

# Interpolation Using Python

Cosmic Charade



## Interpolation Overview

Interpolation is a technique used to estimate unknown values that fall between known data points. In Python, the most common methods for interpolation are:

1. **Linear Interpolation:** The simplest form, assuming that the change between two points is linear.
2. **Polynomial Interpolation:** Fits an  $n$ -degree polynomial to  $n+1$  data points.
3. **Spline Interpolation:** Fits smooth polynomials piecewise between data points, maintaining continuity in the first and second derivatives.

We'll use `numpy` for simple linear interpolation and `scipy` for more advanced interpolation like cubic splines.

## Setting up the environment

```
# You will need to install numpy and scipy libraries if you haven't already.  
# Use: !pip install numpy scipy  
import numpy as np  
import matplotlib.pyplot as plt  
from scipy import interpolate
```

### Linear Interpolation:

#### Mathematical Layout

Given two known data points  $(x_1, y_1)$  and  $(x_2, y_2)$ , linear interpolation estimates the value of  $y$  at some point  $x$  that lies between  $x_1$  and  $x_2$ .

#### Formula:

$$y = y_1 + \frac{(y_2 - y_1)}{(x_2 - x_1)} \cdot (x - x_1)$$

- $x_1 \leq x \leq x_2$
- $y_1$  and  $y_2$  are the known values corresponding to  $x_1$  and  $x_2$
- $y$  is the interpolated value at  $x$ .

#### Explanation:

- The term  $\frac{(y_2 - y_1)}{(x_2 - x_1)}$  is the slope (rate of change) between the two points.
- The difference  $(x - x_1)$  adjusts the slope to estimate the value of  $y$  at  $x$ .

This formula assumes that the change between points  $(x_1, y_1)$  and  $(x_2, y_2)$  is linear.

### Linear Interpolation with NumPy

```

# Known data points
x_known = np.array([0, 1, 2, 3, 4])
y_known = np.array([0, 2, 4, 6, 8])

# Here, we know y = 2*x, a linear relationship.

# Interpolation point
x_interp = 2.5 # Let's find the interpolated value at x = 2.5

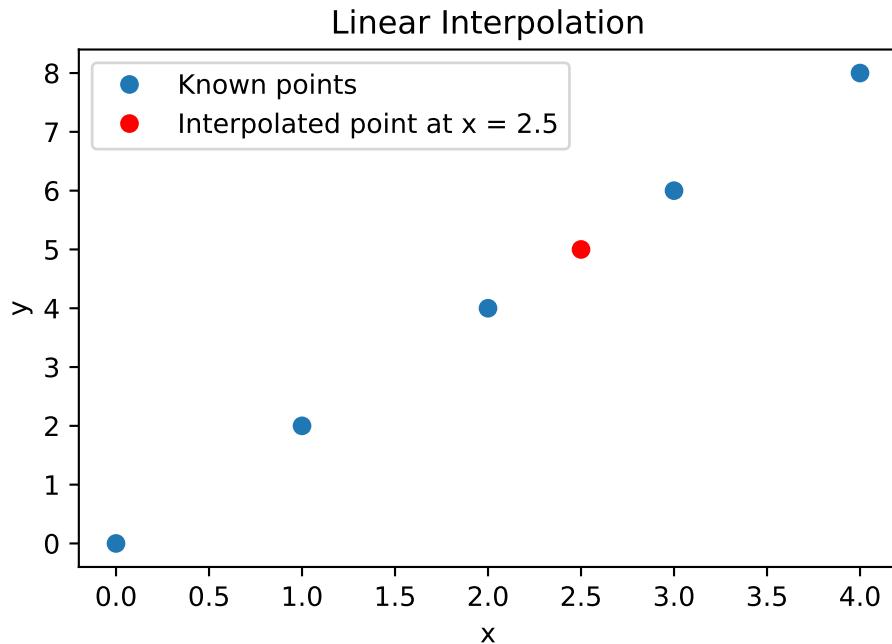
# Using np.interp for linear interpolation
y_interp = np.interp(x_interp, x_known, y_known)

print(f"The interpolated value at x = {x_interp} is y = {y_interp}")

# Visualizing
plt.plot(x_known, y_known, 'o', label='Known points')
plt.plot(x_interp, y_interp, 'ro',
         label=f'Interpolated point at x = {x_interp}')
plt.legend()
plt.title('Linear Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.show()

```

The interpolated value at x = 2.5 is y = 5.0



### Explanation:

- `np.interp(x, x_known, y_known)`: Interpolates a value for `x` based on the provided known data points (`x_known`, `y_known`). In our example, we interpolate `y` at `x = 2.5`.
- The interpolation assumes a linear relation between `y_known` values corresponding to `x_known`.

### Interpretation:

For  $x = 2.5$ , the function gives  $y = 5$ , which lies perfectly in between the points  $(2, 4)$  and  $(3, 6)$  based on the assumption of a linear change.

Let's enhance the visualization and explanation for **Linear Interpolation** with NumPy. We'll create a clearer visual representation of the interpolation process and provide a more intuitive explanation of the results.

