then be computed for a particular coefficient, c_k,

$$\begin{aligned}
\left\|f - \hat{f}\right\|^2 &= \|f\|^2 - 2\sum_{i=1}^{n} c_i\langle f, \phi_i \rangle + \sum_{i=1}^{n} c_i^2\langle \phi_i, \phi_i \rangle = E(c_i) \\
\frac{\partial E}{\partial c_k} &= \frac{\partial}{\partial c_k}\|f\|^2 - 2\sum_{i=1}^{n} \frac{\partial}{\partial c_k} c_i\langle f, \phi_i \rangle + \sum_{i=1}^{n} \frac{\partial}{\partial c_k} c_i^2\langle \phi_i, \phi_i \rangle \\
&= -2\langle f, \phi_k \rangle + 2c_k\langle \phi_k, \phi_k \rangle = 0 \\
c_k &= \frac{\langle f, \phi_k \rangle}{\langle \phi_k, \phi_k \rangle} \qquad (4)
\end{aligned}$$

Thus, the best approximation can be calculated for any given $f(x)$ with any set of orthogonal functions as [1],

$$\hat{f}(x) = \sum_{i=1}^{n} \frac{\langle f, \phi_i \rangle}{\langle \phi_i, \phi_i \rangle} \phi_i(x) \qquad (5)$$

4 Nonlinear Least Squares

A function, $f(x)$ is considered nonlinear when it cannot be expressed as the linear combination of its coefficients such it is nonlinear in its coefficients. A function such as $y = be^{ax}$ is a nonlinear equation that can be linearized by taking the natural logarithm of both sides of the equation, $\ln y = \ln b + ax$ and is able to be approximated by ordinary least squares.

When a model chosen to be approximated to a set of data is a nonlinear function $f(x, \beta)$ that cannot be linearized, the ordinary least squares analysis does not hold true for the model. The Gauss-Netwon approach is used for a nonlinear least squares analysis where the model is approximated by a linear function, commonly the first-order Taylor series [4],

$$r_i = y_i - f(x, \beta)$$

$$r_i \approx y_i - f(x_i, \beta_t) - \nabla_\beta f(x_i, \beta_t)[\beta - \beta_t]$$

we can redefine the Taylor's approximation to a linear function by,

$$\tilde{y}_i = y_i - f(x_i, \beta_t) \qquad \tilde{x}_i = \nabla_\beta f(x_i, \beta_t) \qquad \tilde{\beta} = \beta - \beta_t$$

$$r_i \approx \tilde{y}_i - \tilde{x}_i^T \tilde{\beta} = \tilde{y}_i - J\tilde{\beta}$$

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial \beta_1} & \cdots & \frac{\partial f_1}{\partial \beta_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial \beta_1} & \cdots & \frac{\partial f_n}{\partial \beta_p} \end{bmatrix}$$

where $\tilde{x}_i = J$ is the Jacobian matrix. Now that it is in the form of an ordinary least square with the solution $\mathbf{x} = (A^T A)^{-1} A^T \mathbf{b}$, we can apply this solution to $\|r_i\|^2 = \left\|\tilde{y} - J\tilde{\beta}\right\|^2$ and substitute back $\tilde{y}$ and $\tilde{\beta}$.

$$\tilde{\beta} = (J_t^T J_t)^{-1} J_t^T \tilde{y} = (J_t^T J_t)^{-1} J_t^T r_t$$

$$\beta_{t+1} = \beta_t + (J_t^T J_t)^{-1} J_t^T r_t \tag{6}$$

The solutions for the regression coefficients are obtained and refined by successive approximations until the desired tolerance is achieved.

5 Modelling and Fitting

In this section, we are going to attempt to do least squares regression analysis on several data sets for linear, quadratic and cubic functions. In general, there are several ways to obtain the desired coefficients and best fit of the model. We are mainly going to compute and plot these parameters by using,

i the generalized matrix from Equation 1,

ii the Vandermonde matrix from Equation 2,

iii built-in SciPy modules for least squares and curve fitting.

We are also going to model, fit and plot the approximated relevant functions to series of data points for multiple linear regression (MLR), least squares function approximation and nonlinear least squares.

5.1 Linear model

A simple linear function, $y = a_0 + a_1 x$ is used to model a linear best fit for a set of data, x and y. The data set for this example is generated by using NumPy's random module to give the data some sort of a random noise.

A function is needed to sort the data set into the generalized matrix from Equation 1 before solving the system of equations. To obtain the generalized matrix, the arrays of x and y are passed through a function that sorts the matrix elements iteratively.

```
def get_gmatrices(order, x, y, n):
    z = order + 1
    A = np.empty(shape=(z, z))
    for i in range(z):
        itr = 0
        for j in range(z):
            if i == 0 and j == 0:
                A[i, j] = n
                itr += 1
            else:
                A[i, j] = sum(x ** (i + itr))
                itr += 1
    b = np.empty(shape=(z, 1))
    b[0, 0] = sum(y)
    for k in range(1, z):
        b[k, 0] = sum(y * (x ** k))
    return A, b
```

The function above will give output to the matrix A and $\mathbf{b}$ which can be solved for $\mathbf{x} = A^{-1}\mathbf{b}$ to obtain the coefficients of the linear model, a_0 and a_1, using `np.linalg.solve(A, b)`. The linear best fit is plotted with the coefficients measured in Figure 1. The sum of squared residuals as per previous section is given by $\sum(y - \hat{y})^2$ where $\hat{y}$ is the approximated y values.

