Curve Fitting Using Python

Cosmic Charade

Curve Fitting in Python

Curve fitting is the process of constructing a curve, or mathematical function, that best fits a series of data points. It is a common task in data analysis, scientific computing, and engineering to model relationships between variables.

In Python, curve fitting can be easily implemented using libraries such as **NumPy**, **SciPy**, and **Matplotlib**.

1. Key Concepts in Curve Fitting

- **Dependent and Independent Variables**:
  - **Independent variable** (x): The variable that you control.
  - **Dependent variable** (y): The output that depends on the independent variable.
- **Goal**: To find a mathematical model that describes the relationship between x and y. This relationship is often expressed as:

$$y = f(x)$$

  The function $f(x)$ could be linear, polynomial, exponential, etc.
- **Least-Squares Method**: A common technique used in curve fitting where the function is fitted to minimize the sum of the squares of the differences between the observed values and the values predicted by the function.

2. Tools for Curve Fitting in Python

To perform curve fitting in Python, we primarily use the following libraries:

- `NumPy`: Provides numerical methods and array operations.
- `SciPy`: Offers functions for scientific computing, including optimization and curve fitting.
- `Matplotlib`: For visualizing data and fitting results.

Installation of Required Libraries

You can install the required libraries using `pip`:

```
pip install numpy scipy matplotlib
```

3. Types of Curve Fitting

a) Linear Curve Fitting:

Linear fitting is one of the most fundamental techniques in data analysis, used to model the relationship between two variables by fitting a straight line through a set of data points. It is particularly useful for identifying trends and making predictions based on observed data.

In Python, you can easily perform linear fitting using the **`NumPy`** library, which provides a convenient function `numpy.polyfit()` for this task.

1. What is Linear Fitting?

Linear fitting, also known as **linear regression**, aims to fit a line that best represents the relationship between a dependent variable `y` and an independent variable `x`. The general equation of a straight line is:

$$y = mx + c$$

Where:

- `m` is the slope of the line.
- `c` is the y-intercept, the point where the line crosses the y-axis.

The objective of linear fitting is to find the values of `m` and `c` that minimize the differences between the actual data points and the points predicted by the line.

2. NumPy's `polyfit()` Function

In NumPy, the `polyfit()` function is used to fit a polynomial of a specified degree to data. Since a linear fit is just a polynomial of degree 1 (a straight line), you can use `polyfit()` for this purpose.

The syntax for `numpy.polyfit()` is:

```
numpy.polyfit(x, y, deg)
```

Where:

- `x` is the array of independent variable (input data).
- `y` is the array of dependent variable (output data).
- `deg` is the degree of the polynomial (for linear fitting, this is 1).

`polyfit()` returns the coefficients of the polynomial in descending powers of `x`. For linear fitting, it will return the slope (`m`) and the intercept (`c`).

3. Example: Linear Fit Using NumPy

Let's walk through a basic example to illustrate linear fitting.

Step 1: Import Required Libraries

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

Step 2: Define Your Data

Suppose you have a set of data points for `x` and `y`:

```
# Independent variable (x)
x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# Dependent variable (y) with some random noise
y = np.array([1, 3, 2.5, 5, 4.5, 6, 6.5, 8, 7.5, 9])
```

Step 3: Perform Linear Fitting

Use `numpy.polyfit()` to calculate the slope and intercept of the best-fit line.

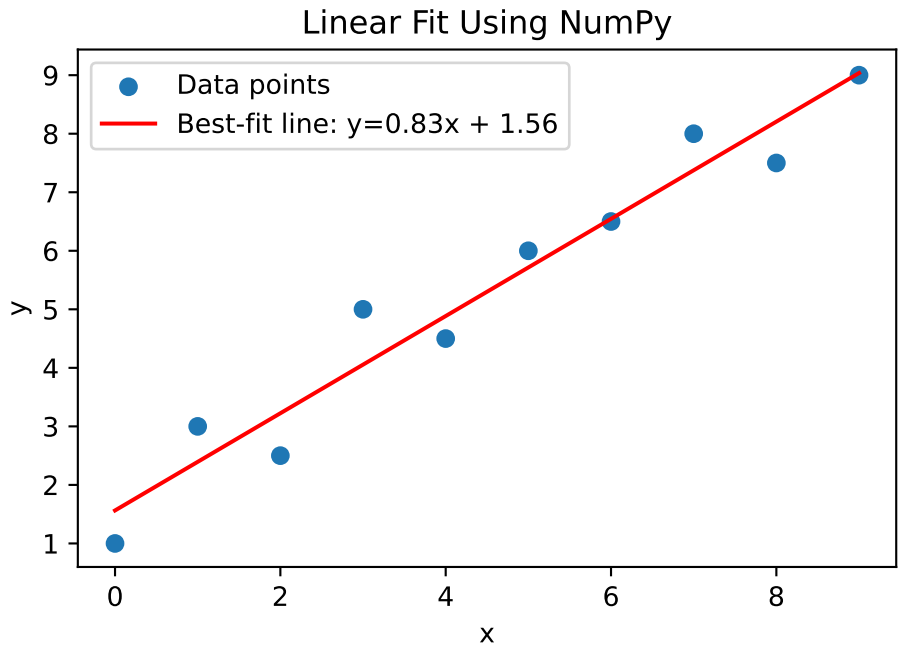
```
# Fit a linear model (degree 1)
m, c = np.polyfit(x, y, 1)

# Print the slope and intercept
print(f"Slope (m): {m}")
print(f"Intercept (c): {c}")
```

```
Slope (m): 0.8303030303030303
Intercept (c): 1.5636363636363626
```