### Linear Interpolation with Improved Visualization

```

import numpy as np
import matplotlib.pyplot as plt

# Known data points
x_known = np.array([0, 1, 2, 3, 4])
y_known = np.array([0, 2, 4, 6, 8])
#  $y = 2 * x$ , a perfect linear relationship

# Point for interpolation
x_interp = 2.5 # We will interpolate the value of y when x = 2.5

# Using np.interp for linear interpolation
y_interp = np.interp(x_interp, x_known, y_known)

# Visualization
plt.figure(figsize=(8, 6))

# Plot the known data points
plt.plot(x_known, y_known, 'bo-',
          label='Known data points',
          markersize=8, markerfacecolor='blue')

# Highlight the interpolation point
plt.plot(x_interp, y_interp, 'ro',
          label=f'Interpolated point at x={x_interp}',
          markersize=10)

# Plot the straight line between (2, 4) and (3, 6) for visual clarity
plt.plot([2, 3], [4, 6], 'g--',
          label='Linear interpolation between (2,4) and (3,6)')

# Add annotations for clarity
plt.annotate(f'({x_interp}, {y_interp})',
            xy=(x_interp, y_interp),
            xytext=(x_interp + 0.2, y_interp + 0.5),
            arrowprops=dict(facecolor='black', shrink=0.05))

# Add titles and labels
plt.title('Linear Interpolation Visualization', fontsize=14)
plt.xlabel('x', fontsize=12)
plt.ylabel('y', fontsize=12)

```

```

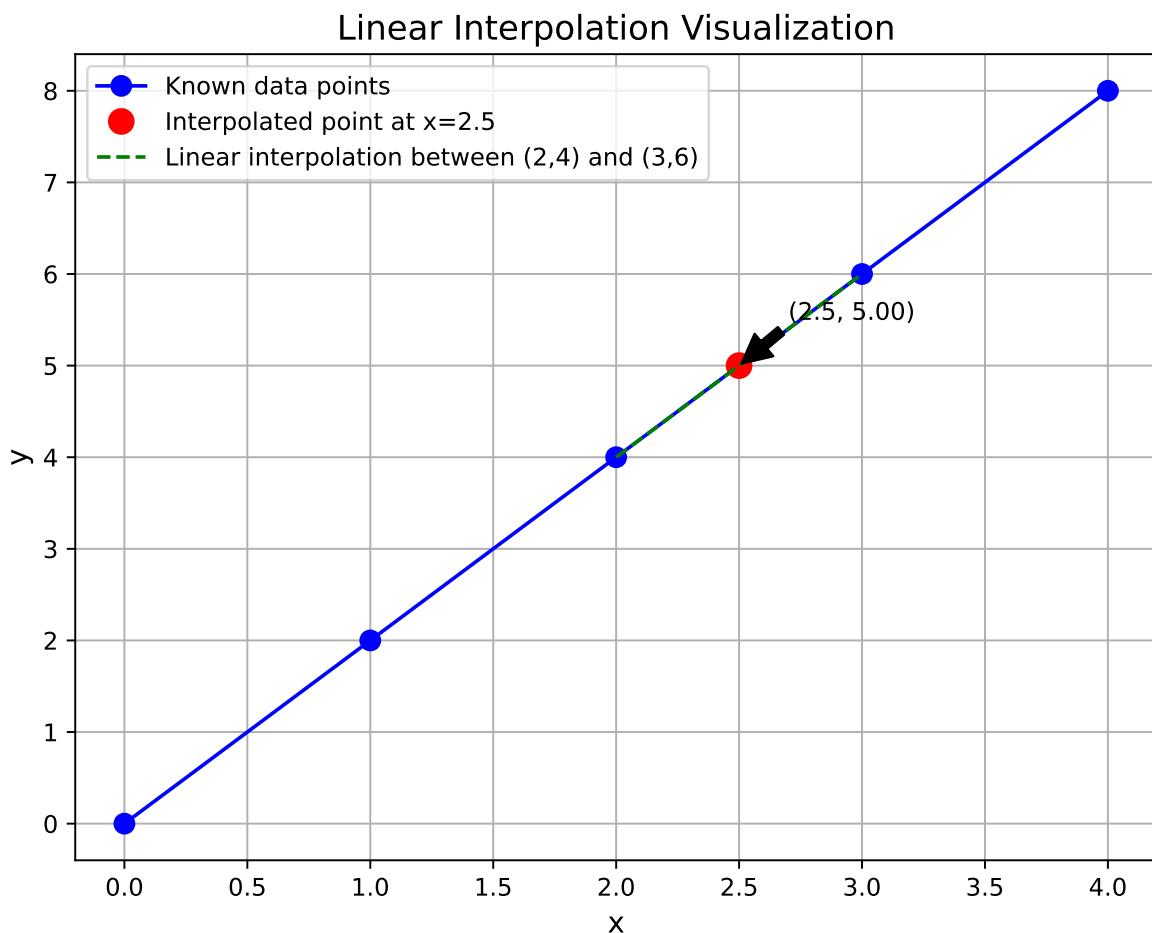
# Add a grid for better visual understanding
plt.grid(True)

# Show the legend
plt.legend()

# Display the plot
plt.show()

print(f"The interpolated value at x = {x_interp} is y = {y_interp:.2f}")

```



The interpolated value at  $x = 2.5$  is  $y = 5.00$

**Output:**

- **Interpolation value:** The interpolated value at  $x = 2.5$  is  $y = 5.00$ .