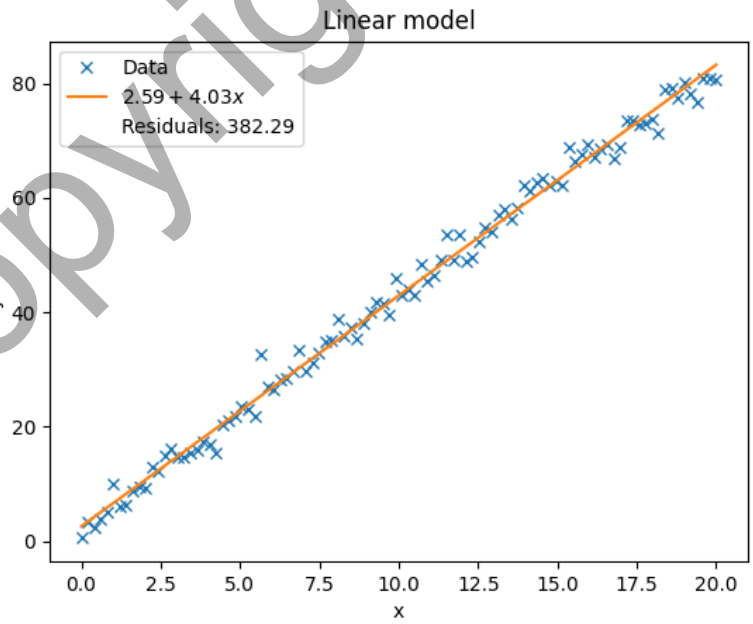


Figure 1: Linear model

5.2 Quadratic model

In addition to the generalized matrix, the coefficients for the best fit can also be obtained by using the Vandermonde matrix. For the quadratic model, the simplest quadratic function $y = a_0 + a_1x + a_2x^2$ is used. The data set for this example is generated via NumPy's random function. Sorting the system of equations in the form of $A\mathbf{x} = \mathbf{b}$, A is a 100×3 matrix where $m > n$ thus rendering solving the system impossible since there are no real solutions.

From the derivation of Equation 3, it is observed that by applying a dot product of the transpose A^T to matrix A, a much denser square matrix is obtained which will make solving the system possible. Now the system of equations can be represented as,

$$A^TA\mathbf{x} = A^T\mathbf{b}$$

A function that takes the values of x and y is constructed such that it sorts the former into a Vandermonde matrix, A and the latter into a column matrix of y values for all the data points.

```
def get_lamatrices(order, x, y, n):
    power = list(range(order + 1))
    M = x[:, np.newaxis] ** power  # [1 x x^2 x^3 ...] form
    k = y.reshape(n, 1)

    return M, k
```

The function `np.newaxis` is extremely useful in constructing the Vandermonde matrix as it adds a new dimension to the array and is able to transform the elements added as seen above where every new column or dimension, the elements of said column is raised to a power.The system of equations can be solved for, in this case, $\mathbf{x} = (M^TM)^{-1}M^T\mathbf{k}$, where the coefficients are obtained and the quadratic best fit is plotted as seen in Figure 2.

```
c = np.linalg.inv(M.T@M)@M.T@k
```

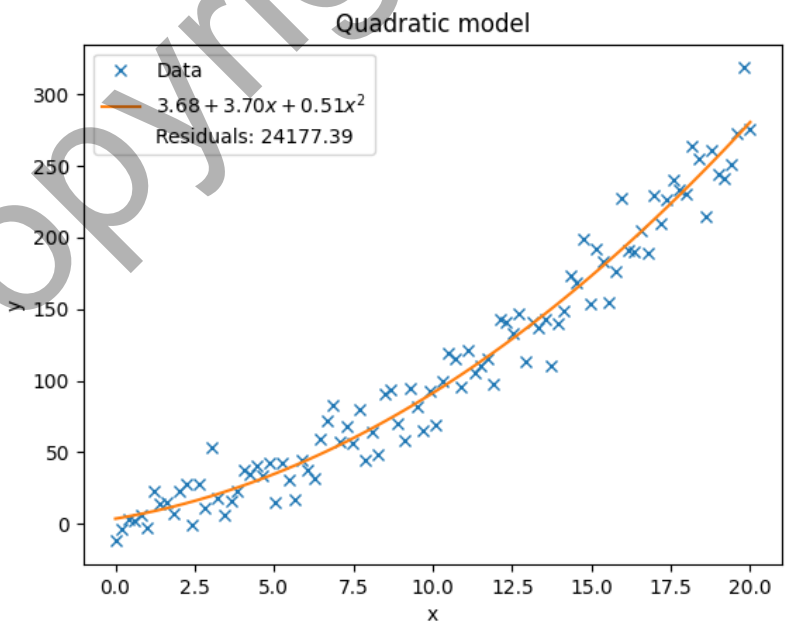


Figure 2: Quadratic model

5.3 Cubic model

As we have seen both ways of solving the system of equations to measure the coefficients of the selected model through the generalized and Vandermonde matrices, it is only apt for us to utilize the already existing algorithms built into the NumPy and SciPy module.