Step 4: Visualize the Data and the Fit

Once you have the slope (`m`) and intercept (`c`), you can calculate the predicted values of `y` (denoted as `y_fit`) and plot the data points and the best-fit line.

```
# Calculate the predicted y values based on the linear fit
y_fit = m * x + c

# Plot the original data points
plt.scatter(x, y, label='Data points')

# Plot the best-fit line
plt.plot(x, y_fit, color='red', label=f'Best-fit line: y={m:.2f}x + {c:.2f}')

# Add labels and legend
plt.xlabel('x')
plt.ylabel('y')
plt.title('Linear Fit Using NumPy')
plt.legend()
plt.show()
```



In this example:

- We first plotted the raw data points using `plt.scatter()`.
- Then, we computed the `y_fit` values using the linear equation `y_fit = m * x + c`.
- Finally, we plotted the best-fit line using `plt.plot()`.

Output

This will produce a scatter plot of the data points and a red line representing the best-fit line through those points.

4. Evaluating the Fit

One way to evaluate the goodness of the fit is to calculate the **R-squared** value, which represents the proportion of the variance in the dependent variable that is predictable from the independent variable. The closer this value is to 1, the better the fit.

The R-squared value can be computed as:

$$R^2 = 1 - \frac{\sum (y - y_{fit})^2}{\sum (y - \bar{y})^2}$$

Where:

- y_{fit} is the predicted `y` value from the linear fit.
- $\bar{y}$ is the mean of the observed `y` values.

You can compute the R-squared value in Python as follows:

```
# Calculate the residual sum of squares (SS_res)
ss_res = np.sum((y - y_fit)**2)

# Calculate the total sum of squares (SS_tot)
ss_tot = np.sum((y - np.mean(y))**2)

# Calculate R-squared
r_squared = 1 - (ss_res / ss_tot)

print(f"R-squared: {r_squared:.2f}")
```

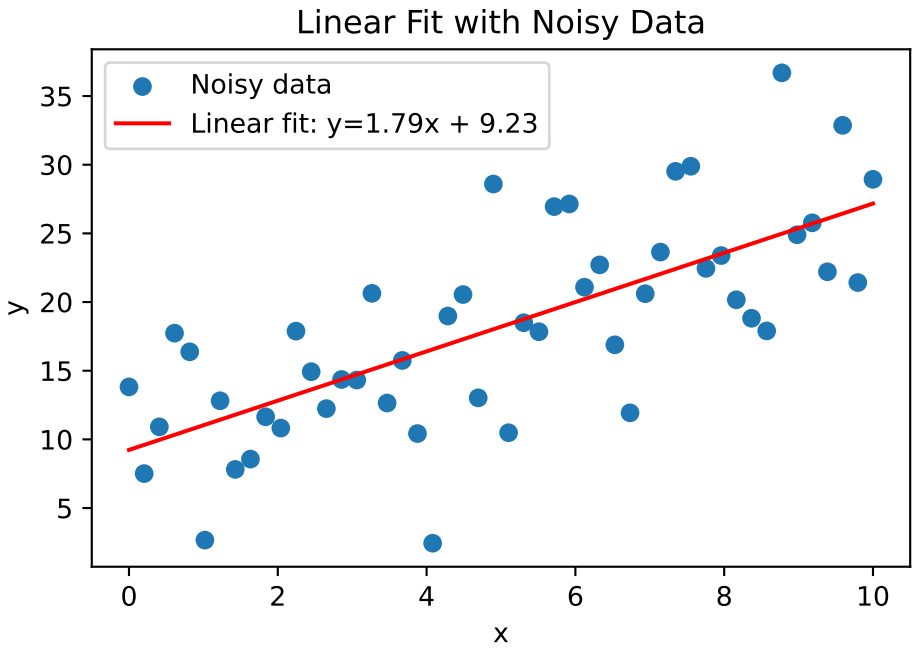
```
R-squared: 0.95
```

5. Example: Noisy Data Linear Fit

Real-world data is often noisy, meaning it has variability that isn't explained by a simple linear relationship. Let's add some noise to our data and see how the linear fit adapts.

```
# Generating noisy data
np.random.seed(0)  # For reproducibility
x = np.linspace(0, 10, 50)
y = 2.5 * x + 5 + np.random.randn(50) * 5

# Line with slope=2.5, intercept=5 plus random noise

# Perform linear fit
m, c = np.polyfit(x, y, 1)

# Calculate predicted values
y_fit = m * x + c

# Plot the data and the linear fit
plt.scatter(x, y, label='Noisy data')
plt.plot(x, y_fit, color='red', label=f'Linear fit: y={m:.2f}x + {c:.2f}')
plt.xlabel('x')
plt.ylabel('y')
```

```
plt.title('Linear Fit with Noisy Data')
plt.legend()
plt.show()
```



In this example, we added random noise to the data using `np.random.randn()`, making the data more realistic and challenging to fit. Despite the noise, `numpy.polyfit()` will still find the best-fit line that minimizes the errors.