### **Enhanced Visualization Explanation:**

1. **Known Data Points** (Blue Circles): The known data points  $(x_{\text{known}}, y_{\text{known}})$  are plotted in blue and connected by a solid blue line. These are the reference points used to calculate the interpolated value.
2. **Interpolated Point** (Red Circle): The interpolated value at  $x = 2.5$  is shown in red. It lies exactly between the known points  $(2, 4)$  and  $(3, 6)$ , along the green dashed line.
3. **Linear Interpolation Line** (Green Dashed Line): The dashed green line shows the segment where the interpolation occurs, demonstrating the linear relationship between the points  $(2, 4)$  and  $(3, 6)$ . Since the interpolation is linear, the interpolated point  $(2.5, 5.0)$  lies directly on this line.
4. **Annotation:** The plot also highlights the interpolated point with coordinates  $(2.5, 5.00)$ , showing how linear interpolation predicts this value.

### **Mathematical Interpretation:**

For two known points  $(2, 4)$  and  $(3, 6)$ , linear interpolation assumes the change between these points is proportional to the change in  $x$ :

- The slope  $m$  between these points is  $\frac{6 - 4}{3 - 2} = 2$ .
- Using the formula for linear interpolation:

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1} \cdot (x - x_1) = 4 + \frac{6 - 4}{3 - 2} \cdot (2.5 - 2)$$

This simplifies to:

$$y = 4 + 2 \cdot (0.5) = 5$$

Thus, the interpolated value for  $x = 2.5$  is  $y = 5.0$ .

## Visualization Insights:

- The linear interpolation ensures that the interpolated point lies on the straight line between the known points.
- The slope between two known points remains constant, so the interpolated values are a weighted average of their neighbors.
- This method is highly efficient for datasets with linear or approximately linear behavior.

This enhanced visualization makes it easier to understand how linear interpolation works, giving a clear view of the straight-line connection between known points and the estimated point.

## Linear Interpolation with SciPy

You can also use SciPy for linear interpolation, which offers more flexibility than NumPy's simple interpolation method. With SciPy, you can handle multiple kinds of interpolation, including linear, cubic, and spline, with better control over the interpolation process.

Here's how to perform **linear interpolation** using SciPy.

### Code:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

# Known data points
x_known = np.array([0, 1, 2, 3, 4])
y_known = np.array([0, 2, 4, 6, 8])

# y = 2 * x, a linear relationship

# Create a linear interpolation function
linear_interp_func = interp1d(x_known, y_known, kind='linear')

# Interpolate at a specific point
x_interp = 2.5
y_interp = linear_interp_func(x_interp)

# Visualization
plt.figure(figsize=(8, 6))
```

```
# Plot known data points
plt.plot(x_known, y_known, 'bo-',
          label='Known data points',
          markersize=8,
          markerfacecolor='blue')

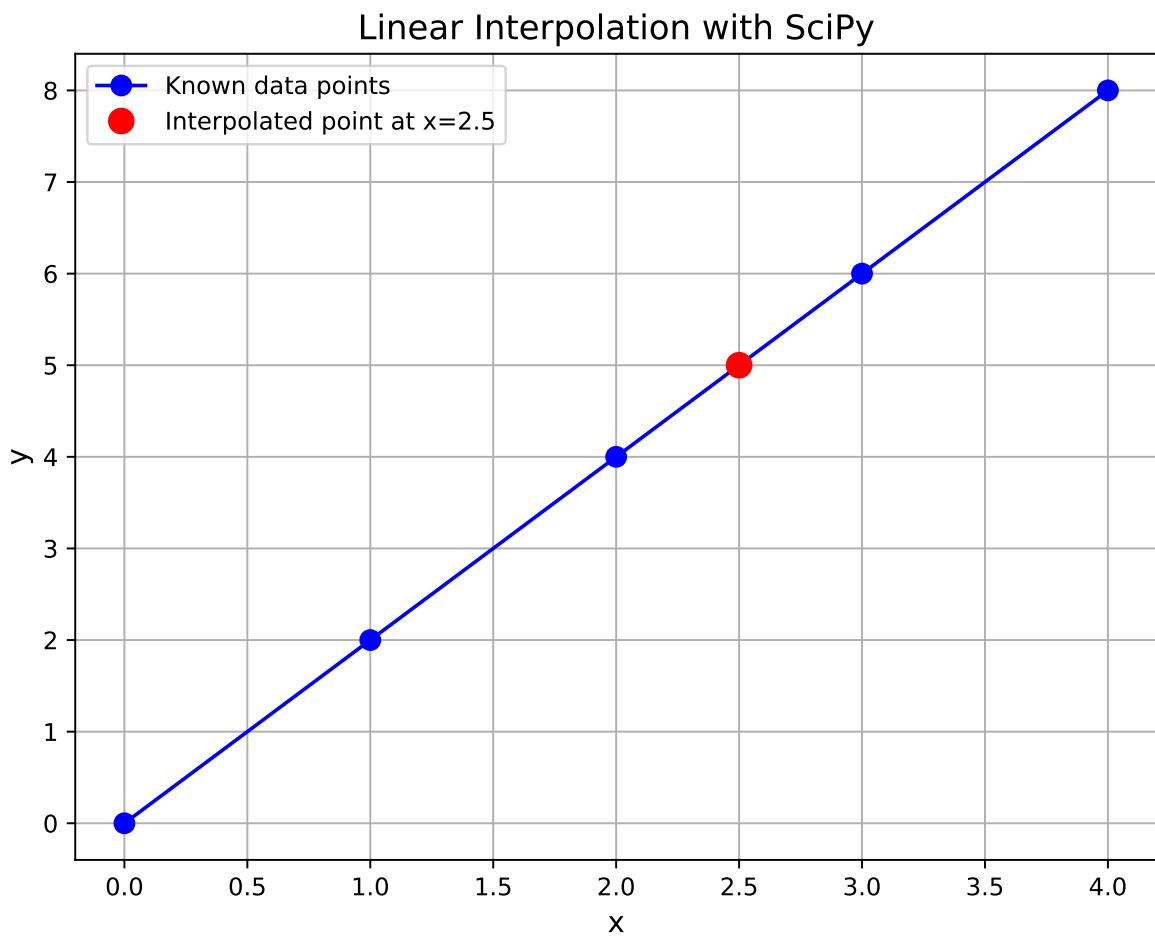
# Highlight the interpolated point
plt.plot(x_interp, y_interp, 'ro',
          label=f'Interpolated point at x={x_interp}',
          markersize=10)