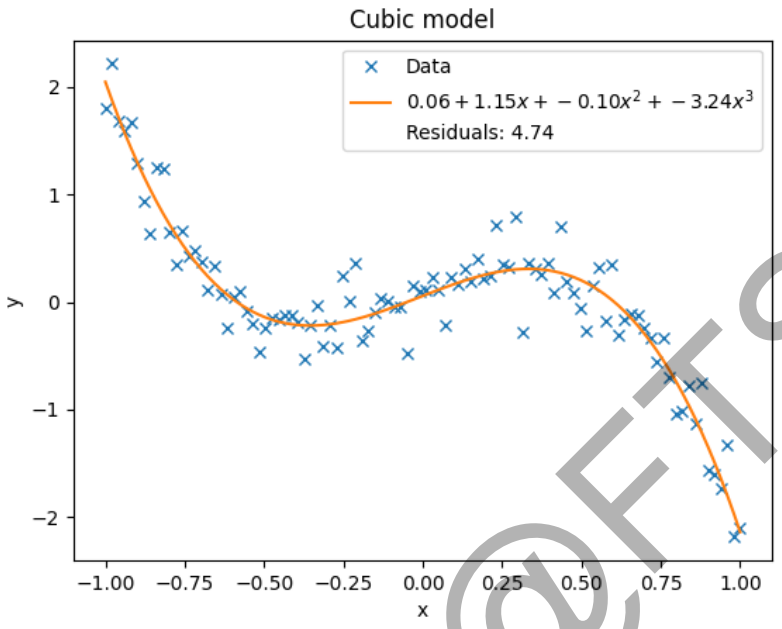


Figure 3: Cubic model using `scipy.linalg.lstsq`

The first method is `np.linalg.lstsq` and `scipy.linalg.lstsq` where it takes the Vandermonde matrix A and $\mathbf{b}$ as its argument and yields the least squares solutions, its sum of squared residuals, rank of matrix A and singular values of A [5].

```
from scipy.linalg import lstsq

z, res, rank, s = lstsq(M, k)
print('Coefficients:', z.flatten())
print('Residuals', res)
```

The second method, `scipy.optimize.curve_fit` is advanced such that it uses non-linear least squares to fit a desired function to data [5, 6]. The method takes the desired function or model to be approximated to and the arrays of values for the data set, x and y with some other optional arguments such as initial guesses, uncertainty for y and bounds. The method yields the optimal values of the coefficients for the model that minimized the squared residuals and the estimated covariance for the coefficients.

```
def func(x, a, b, c, d):
    return a*x**3 + b*x**2 + c*x + d

x = np.linspace(0, 20, 100)
y = func(x, 3, 1, 4, 9) + np.random.normal(0, 1000, x.shape)

coeff, cov = scipy.optimize.curve_fit(func, x, y)
```

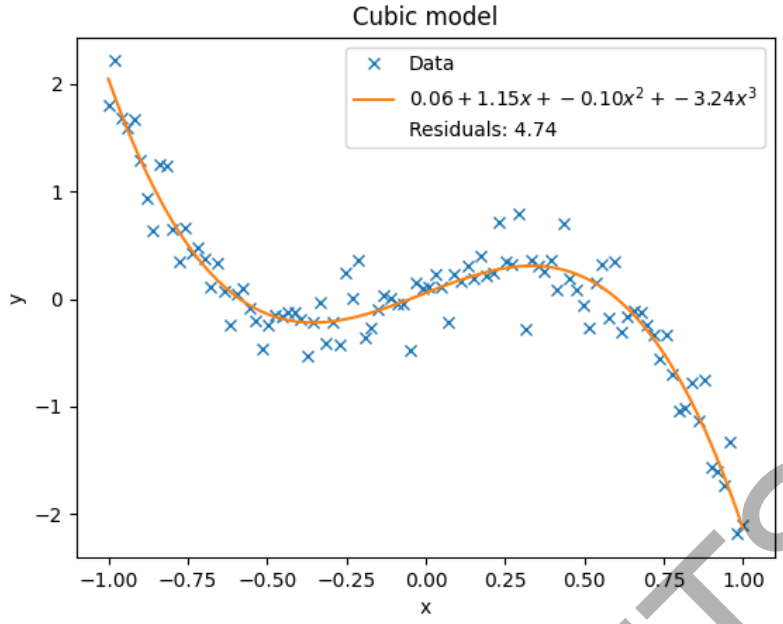


Figure 4: Cubic model using `scipy.optimize.curve_fit`

There are some other implementations that could be of use in regards to curve fitting via least squares such as NumPy's `polyfit`, SciPy's `least_squares`,`leastsq` and scikit-learn's `linear_model.LinearRegression`.

All of the models used above are polynomial models of degree 1, 2 and 3 respectively. A general polynomial regression Python code that utilizes the three aforementioned methods — the generalized matrix, the Vandermonde matrix and built-in SciPy modules — is attached as Appendix A and another Python code that uses the module scikit-learn is available in Appendix B.

5.4 Example: Global annual mean temperature

A data set of the global annual mean temperature anomalies over the course of 1880-2020 is obtained from NASA [7, 8]. By solving for the coefficients using Equation 3, the least squares polynomial regressions of degree 1, 2 and 3 are calculated and plotted against each other.