6. Practical Applications of Linear Fitting

Linear fitting has numerous applications in real-world scenarios:

- **Predicting Trends**: Linear models are often used to predict trends in financial markets, weather, and sales data.
- **Calibration**: Linear fitting is used to calibrate instruments by finding the relationship between input and output measurements.
- **Simple Machine Learning Models**: Linear regression is a core technique in machine learning, forming the basis for more complex models.
- **Data Analysis**: In scientific research, linear fitting helps in analyzing experimental data to understand relationships between variables.

Understanding linear fitting through NumPy is essential for data analysis, predictive modeling, and scientific research, especially when working with large datasets or noisy real-world data.

b) Polynomial Curve Fitting

Polynomial curve fitting is a technique used to model relationships between variables when a linear fit is insufficient to capture the underlying trends in the data. Instead of fitting a straight line (as in linear fitting), polynomial curve fitting fits a polynomial of a specified degree to the data. This method allows for greater flexibility in capturing more complex patterns.

In this guide, we will explore what polynomial curve fitting is, how it can be implemented using Python (primarily with **NumPy**), and provide examples to help you understand the process.

1. What is Polynomial Curve Fitting?

Polynomial curve fitting attempts to fit a polynomial function of a specified degree to a set of data points. The general form of a polynomial is:

$$y = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Where:

- y is the dependent variable (the predicted value).
- x is the independent variable.
- $a_0, a_1, \ldots, a_n$ are the polynomial coefficients.
- n is the degree of the polynomial.

The goal is to find the polynomial coefficients $a_0, a_1, \ldots, a_n$ that best fit the data by minimizing the error between the predicted y-values and the observed data.

2. Why Use Polynomial Fitting?

While linear fitting models the data as a straight line, many real-world datasets have non-linear patterns. Polynomial fitting allows us to capture these patterns by introducing more flexibility through higher-degree polynomials. However, it's essential to use polynomial fitting carefully, as increasing the degree too much can lead to overfitting, where the model captures the noise rather than the underlying trend.

Polynomial fitting can be useful in fields such as:

- Engineering (to model physical processes)

- Economics (for forecasting trends)
- Machine learning (for regression analysis)
- Scientific research (to analyze experimental data)

3. NumPy's `polyfit()` for Polynomial Curve Fitting

In Python, **NumPy** provides the `polyfit()` function to fit polynomials to data. The syntax is the same as for linear fitting, except that you specify the degree of the polynomial you want to fit.

The general syntax is:

```
numpy.polyfit(x, y, deg)
```

Where:

- `x` is the array of independent variable values (input data).
- `y` is the array of dependent variable values (output data).
- `deg` is the degree of the polynomial you want to fit.

The function returns the coefficients of the polynomial in descending powers of `x`.

4. Example: Polynomial Curve Fitting with NumPy

Step 1: Import Required Libraries

First, you need to import the necessary libraries:

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

Step 2: Define the Data

Let's define a set of data points for `x` and `y`. This data exhibits a non-linear relationship:

```
# Independent variable (x)
x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Dependent variable (y) with some non-linear trend
y = np.array([1, 8, 15, 24, 35, 48, 63, 80, 99, 120, 143])
```

Step 3: Perform Polynomial Fitting

Let's fit a polynomial of degree 2 (quadratic) to the data using `numpy.polyfit()`:

```
# Fit a polynomial of degree 2
coefficients = np.polyfit(x, y, 2)

# Print the coefficients of the polynomial
print("Coefficients of the polynomial:", coefficients)
```

```
Coefficients of the polynomial: [0.96503497 4.44055944 1.83916084]
```

The `polyfit()` function will return the coefficients of the polynomial in descending powers of `x`. For a quadratic polynomial (degree 2), it returns three coefficients a_2, a_1, a_0 which represent:

$$y = a_2x^2 + a_1x + a_0$$

Step 4: Generate the Polynomial Function

Once we have the coefficients, we can use `numpy.poly1d()` to create a polynomial function that can be evaluated at any `x` value.