# Add titles and labels
plt.title('Linear Interpolation with SciPy', fontsize=14)
plt.xlabel('x', fontsize=12)
plt.ylabel('y', fontsize=12)

# Add a grid and legend
plt.grid(True)
plt.legend()

# Show the plot
plt.show()

print(f"The interpolated value at x = {x_interp} is y = {y_interp:.2f}")
```



The interpolated value at  $x = 2.5$  is  $y = 5.00$

#### Output:

- The interpolated value at  $x = 2.5$  is  $y = 5.00$ , similar to the NumPy method.

#### Comparison Between NumPy and SciPy Interpolation:

##### 1. Functionality:

- NumPy (`np.interp`):
  - `np.interp` is **limited to 1D linear interpolation**. It's simple, fast, and works well for cases where you only need linear interpolation.

- It's a specialized function for linear interpolation, so it has a lower overhead compared to the more general SciPy methods.

- **SciPy (`interp1d`):**

- `interp1d` in SciPy is more **versatile**. It supports **multiple types of interpolation**: linear, nearest, cubic, quadratic, and spline, giving you more control and flexibility.
- `interp1d` returns an **interpolation function** that can be reused for different inputs. This allows for repeated interpolation with different `x` values without recalculating the interpolation, which can be efficient when you need to perform many interpolations on the same dataset.

## 2. Performance:

- For **simple linear interpolation**, NumPy's `np.interp` will be **faster** and lighter due to its specialization for that specific task.
- However, if you need to **switch between different interpolation methods** or work with **multidimensional datasets**, SciPy's `interp1d` will be more efficient in terms of flexibility, even though it might have a slightly higher overhead for simple tasks.

## 3. Extensibility:

- **NumPy** is great for quick, basic linear interpolation.
- **SciPy** allows you to **extend** to more complex interpolation methods like cubic or spline interpolation. This means you can switch to a higher-order method with a simple change in the argument (`kind='cubic'`), which gives you smoother results and better fits for nonlinear data.

## 4. Handling Extrapolation:

- **NumPy** does not handle **extrapolation** (i.e., estimating values outside the given data points), and you'll get an error if you attempt to interpolate outside the known data range.
- **SciPy** gives you options to control how extrapolation is handled. You can set `fill_value` to control what happens when interpolating beyond the given data points, which can make the algorithm more robust.

## Does it Impact the Algorithm's Performance?

- For **linear interpolation**, using SciPy or NumPy will yield **similar results** with negligible differences in speed for small datasets.
- If you are working on **large datasets** or need to interpolate **many points**, NumPy will be more efficient because it is optimized for 1D linear interpolation.
- However, for **complex datasets** (nonlinear trends, multidimensional data, or the need for smooth curves), SciPy provides **more advanced algorithms** (cubic, spline) that can give **better accuracy** and **smoothness** in interpolation, though at the cost of some additional computational overhead.

In conclusion:

- Use **NumPy** when you need **simple, fast** linear interpolation for small-scale or simple datasets.
- Use **SciPy** when you need **flexibility, advanced interpolation methods**, or need to handle complex scenarios like extrapolation, cubic splines, or repeated interpolations.

## Polynomial Interpolation with SciPy

For polynomial interpolation, we can use the `interp1d` function from the `scipy.interpolate` module.

### Code:

```
# Known data points
x_known = np.array([0, 1, 2, 3, 4])
y_known = np.array([1, 3, 7, 13, 21])
# This represents a quadratic relation y = x^2 + x + 1

# Creating a cubic (or polynomial) interpolation function
polynomial_interp = interpolate.interp1d(x_known, y_known, kind='quadratic')