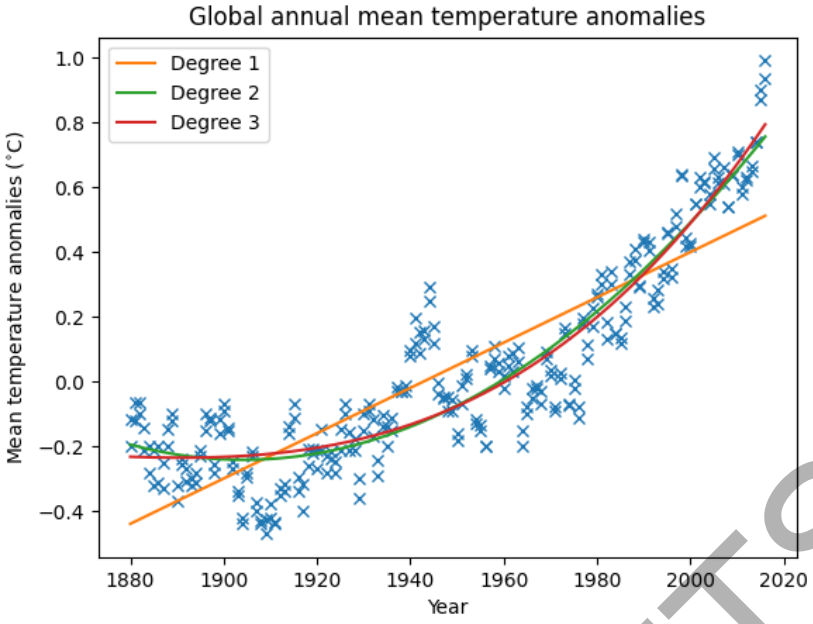


Figure 5: Polynomial models of degree 1, 2 and 3

The sum of squared residuals for degree 1, 2 and 3 are measured to be 7.0454, 3.6274 and 3.5670 respectively. The smallest sum of squared residuals is obtained when the cubic model is used for regression. By utilizing every methods and modules to obtain the least squares solutions as shown previously, it is observed that there is no significant discrepancy between the solutions for the cubic model.

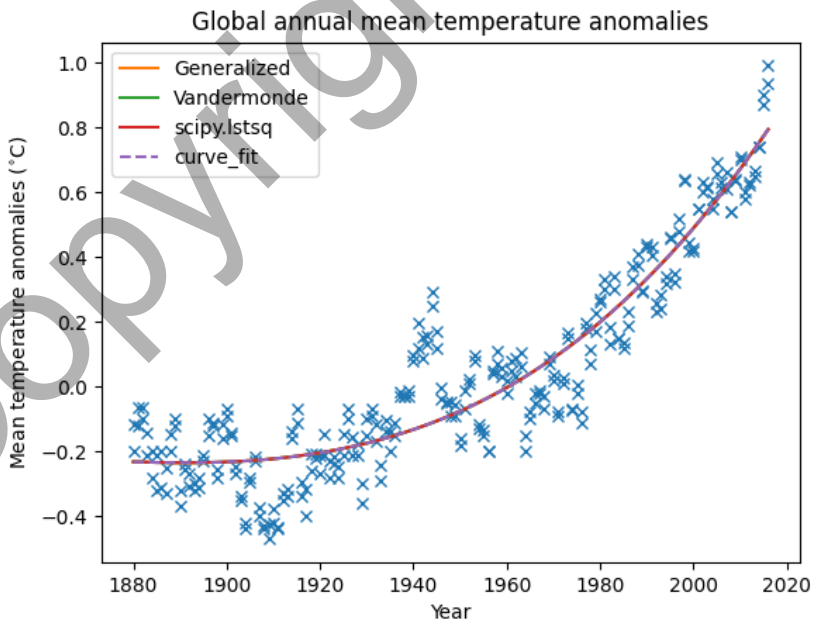


Figure 6: Cubic model

```
>> [-1.96857536e+03  3.17484235e+00 -1.70649924e-03  3.05672104e-07]
   Generalized
>> Residuals: 3.5670440372642203
```

```
>> [-1.96853127e+03  3.17477441e+00 -1.70646436e-03  3.05666135e-07]
   Vandermonde
>> Residuals: 3.5670440371850045
>> [-1.96847245e+03  3.17468377e+00 -1.70641782e-03  3.05658170e-07] lstsq
>> Residuals: 3.5670440470114393 [3.56704404]
>> [-1.97014946e+03  3.17726763e+00 -1.70774452e-03  3.05885183e-07]
   curve_fit
>> Residuals: 3.567044070463524
```

5.5 Multiple linear regression

Given a data set where there are multiple independent variables, x, the best fit can be obtained by solving a system of equation in the form of Equation 3 where A is the Vandermonde matrix and the elements is sorted based on the chosen model.

A data set of the US economy from 1976 to 1987 has several interesting variables and indicators such as the gross national product (GNP), the purchasing power of US dollar, the price of crude oil per barrel, the amount of foreign investments among others. The first two indicators, GNP (x_1) and purchasing power (x_2) are chosen against the consumer debt (in billions) as the dependent variable y and a linear model is chosen,

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

For this least squares multiple linear regression computation, scikit-learn's `linear_model.LinearRegression` method is used for the computation of the regression coefficients. The matrix A is constructed by stacking a column matrix of 1s and the respective column values of x_1 and x_2 from the dataframe [9, 10].

```
X = df[['PURCHASE', 'GNP']].values.reshape(-1,2)
y = df['CONSUMER'].values

x1 = X[:, 0]
x2 = X[:, 1]

x1_pred = np.linspace(min(x1), max(x1), 100)
x2_pred = np.linspace(min(x2), max(x2), 100)
x1m_pred, x2m_pred = np.meshgrid(x1_pred, x2_pred)
modelxs = np.array([x1m_pred.flatten(), x2m_pred.flatten()]).T

ols = linear_model.LinearRegression()
model = ols.fit(X, y)
print('c', model.coef_, model.intercept_)
predicted = model.predict(modelxs) # for the best fit plotting
```

The best fit is plotted where the dependent variable, y is on the z-axis and the two independent variables on x and y-axis. Based on the result of the fit, the linear regression model obtained is,

$$y = -7.855 \times 10^2 + (3.361 \times 10^2)\, x_1 + (2.618 \times 10^{-1})\, x_2$$

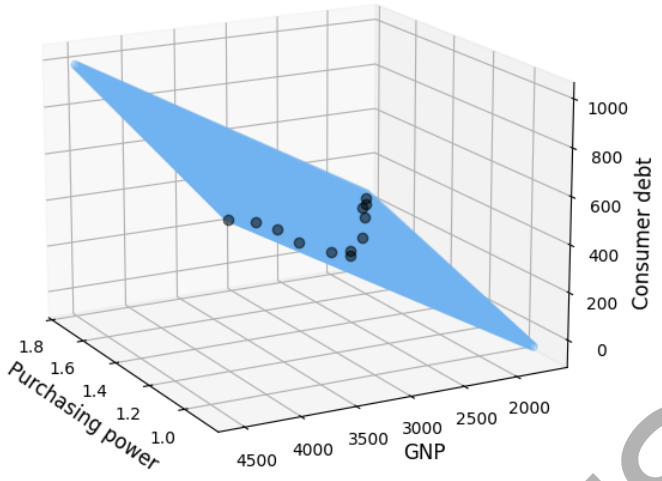


Figure 7: Gross national product (GNP) and purchasing power of US dollar against consumer debt (in billions) between 1976-1987 in USA.