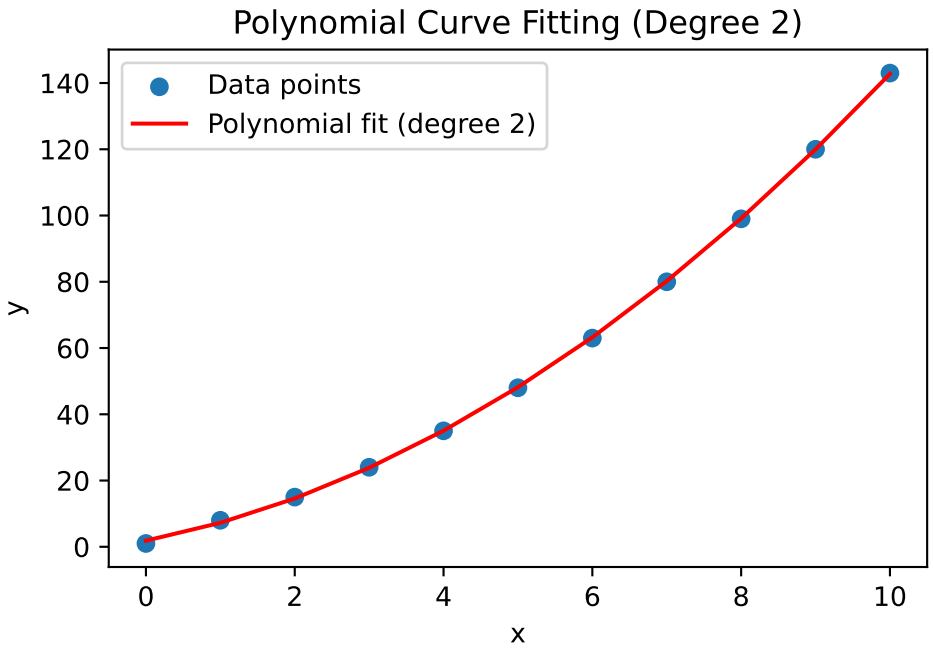
```
# Create a polynomial function from the coefficients
polynomial = np.poly1d(coefficients)

# Calculate the fitted y values
y_fit = polynomial(x)

# Print the fitted values
print("Fitted y values:", y_fit)
```

```
Fitted y values: [  1.83916084   7.24475524  14.58041958  23.84615385  35.04195804
  48.16783217  63.22377622  80.20979021  99.12587413 119.97202797
 142.74825175]
```

Step 5: Plot the Data and the Polynomial Fit

Now that we have both the original data and the fitted polynomial, we can visualize them using `matplotlib`. You will see a scatter plot of the original data points with a smooth curve representing the polynomial fit.

```
# Plot the original data points
plt.scatter(x, y, label='Data points')

# Plot the polynomial curve
plt.plot(x, y_fit, color='red', label=f'Polynomial fit (degree 2)')

# Add labels and legend
plt.xlabel('x')
plt.ylabel('y')
plt.title('Polynomial Curve Fitting (Degree 2)')
plt.legend()
plt.show()
```

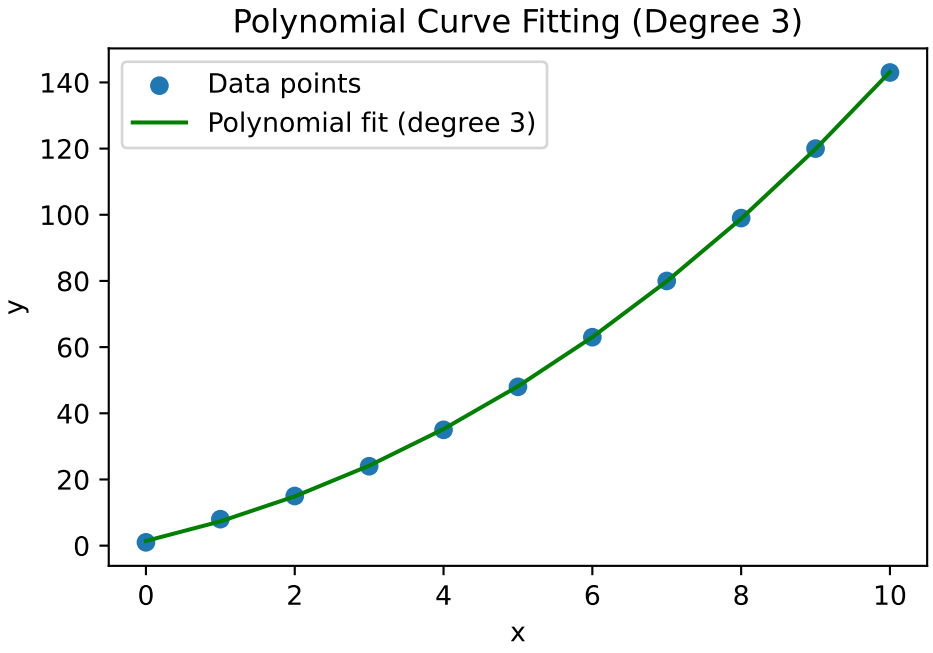


5. Higher-Degree Polynomial Fitting

You can easily experiment with higher-degree polynomials by adjusting the `deg` parameter in `numpy.polyfit()`.