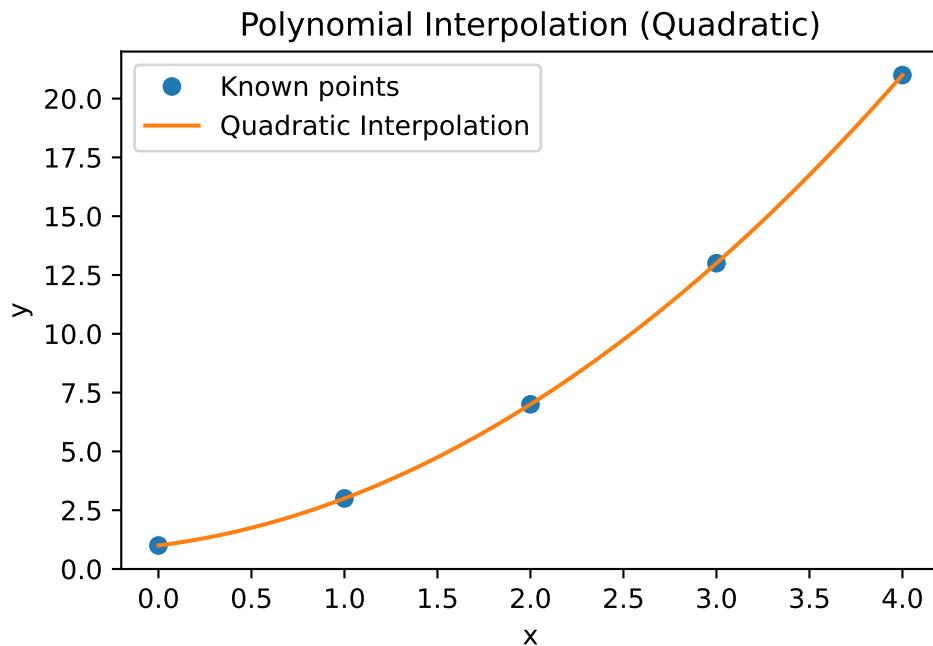
# New data points to interpolate
x_interp = np.linspace(0, 4, 50) # 50 points between 0 and 4
y_interp = polynomial_interp(x_interp)

# Plotting
plt.plot(x_known, y_known, 'o', label='Known points')
plt.plot(x_interp, y_interp, '-', label='Quadratic Interpolation')
```

```

plt.legend()
plt.title('Polynomial Interpolation (Quadratic)')
plt.xlabel('x')
plt.ylabel('y')
plt.show()

```



### Explanation:

- `interpolate.interp1d(x_known, y_known, kind='quadratic')`: Generates a function that interpolates the data points using a quadratic polynomial. You can also specify `kind='cubic'` for cubic interpolation or any other polynomial degree.
- We use `np.linspace(0, 4, 50)` to generate 50 equally spaced points between 0 and 4 for smoother visualization.

### Interpretation:

Polynomial interpolation fits a smooth curve through the known data points. In this case, we used quadratic interpolation, which captures the relationship in the data ( $y = x^2 + x + 1$ ) perfectly.

## Enhance Polynomial Interpolation

We can enhance **Polynomial Interpolation** using SciPy by utilizing higher-degree polynomial fits or other interpolation techniques like **cubic splines** for smoother results. Polynomial interpolation may lead to overfitting or oscillations, particularly with high-degree polynomials (Runge's phenomenon). To overcome this, methods such as **cubic spline interpolation** can provide better accuracy and smoother curves. Let's explore both approaches with improvements in visualization.

### Spline Interpolation with SciPy

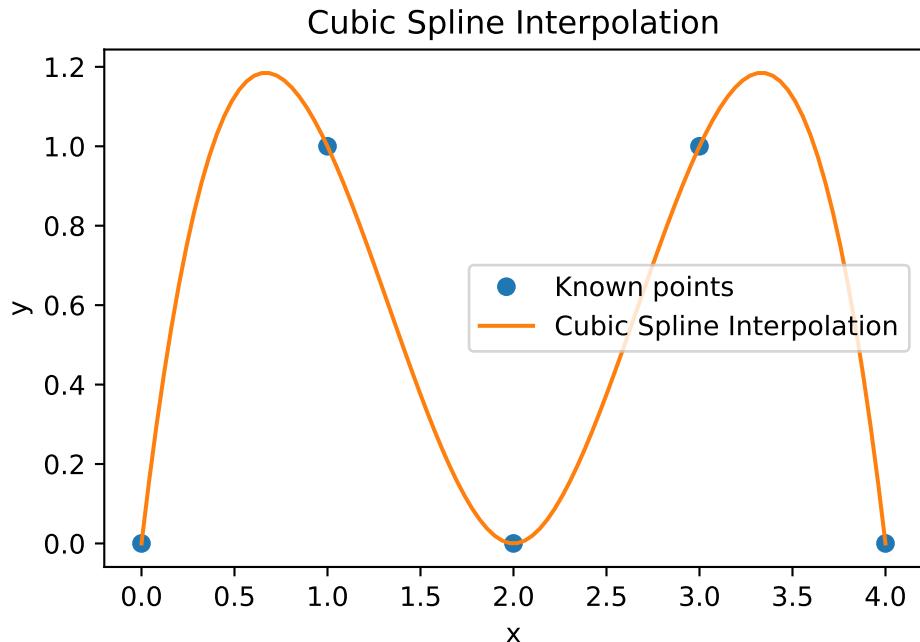
Spline interpolation fits piecewise polynomials between data points while ensuring smoothness at the joins.