As the model contains more independent variables, it will be difficult to visualize the multidimensional plot for the best fit regression but it is entirely possible to measure its regression coefficients. However, it is good practice to compute the correlation between the independent variables as to avoid multicollinearity which can introduce instability in the regression coefficients. The values for some of the collinear coefficients may be unreliable, but it is still useful for prediction of that particular model. It is a concern when we are trying to deduce a conclusion of any individual independent variable on the dependent variable [9].

5.6 Function approximation of $\sin \pi x$

Given a function $f(x) = \sin \pi x$ over $[-1, 1]$, it is possible to approximate a sum of functions, $\hat{f}(x)$ that would yield a close match to the original function $f(x)$ where,

$$\hat{f}(x) = \sum_{i=1}^{n} c_i \phi_i(x) = \hat{f}(x) = \sum_{i=1}^{n} \frac{\langle f, \phi_i \rangle}{\langle \phi_i, \phi_i \rangle} \phi_i(x)$$

The function, $\phi_i(x)$ is usually chosen to be of orthogonal functions in which case, the Legendre polynomials are chosen here [1]. Since a sinusoidal over $[-1, 1]$ resembles a cubic function, it is highly possible that a combination of the Legendre polynomials of the third order would result in a cubic function would minimize the sum of squared residuals. Hence, $\hat{f}(x)$ is chosen to be iterated until the third order of the Legendre polynomials $\phi_i(x) = P_i(x)$ for $i = 0, 1, 2, 3$ which are readily available as functions in a SciPy method, `scipy.special.legendre`.

$$P_0(x) = 1 \qquad P_1(x) = x \qquad P_2(x) = \frac{1}{2}(3x^2 - 1) \qquad P_3(x) = \frac{1}{2}(5x^3 - 3x)$$

To calculate for the coefficient, c_i, the inner product of $\langle f, \phi_i \rangle$ and $\langle \phi_i, \phi_i \rangle$ can be measured by using the integration method available through SciPy, `scipy.integrate.quad` [5].

```
func = lambda x: np.sin(np.pi*x)
func2 = lambda x: 0.5(3*x**2-1)
func_product = lambda x: func(x)*func2(x) # Multiplying two functions
```

```
a, b = -1, 1

integral_f = quad(func, a, b)
integral_comb = quad(func_product, a, b)
```

After being able to measure all the coefficients, the approximated function $\hat{f}(x)$ can be obtained by summing all the product of coefficients and Legendre polynomials and plotted against the actual original function.

```
def y(order, x, coeff):
    polyn = 0
    for i in range(order + 1):
        polyn += coeff[i] * legendre(i)(x)
    return polyn
```

$$\hat{f}(x) = (0.955)x + (4.213 \times 10^{-18})x^2 + (-1.158)x^3$$

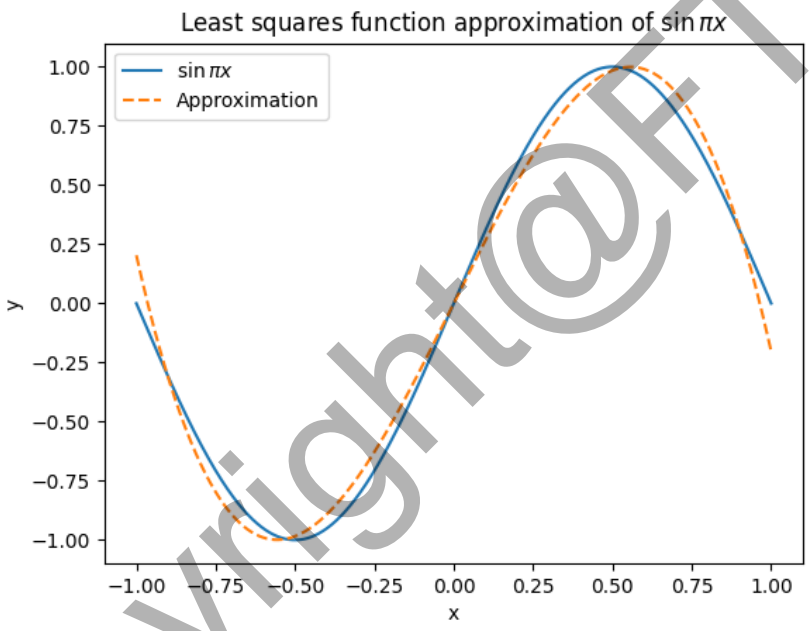


Figure 8: $\sin \pi x$ approximation using Legendre polynomials

5.7 Nonlinear regression

Given a nonlinear function to be approximated with,

$$f(x) = \frac{ax}{b + e^x}$$

there are three components of importance, β_t, J_t and r_t, that are needed for the optimal least squares solutions,

$$\beta_{t+1} = \beta_t + (J_t^T J_t)^{-1} J_t^T r_t$$

Since solving a nonlinear least squares regression is done iteratively, an initial guess condition for β needs to be defined along with a function that computes the Jacobian matrix and the column matrix of the residuals r_t.