For example, let's fit a 3rd-degree polynomial:

```
# Fit a polynomial of degree 3
coefficients_3 = np.polyfit(x, y, 3)

# Create a polynomial function for degree 3
polynomial_3 = np.poly1d(coefficients_3)

# Calculate the fitted y values
y_fit_3 = polynomial_3(x)

# Plot the original data points and the cubic polynomial fit
plt.scatter(x, y, label='Data points')
plt.plot(x, y_fit_3, color='green', label=f'Polynomial fit (degree 3)')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Polynomial Curve Fitting (Degree 3)')
plt.legend()
plt.show()
```



This will create a cubic curve that may fit the data more closely than the quadratic fit.

6. Evaluating the Polynomial Fit

When fitting a polynomial, it's crucial to evaluate how well the model fits the data. One common measure is the **R-squared** value, which tells you the proportion of the variance in `y` that is explained by the model. A value closer to 1 indicates a better fit.

Here's how to compute the **R-squared** value for polynomial fitting:

```
# Calculate the residual sum of squares (SS_res)
ss_res = np.sum((y - y_fit)**2)

# Calculate the total sum of squares (SS_tot)
ss_tot = np.sum((y - np.mean(y))**2)

# Calculate R-squared
r_squared = 1 - (ss_res / ss_tot)

print(f"R-squared: {r_squared:.2f}")
```

```
R-squared: 1.00
```

7. Overfitting: A Cautionary Note

While higher-degree polynomials offer more flexibility, they also increase the risk of **overfitting**. Overfitting occurs when the model fits the noise in the data rather than the underlying trend. This can lead to poor predictions on new data.

To avoid overfitting, you should:

- Use cross-validation techniques to assess the performance of the model on unseen data.
- Keep the degree of the polynomial as low as possible while still capturing the trend.
- Use regularization techniques (such as Ridge or Lasso regression) if you are working in a machine learning context.

8. Summary

- **Polynomial curve fitting** is a useful technique for modeling complex relationships between variables when linear models are insufficient.
- In Python, **NumPy's `polyfit()`** function makes it easy to fit polynomials to data.
- You can visualize the fitted polynomial using **Matplotlib** and evaluate its performance using metrics like the **R-squared** value.

- While polynomial fitting offers flexibility, it's important to be cautious of overfitting, especially when using high-degree polynomials.

By understanding and using polynomial curve fitting, you can model more intricate data patterns and make better predictions in various fields of research and data analysis.

C) Non-linear Curve Fitting

1. Introduction to Non-linear Curve Fitting

Non-linear curve fitting is a method used to model the relationship between a dependent variable and one or more independent variables when the relationship is not linear. Unlike linear fitting, where the model is represented as a straight line, non-linear fitting involves more complex equations such as exponential, logarithmic, power, and trigonometric functions. These models are essential when the data exhibits non-linear patterns, and a straight-line approximation is insufficient to describe the relationship between the variables.

Non-linear curve fitting finds applications in various fields like physics, biology, economics, and engineering, where the underlying relationships between variables are often non-linear.

2. What is Non-linear Curve Fitting?

Non-linear curve fitting involves fitting a non-linear equation (function) to a set of data points. The equation can take various forms depending on the relationship between the variables. The goal is to determine the best-fitting parameters for the chosen model that minimize the difference between the observed data points and the predicted values.

In non-linear fitting, the equation can be represented as:

$$y = f(x, \mathbf{a}) + \epsilon$$

Where:

- y is the dependent variable (response).
- x is the independent variable (predictor).
- $f(x, \mathbf{a})$ is a non-linear function of x and the parameters $\mathbf{a}$.
- ϵ is the error or noise.

3. Key Differences Between Linear and Non-linear Fitting

- **Linear Fitting**: Fits a straight line (or hyperplane in higher dimensions) to the data. The relationship between variables is linear, and parameters can be estimated using analytical solutions.

$$y = mx + c$$

- **Non-linear Fitting**: Fits a curve defined by a non-linear function to the data. The relationship between variables is non-linear, and parameters are estimated using iterative numerical methods.

  Examples:

  - Exponential function: $y = ae^{bx}$
  - Logistic function: $y = \frac{L}{1+e^{-k(x-x_0)}}$
  - Power-law function: $y = ax^b$

4. Mathematical Background

Non-linear Least Squares

The most common method for non-linear curve fitting is the **non-linear least squares** method. The goal is to find the set of parameters θ that minimize the sum of the squares of the residuals between the observed data y_i and the model predictions $f(x_i, \theta)$:

$$\text{Minimize } S(\theta) = \sum_{i=1}^{n} [y_i - f(x_i, \theta)]^2$$

Since the function f is non-linear in parameters θ, we cannot solve for θ analytically. Instead, we use numerical optimization techniques.