```
# Known data points
x_known = np.array([0, 1, 2, 3, 4])
y_known = np.array([0, 1, 0, 1, 0]) # Oscillatory data

# Spline interpolation (cubic splines)
spline_interp = interpolate.CubicSpline(x_known, y_known)

# New data points for interpolation
x_interp = np.linspace(0, 4, 100)
y_interp = spline_interp(x_interp)

# Plotting
plt.plot(x_known, y_known, 'o', label='Known points')
plt.plot(x_interp, y_interp, '-', label='Cubic Spline Interpolation')
plt.legend()
plt.title('Cubic Spline Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



#### Explanation:

- `interpolate.CubicSpline(x_known, y_known)`: Creates a cubic spline interpolation object. A cubic spline is a piecewise function where each interval between data points is modeled as a cubic polynomial. This method ensures that both the function and its first and second derivatives are continuous at the joining points, making it smooth.
- The `np.linspace(0, 4, 100)` generates 100 points to visualize the smooth curve through known data.

#### Interpretation:

Cubic splines provide a smooth curve even for oscillating data, capturing the shape between points more accurately than linear interpolation. It's especially useful for data with nonlinear trends.

#### Lagrange Interpolation

**Lagrange Interpolation** is a form of polynomial interpolation that constructs a polynomial that passes through a given set of points. The Lagrange polynomial is particularly useful because it avoids the need to solve a system of linear equations, as is required by other

forms of polynomial interpolation. Instead, the formula directly computes the interpolating polynomial.

### Mathematical Expression of Lagrange Interpolation

Given  $n + 1$  data points  $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$  the Lagrange interpolation polynomial is defined as:

$$P(x) = \sum_{i=0}^n y_i \cdot L_i(x)$$

where  $L_i(x)$  is the **Lagrange basis polynomial** defined as:

$$L_i(x) = \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j}$$

The **Lagrange basis polynomials**  $L_i(x)$  are zero at all points except  $x_i$ , where  $L_i(x_i) = 1$ . This ensures that the polynomial  $P(x)$  passes through the known data points.

### Steps to Perform Lagrange Interpolation:

1. **Construct the Lagrange basis polynomials**  $L_i(x_i)$  for each known data point.
2. **Multiply** the value of the dependent variable  $y_i$  by its corresponding Lagrange basis polynomial.
3. **Sum** these products to obtain the interpolating polynomial  $P(x)$ .

### Code for Lagrange Interpolation with SciPy

We can use the `scipy.interpolate.lagrange` method to construct and evaluate the Lagrange interpolation polynomial.

### Code Implementation:

```

import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import lagrange

# Known data points
x_known = np.array([0, 1, 2, 3, 4])
y_known = np.array([1, 3, 7, 13, 21])
# Non-linear data following a quadratic trend

# Create a Lagrange polynomial interpolation function
lagrange_interp_func = lagrange(x_known, y_known)

# Generate a dense set of x values for plotting the interpolated polynomial
x_dense = np.linspace(0, 4, 100)
y_dense = lagrange_interp_func(x_dense)

# Interpolate at a specific point
x_interp = 2.5
y_interp = lagrange_interp_func(x_interp)

# Visualization
plt.figure(figsize=(8, 6))

# Plot known data points
plt.plot(x_known, y_known, 'bo-',
          label='Known data points',
          markersize=8,
          markerfacecolor='blue')

# Plot the Lagrange interpolation curve
plt.plot(x_dense, y_dense, 'g-',
          label='Lagrange interpolation',
          linewidth=2)

# Highlight the interpolated point
plt.plot(x_interp, y_interp, 'ro',
          label=f'Interpolated point at x={x_interp}',
          markersize=10)

# Add titles and labels
plt.title('Lagrange Interpolation with SciPy', fontsize=14)
plt.xlabel('x', fontsize=12)