The Jacobian matrix is essentially a matrix of partial derivatives of the function with respect to each of the coefficients. These partial derivatives can be calculated numerically by using the finite difference method where h is a small change in x [1],

$$\frac{\partial f}{\partial a} = \frac{f(x+h) - f(x-h)}{2h}$$

```
def Jacobian(f, x, a, b):
    h = 1e-6
    pdv_a = (f(x, a+h, b)-f(x, a-h, b))/(2*h)
    pdv_b = (f(x, a, b+h)-f(x, a, b-h))/(2*h)
    return np.column_stack([pdv_a, pdv_b])
```

The Gauss-Newton equation can be made into a function that takes the nonlinear function, the values of the data set, the initial guess values for a and b and the number of iterations.

```
def GaussNewton(f, x, y, a0, b0, iterations): # Gauss-Newton
    tol = 1e-12
    g_i = g = np.array([a0, b0])
    for itr in range(iterations):
        g = g_i
        J = Jacobian(f, x, g[0], g[1])
        r = y - f(x, g[0], g[1])
        g_i = g + np.linalg.inv(J.T@J)@J.T@r
        if np.linalg.norm(g-g_i) < tol:
            print(np.linalg.norm(g-gi), 'test')
            break
    return g
```

By calling the Gauss-Newton function for a certain data set and guess values, it will return the values of the coefficients that can be plotted.

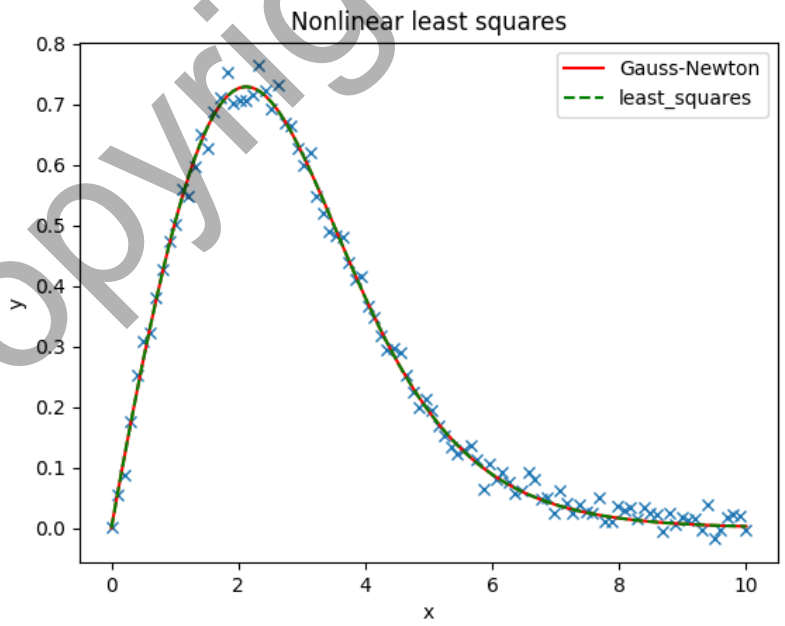


Figure 9: Nonlinear least squares of function $f(x) = \frac{ax}{b+e^x}$.

6 Conclusion

The least squares method is one of the most widely used and reliable approach in regression analysis with the ability to approximate data set with a wide range of functions. In this report, we have shown several models that can be approximated to such as the linear, quadratic and cubic models. The least square solutions to these models have been computed using the various methods we have discussed and shown no significant variation of the coefficients and the residuals between the methods – generalized matrix, Vandermonde matrix and built-in modules. We have also studied, modelled and fitted other types of regression such as multiple linear regression (MLR), least squares function approximation and nonlinear regression. This work is part of the underlying study for the project "*Implementing quantum algorithm for least squares data fitting through IBM-Q*".

Acknowledgements

This work is related to the ongoing research by Ts. Dr. Bahari Idrus and Assoc. Professor. Dr. Mohammad Khatim Hasan funded by GUP-2020-061.

References

[1] J.F. Epperson. *An Introduction to Numerical Methods and Analysis*. Wiley, 2013. ISBN: 9781118367599. URL: https://books.google.com.my/books?id=310lAgAAQBAJ.

[2] Gilbert Strang. *Introduction to Linear Algebra*. Fourth. Wellesley, MA: Wellesley-Cambridge Press, 2009. ISBN: 9780980232714 0980232716 9780980232721 0980232724 9788175968110 8175968117.

[3] Nathaniel E. Helwig. *Multiple Linear Regression*. Jan. 2017.

[4] Geof H. Givens and Jennifer A. Hoeting. *Computational Statistics*. eng. 2nd ed. Hoboken, N.J: Wiley, 2013. ISBN: 9780470533314.

[5] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. 2001–. URL: http://www.scipy.org/.

[6] Eric Ayars. *Computational Physics With Python*. California State University, 2013.

[7] . *GISS Surface Temperature Analysis (GISTEMP), version 4*. publisher: NASA Goddard Institute for Space Studies. 2021. URL: https://data.giss.nasa.gov/gistemp/.

[8] Nathan J. L. Lenssen et al. "Improvements in the GISTEMP Uncertainty Model". en. In: *Journal of Geophysical Research: Atmospheres* 124.12 (June 2019), pp. 6307–6326. ISSN: 2169-897X, 2169-8996. DOI: 10.1029/2018JD029522. URL: https://onlinelibrary.wiley.com/doi/abs/10.1029/2018JD029522 (visited on 04/14/2021).

[9] Eric Kim. *Multiple Linear Regression and Visualization in Python*. Nov. 2019. URL: https://aegis4048.github.io/mutiple_linear_regression_and_visualization_in_python.