5. Required Libraries For Non-linear Curve Fitting

Python offers several libraries to perform non-linear curve fitting. Two commonly used libraries are:

1. **SciPy**: Provides the `curve_fit()` function in the `scipy.optimize` module for non-linear fitting.
2. **NumPy**: Often used in combination with `SciPy` for handling arrays and mathematical computations.
3. **Matplotlib**: For visualizing the fitted curves.

6 Optimization Methods

- **Gradient Descent**: Iteratively moves towards the minimum of the cost function by taking steps proportional to the negative of the gradient.
- **Levenberg-Marquardt Algorithm**: An efficient algorithm that interpolates between the Gauss-Newton algorithm and the method of gradient descent.
- **Trust Region Methods**: Adjusts the step size based on how well the model fits the data in a localized region.

7. Non-linear Curve Fitting in Python

Python provides several libraries for non-linear curve fitting. The most commonly used is the **`scipy.optimize`** module, specifically the **`curve_fit`** function.

7.1. Importing Required Libraries

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

7.2. The `curve_fit` Function

The `curve_fit` function uses non-linear least squares to fit a function to data.

Syntax:

```
popt, pcov = curve_fit(f, xdata, ydata, p0=None, bounds=(-np.inf, np.inf))
```

- `f`: The model function, $f(x, \theta)$, that takes the independent variable and parameters.

- `xdata`: The independent variable where data is measured.
- `ydata`: The dependent data — nominally $f(xdata, \theta)$.
- `p0`: Initial guess for the parameters (optional).
- `bounds`: Lower and upper bounds on parameters (optional).
- `popt`: Optimal values for the parameters.
- `pcov`: The estimated covariance of `popt`.

8. Examples of Non-linear Curve Fitting

Example 1: Fitting an Exponential Function

The Exponential Model

An exponential growth model:

$$y = ae^{bx} + c$$

Where:

- a, b, and c are parameters to be estimated.

Generating Synthetic Data

```
# Generating synthetic data
np.random.seed(0)
xdata = np.linspace(0, 4, 50)
a_true = 2.5
b_true = 1.3
c_true = 0.5
y = a_true * np.exp(b_true * xdata) + c_true
y_noise = 0.2 * np.random.normal(size=xdata.size)
ydata = y + y_noise
```

Defining the Model Function

```
def exponential_func(x, a, b, c):
    return a * np.exp(b * x) + c
```

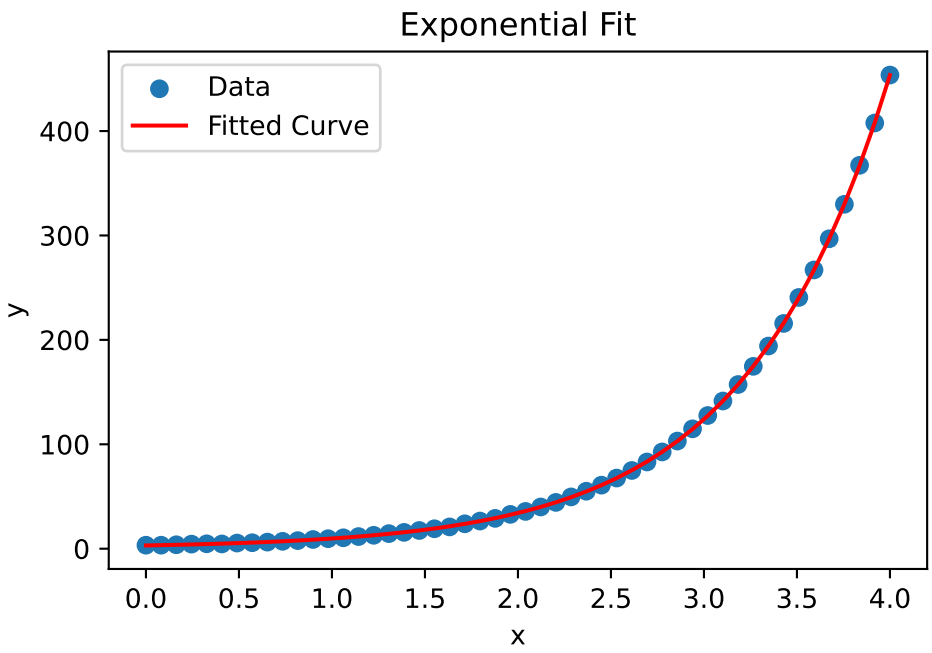
Performing the Curve Fit

```
# Initial guess for parameters [a, b, c]
initial_guess = [1, 1, 1]

# Curve fitting
popt, pcov = curve_fit(exponential_func, xdata, ydata, p0=initial_guess)

# Optimal parameters
a_opt, b_opt, c_opt = popt
print(f"Estimated parameters: a={a_opt}, b={b_opt}, c={c_opt}")
```

```
Estimated parameters: a=2.4892665128679488, b=1.3009604986650651, c=0.6302733688316937
```

Plotting the Results

```
# Generate values using the fitted function
y_fit = exponential_func(xdata, *popt)

plt.scatter(xdata, ydata, label='Data')
plt.plot(xdata, y_fit, 'r-', label='Fitted Curve')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Exponential Fit')
plt.legend()
plt.show()
```



Interpretation

- `popt` contains the estimated parameters.
- The plot shows the data points and the fitted exponential curve.