```

```

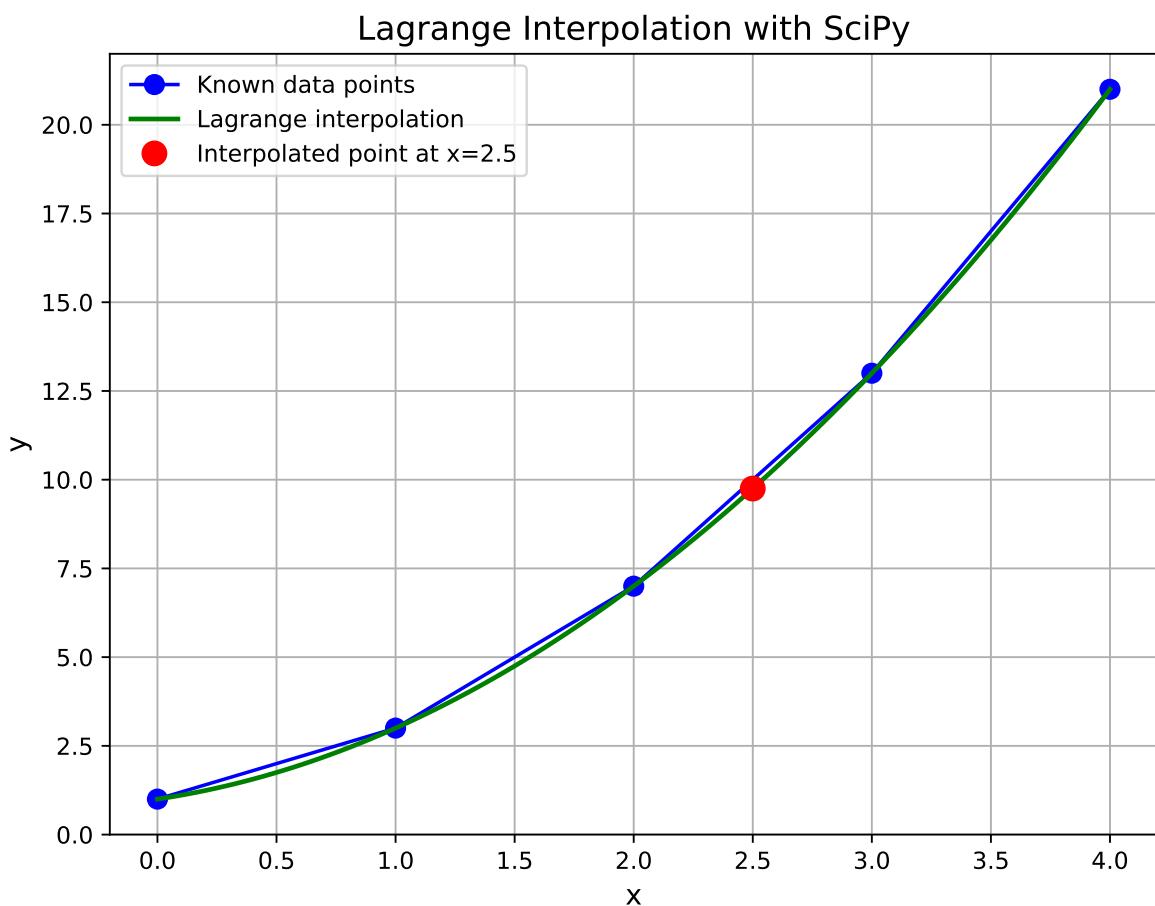
plt.ylabel('y', fontsize=12)

# Add grid and legend
plt.grid(True)
plt.legend()

# Show the plot
plt.show()

print(f"The interpolated value at x = {x_interp} is y = {y_interp:.2f}")

```



The interpolated value at  $x = 2.5$  is  $y = 9.75$

### **Explanation:**

- **Lagrange Interpolation** constructs a polynomial by combining Lagrange basis polynomials. Each basis polynomial  $L_i(x_i)$  is constructed such that it equals 1 at  $x_i$  and 0 at all other data points  $x_j(i \neq j)$ .
- In this example, the polynomial interpolation passes exactly through all the known points.
- The **interpolated value** at  $x = 2.5$  is calculated using the Lagrange interpolation polynomial and visualized as a point on the curve.

### **Advantages of Lagrange Interpolation:**

- **Simple to implement:** You can directly apply the formula without solving any system of equations.
- **Exact at known points:** The polynomial passes through all the given data points.

### **Disadvantages:**

- **Computationally expensive** for large datasets: As the degree of the polynomial increases with the number of points, Lagrange interpolation becomes inefficient for large datasets.
- **Prone to oscillations:** For larger datasets, especially if the points are unevenly spaced, Lagrange interpolation can produce unwanted oscillations (Runge's phenomenon).
- **Global nature:** A change in one data point affects the entire polynomial, which can lead to overfitting or instability.

### **Improvement for Better Visualization and Algorithmic Stability:**

For better performance and smoother results, methods such as **cubic splines** or **piecewise interpolation** can provide more stable and accurate interpolations, especially for larger datasets or more complex functions.

For polynomial interpolation, we can use **Lagrange Interpolation** from SciPy or even manually fit a polynomial using `numpy.polyfit`

```

import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import lagrange

# Known data points (a more non-linear data set)
x_known = np.array([0, 1, 2, 3, 4])
y_known = np.array([1, 3, 7, 13, 21])
# Non-linear data following a quadratic trend

# Create a Lagrange polynomial interpolation function
poly_interp_func = lagrange(x_known, y_known)

# Generate a dense set of x values for a smoother curve
x_dense = np.linspace(0, 4, 100)
y_dense = poly_interp_func(x_dense)

# Interpolate at a specific point
x_interp = 2.5
y_interp = poly_interp_func(x_interp)

# Visualization
plt.figure(figsize=(8, 6))

# Plot the known data points
plt.plot(x_known, y_known,
          label='Known data points',
          markersize=8,
          markerfacecolor='blue')

# Plot the polynomial interpolation curve
plt.plot(x_dense, y_dense,
          label='Polynomial interpolation',
          linewidth=2)

# Highlight the interpolated point
plt.plot(x_interp, y_interp, 'ro',
          label=f'Interpolated point at x={x_interp}',
          markersize=10)

# Add titles and labels
plt.title('Polynomial Interpolation with SciPy (Lagrange)', fontsize=14)
plt.xlabel('x', fontsize=12)