[10] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

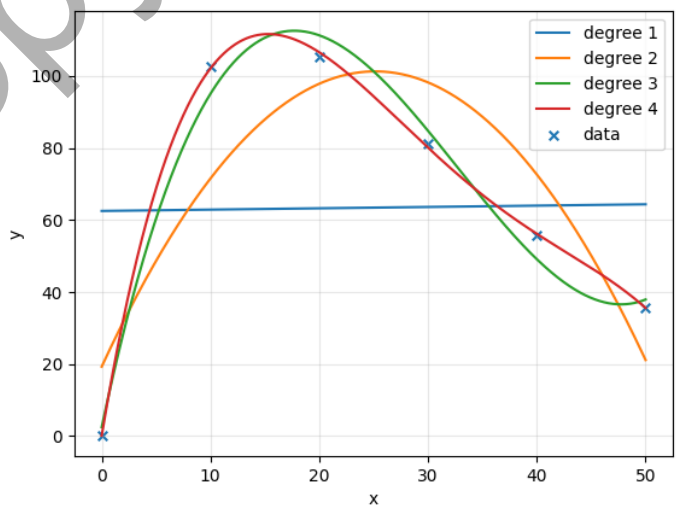
A Polynomial regression

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.linalg import lstsq
from scipy.optimize import curve_fit

def func(order, x, *a):
    polyn = 0
    i = 0
    for p in range(order + 1):
        polyn += a[i] * x ** p
        i += 1
    return polyn

def get_lamatrices(order, x, y, n):
    power = list(range(order + 1))
    M = x[:, np.newaxis] ** power  # [1 x x^2 x^3 ...] form
    k = y.reshape(n, 1)
    return M, k

def get_gmatrices(order, x, y, n):
    z = order + 1
    A = np.empty(shape=(z, z))
    for i in range(z):
        itr = 0
        for j in range(z):
            if i == 0 and j == 0:
                A[i, j] = n
                itr += 1
            else:
                A[i, j] = sum(x ** (i + itr))  # Generalized matrix form for
    A
                itr += 1

    b = np.empty(shape=(z, 1))
    b[0, 0] = sum(y)
    for k in range(1, z):
        b[k, 0] = sum(y * (x ** k))

    return A, b

def get_error(y, fh):
    return np.sum((y - fh) ** 2)

df = pd.read_csv('annualglobaltemp.csv')
xdata = df[['Year']].to_numpy().reshape(-1).T  # Down one dimension,
   transpose
ydata = df[['Mean']].to_numpy().reshape(-1).T

plt.plot(df['Year'], df['Mean'], 'x')

n = xdata.size
x, y = xdata.astype('longdouble'), ydata
xlin = np.linspace(min(x), max(x), 1500)
orderlst = [3]

def fcv(x, *s):
    return s[0] + s[1]*x + s[2]*x**2 + s[3]*x**3
```

```
57
58 for order in orderlst:
59     A, b = get_lamatrices(order, x, y, n)
60     M, k = get_gmatrices(order, x, y, n)
61
62     d = np.linalg.solve(M, k)
63     d_err = get_error(y, func(order, x, *d.flatten()))
64     print(d.flatten(), 'Generalized')
65     print('Residual:', d_err)
66
67     c = np.linalg.inv(A.T @ A) @ A.T @ b
68     print(c.flatten(), 'Vandermonde')
69     print('Residuals:', get_error(y, func(order, x, *c.flatten())))
70
71     z, res, rank, s = lstsq(A, b)  # SVD decomposition
72     print(z.flatten(), 'lstsq')
73     z_err = get_error(y, func(order, x, *z.flatten()))
74     print('Residuals:', z_err, res)
75
76     p, _ = curve_fit(fcv, x, y, p0=[1, 1, 1, 1])
77     y_cv = fcv(xlin, *p)
78     print(p, 'curve_fit')
79     print('Residuals:', get_error(y, fcv(x, *p)))
80
81     #print(np.linalg.norm(y-func(order, x, *c.flatten())))
82
83     plt.plot(xlin, func(order, xlin, *d.flatten()), label=r'Generalized')
84     plt.plot(xlin, func(order, xlin, *c.flatten()), label=f'Vandermonde')
85     plt.plot(xlin, func(order, xlin, *z.flatten()), label=r'scipy.lstsq')
86     plt.plot(xlin, y_cv, '--', label=r'curve_fit')
87
88 plt.legend()
89 plt.xlabel('Year')
90 plt.ylabel('Mean temperature anomalies ($^{\circ}$C)')
91 plt.title('Global annual mean temperature anomalies')
92 plt.show()
```

B Polynomial regression – sci-kit learn

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

x_plot = np.linspace(0, 50, 1000)

x = np.array([0, 10, 20, 30, 40, 50], dtype=np.longdouble)
y = np.array([0, 102.6903, 105.4529, 81.21744, 55.6016, 35.6859])

# Column matrix
X = x[:, np.newaxis]
X_plot = x_plot[:, np.newaxis]

plt.scatter(x, y, s=30, marker='x', label='data')
degreelst = [1, 2, 3, 4]

for degree in degreelst:
    model = make_pipeline(PolynomialFeatures(degree), LinearRegression(
    fit_intercept=False))
    model.fit(X, y)
    polyn = model.named_steps['linearregression']
    coeff = polyn.coef_
    print('Coefficients:', coeff)
    print('R^2:', model.score(X, y))
    y_plot = model.predict(X_plot)
    plt.plot(x_plot, y_plot, label=f'degree {degree}')

plt.grid(alpha=0.3)
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



C Least squares function approximation

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import legendre
from scipy.integrate import quad

func = lambda x: np.sin(np.pi*x) # Original function to be approximated
orth = [lambda x: 1, lambda x: x, lambda x: x**2-(1/3), lambda x: x**3-(3/5)
    *x] # Test

def c(f, phi2, a, b):
    num = quad(f, a, b) # Integral of f, limits a to b
    dnm = quad(phi2, a, b)
    return num[0]/dnm[0] # c_i = \frac{\langle f,\phi_i \rangle}{\langle \
    phi_i,\phi_i \rangle}

def y(order, x, coeff):
    polyn = 0
    for i in range(order + 1):
        polyn += coeff[i] * legendre(i)(x) # \hat{f}(x) = \sum_{i=1}^n c_i \
    phi_i(x)
    return polyn

def sqerror(y, fhat):
    return np.sum((y-fhat)**2)

def main():
    order = 3
    a, b = -1, 1
    coeff = []
    xlin = np.linspace(a, b, 1000)

    for i in range(order+1):
        phi = legendre(i) # Orthogonal function -- Legendre polynomials

        f = lambda x: func(x)*phi(x) # Combining 2 functions
        phi2 = lambda x: phi(x)*phi(x)

        coeff.append(c(f, phi2, a, b))

    print(coeff)
    print('Squared error:', sqerror(func(xlin), y(order,  xlin, coeff)))

    plt.plot(xlin, func(xlin), label=r'$\sin\,{\pi x}$')
    plt.plot(xlin, y(order,  xlin, coeff), '--', label=r'Approximation')
    plt.legend()
    plt.title('Least squares function approximation of $\sin\,{\pi x}$')
    plt.show()

if __name__ == '__main__':
    main()
```