Example 2: Fitting an Logistic Growth Function

The Logistic Model:

$$y = \frac{L}{1 + e^{-k(x-x_0)}}$$

Where:

- L: The curve's maximum value.
- k: The logistic growth rate.
- x_0: The x-value of the sigmoid's midpoint.

Generating Synthetic Data

```
# Generating synthetic data
xdata = np.linspace(0, 10, 100)
L_true = 1
k_true = 1
x0_true = 5
y = L_true / (1 + np.exp(-k_true * (xdata - x0_true)))
y_noise = 0.05 * np.random.normal(size=xdata.size)
ydata = y + y_noise
```

Defining the Model Function

```
def logistic_func(x, L, k, x0):
    return L / (1 + np.exp(-k * (x - x0)))
```

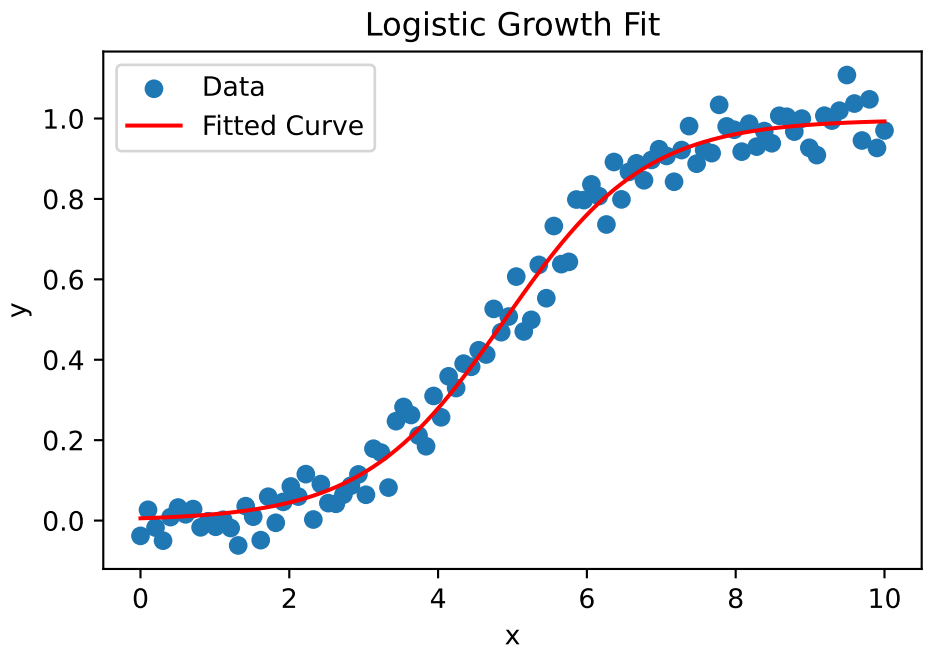
Performing the Curve Fit

```
# Initial guess for parameters [L, k, x0]
initial_guess = [0.5, 0.5, 2]

# Curve fitting
popt, pcov = curve_fit(logistic_func, xdata, ydata, p0=initial_guess)

# Optimal parameters
L_opt, k_opt, x0_opt = popt
print(f"Estimated parameters: L={L_opt}, k={k_opt}, x0={x0_opt}")
```

```
Estimated parameters: L=0.997307712869784, k=1.0561667543216242, x0=4.894440099712688
```

Plotting the Results

```
# Generate values using the fitted function
y_fit = logistic_func(xdata, *popt)

plt.scatter(xdata, ydata, label='Data')
plt.plot(xdata, y_fit, 'r-', label='Fitted Curve')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Logistic Growth Fit')
plt.legend()
plt.show()
```



Interpretation

- The logistic function models S-shaped growth patterns.
- The fitted curve should match the data's S-shaped trend.

Example 3: The Power-Law Model

You can fit any custom function as long as it can be defined in Python. Here we're intereested to fit a ***Power Law Model***

A power-law model:

$$y = ax^b$$

Defining the Model Function

```
def power_law_func(x, a, b):
    return a * x ** b
```

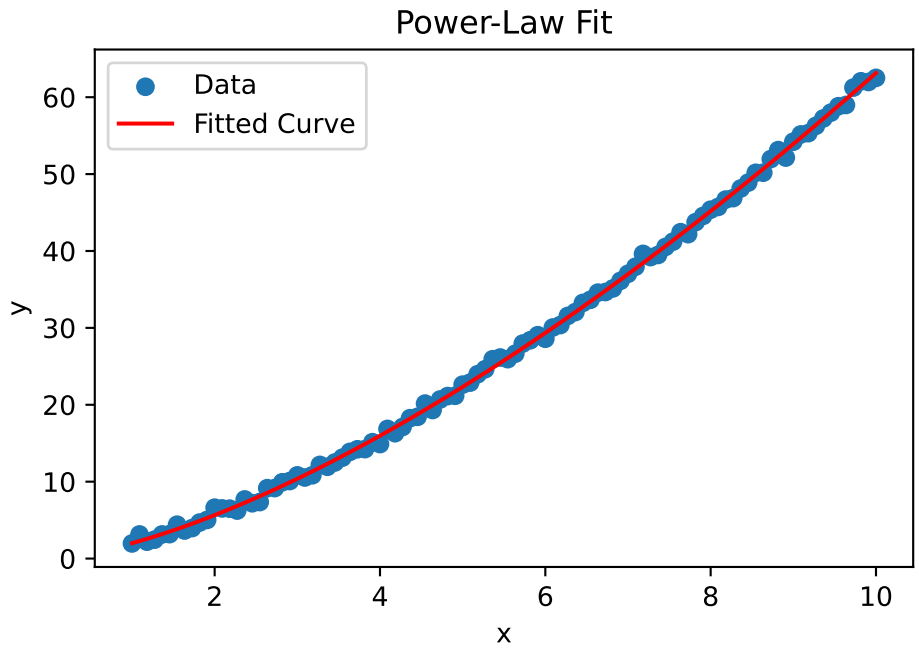
Performing the Curve Fit

```
# Generating synthetic data
xdata = np.linspace(1, 10, 100)
a_true = 2
b_true = 1.5
y = power_law_func(xdata, a_true, b_true)
y_noise = 0.5 * np.random.normal(size=xdata.size)
ydata = y + y_noise

# Initial guess for parameters [a, b]
initial_guess = [1, 1]

# Curve fitting
popt, pcov = curve_fit(power_law_func, xdata, ydata, p0=initial_guess)

# Optimal parameters
a_opt, b_opt = popt
print(f"Estimated parameters: a={a_opt}, b={b_opt}")
```

```
Estimated parameters: a=1.996849080666318, b=1.4999660699935555
```

Plotting the Results

```
# Generate values using the fitted function
y_fit = power_law_func(xdata, *popt)

plt.scatter(xdata, ydata, label='Data')
plt.plot(xdata, y_fit, 'r-', label='Fitted Curve')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Power-Law Fit')
plt.legend()
plt.show()
```



Evaluating the Fit

Residuals Analysis

Residuals are the differences between the observed values and the values predicted by the model:

$$\text{Residual}_i = y_i - f(x_i, \theta)$$

- Plotting residuals can help detect patterns that suggest a poor fit.

```
# Calculate residuals
residuals = ydata - exponential_func(xdata, *popt)

# Plot the residuals
plt.scatter(xdata, residuals)
plt.axhline(0, color='red', linestyle='--')
plt.xlabel("X-data")
plt.ylabel("Residuals")
plt.title("Residual Plot")
plt.show()
```

R-squared (R^2) Value

The coefficient of determination R^2 indicates how well data fit a statistical model.

$$R^2 = 1 - \frac{\sum_i (y_i - f(x_i, \theta))^2}{\sum_i (y_i - \bar{y})^2}$$

```
ss_res = np.sum(residuals**2)
ss_tot = np.sum((ydata - np.mean(ydata))**2)
r_squared = 1 - (ss_res / ss_tot)
print(f"R-squared: {r_squared}")
```

- R^2 ranges from 0 to 1; values closer to 1 indicate a better fit.

9 Confidence Intervals

- The covariance matrix `pcov` returned by `curve_fit` can be used to estimate the standard errors of the parameters.

```
perr = np.sqrt(np.diag(pcov))
print(f"Parameter standard deviations: {perr}")
```

```
Parameter standard deviations: [0.0236384 0.0057124]
```

10. Practical Considerations

10.1. Initial Parameter Estimates

- Good initial guesses (`p0`) can significantly improve the convergence of the fitting algorithm.
- Poor initial estimates might lead to convergence to local minima or failure to converge.

10.2. Convergence Issues

- **Non-convergence**: The optimization algorithm may fail to converge to a solution.
- **Local Minima**: The algorithm may find a local minimum rather than the global minimum.

Solutions:

- Experiment with different initial parameter estimates.
- Use bounds to restrict the parameter space.
- Scale the data if necessary.

10.3. Data Quality

- **Outliers**: Extreme values can skew the fit.
- **Noise**: High levels of noise can make it difficult to find the underlying trend.
- **Sample Size**: More data points can lead to a more reliable fit.

10.4. Overfitting

- Using a model that's too complex can lead to overfitting, where the model fits the noise rather than the underlying trend.
- Use the simplest model that adequately fits the data.

11. Conclusion

Non-linear curve fitting is a powerful tool for modeling complex relationships in data. By using Python's `scipy.optimize.curve_fit` function, you can fit custom non-linear functions to data with relative ease. It's essential to understand the underlying model, choose appropriate initial parameters, and evaluate the fit thoroughly to ensure reliable results.