D Multiple linear regression

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import linear_model
import seaborn as sn

df = pd.read_csv('usecons7687.csv')

"""
CRUDE = dollars/barrel crude oil
INTEREST = % interest on ten yr. U.S. treasury notes
FOREIGN = foreign investments/billions of dollars
DJIA = Dow Jones industrial average
GNP = GNP/billions of dollars
PURCHASE = purchasing power U.S. dollar (1983 base)
CONSUMER = consumer debt/billions of dollars

data from https://college.cengage.com/mathematics/brase/understandable_statistics/7e/students/datasets/mlr/frames/frame.html
"""

X = df[['PURCHASE', 'GNP']].values.reshape(-1,2)
y = df['CONSUMER'].values

x1 = X[:, 0]
x2 = X[:, 1]

x1_pred = np.linspace(min(x1), max(x1), 100)
x2_pred = np.linspace(min(x2), max(x2), 100)
x1m_pred, x2m_pred = np.meshgrid(x1_pred, x2_pred)
modelxs = np.array([x1m_pred.flatten(), x2m_pred.flatten()]).T

ols = linear_model.LinearRegression()
model = ols.fit(X, y)
print(model.coef_)
predicted = model.predict(modelxs)
print(predicted)

r2 = model.score(X, y)

plt.style.use('default')
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')

ax.plot(x1, x2, y, color='k', zorder=15, linestyle='none', marker='o', alpha=0.5)
ax.scatter(x1m_pred.flatten(), x2m_pred.flatten(), predicted, facecolor=(0,0,0,0), s=20, edgecolor='#70b3f0')
ax.set_xlabel('Purchasing power', fontsize=12)
ax.set_ylabel('GNP', fontsize=12)
ax.set_zlabel('Consumer debt', fontsize=12)
ax.view_init(elev=28, azim=120)
fig.tight_layout()
plt.show()

corrMatrix = df.corr(method='pearson')
sn.heatmap(corrMatrix, annot=True)
plt.show()
```

E Nonlinear least squares

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import least_squares

def func(x, a, b):
    return a*x/(b+np.exp(x))

def Jacobian(f, x, a, b):
    h = 1e-6
    pdv_a = (f(x, a+h, b)-f(x, a-h, b))/(2*h) # Finite difference: pdvF = (F(x+h)-F(x-h))/(2h)
    pdv_b = (f(x, a, b+h)-f(x, a, b-h))/(2*h)
    return np.column_stack([pdv_a, pdv_b])

def GaussNewton(f, x, y, a0, b0, iterations): # Gauss-Newton
    tol = 1e-12
    g_i = g = np.array([a0, b0])
    for itr in range(iterations):
        g = g_i
        J = Jacobian(f, x, g[0], g[1])
        r = y - f(x, g[0], g[1])
        g_i = g + np.linalg.inv(J.T@J)@J.T@r # \beta_{i+1} = \beta{i} + (J_i^T J_i)^{-1} J_i^T r_i
        if np.linalg.norm(g-g_i) < tol:
            break
    return g

def r(a, x, y):
    return a[0]*x/(a[1]+np.exp(x)) - y

def main():
    x = np.linspace(0, 10, 50)
    y = func(x, 6, 9) + np.random.normal(0, 0.02, x.shape) # + Noise

    a, b = GaussNewton(func, x, y, 2.5, 0.6, 50)
    print(a, b)
    y_hat = func(x, a, b)

    a_guess = np.array([2, 3])
    res = least_squares(r, a_guess, args=(x, y)) # Scipy's least squares module
    res_cauchy = least_squares(r, a_guess, loss='cauchy', f_scale=0.02, args=(x, y)) # Cauchy loss function
    print(res.x)
    y_hatscp = func(x, res.x[0], res.x[1])

    plt.plot(x, y, 'x')
    plt.plot(x, y_hat, label=r'Gauss-Newton')
    plt.plot(x, y_hatscp, label=r'least_squares')
    plt.legend()
    plt.show()

if __name__ == '__main__':
    main()
```