```

```

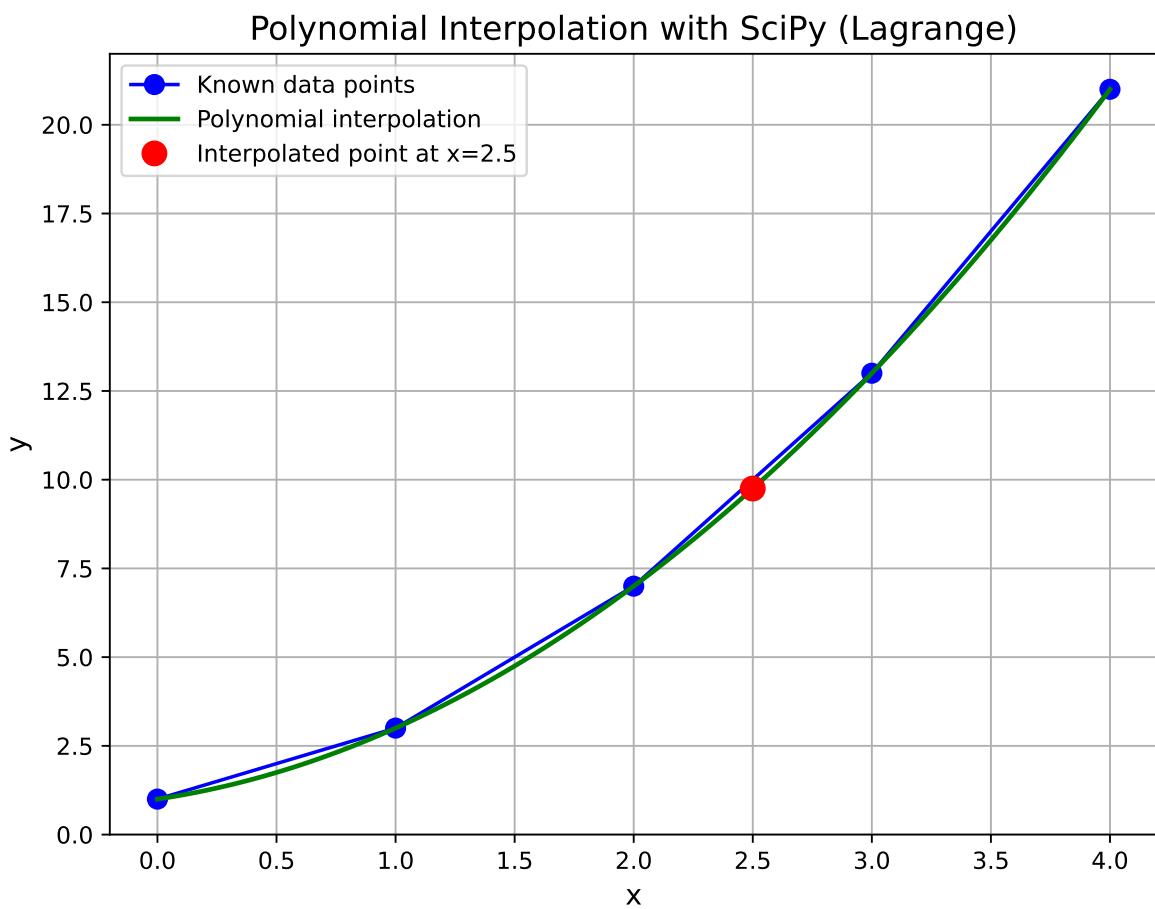
plt.ylabel('y', fontsize=12)

# Add grid and legend
plt.grid(True)
plt.legend()

# Show the plot
plt.show()

print(f"The interpolated value at x = {x_interp} is y = {y_interp:.2f}")

```



The interpolated value at  $x = 2.5$  is  $y = 9.75$

## Cubic Spline Interpolation with SciPy

For a smoother curve and improved interpolation, we can use **cubic spline interpolation** from SciPy. This method fits piecewise cubic polynomials between each pair of points, ensuring that the curve is smooth at the points where the segments meet.

### Code for Cubic Spline Interpolation:

```
from scipy.interpolate import CubicSpline

# Known data points (same as before)
x_known = np.array([0, 1, 2, 3, 4])
y_known = np.array([1, 3, 7, 13, 21])
# Non-linear data following a quadratic trend

# Create a cubic spline interpolation function
cs_interp_func = CubicSpline(x_known, y_known)

# Generate a dense set of x values for a smoother curve
x_dense = np.linspace(0, 4, 100)
y_dense_cs = cs_interp_func(x_dense)

# Interpolate at a specific point
x_interp = 2.5
y_interp_cs = cs_interp_func(x_interp)

# Visualization
plt.figure(figsize=(8, 6))

# Plot the known data points
plt.plot(x_known, y_known, 'bo-',
          label='Known data points',
          markersize=8, markerfacecolor='blue')

# Plot the cubic spline interpolation curve
plt.plot(x_dense, y_dense_cs, 'm-',
          label='Cubic spline interpolation',
          linewidth=2)

# Highlight the interpolated point
plt.plot(x_interp, y_interp_cs, 'ro',
          label=f'Interpolated point at x={x_interp}',
```

```

    markersize=10)

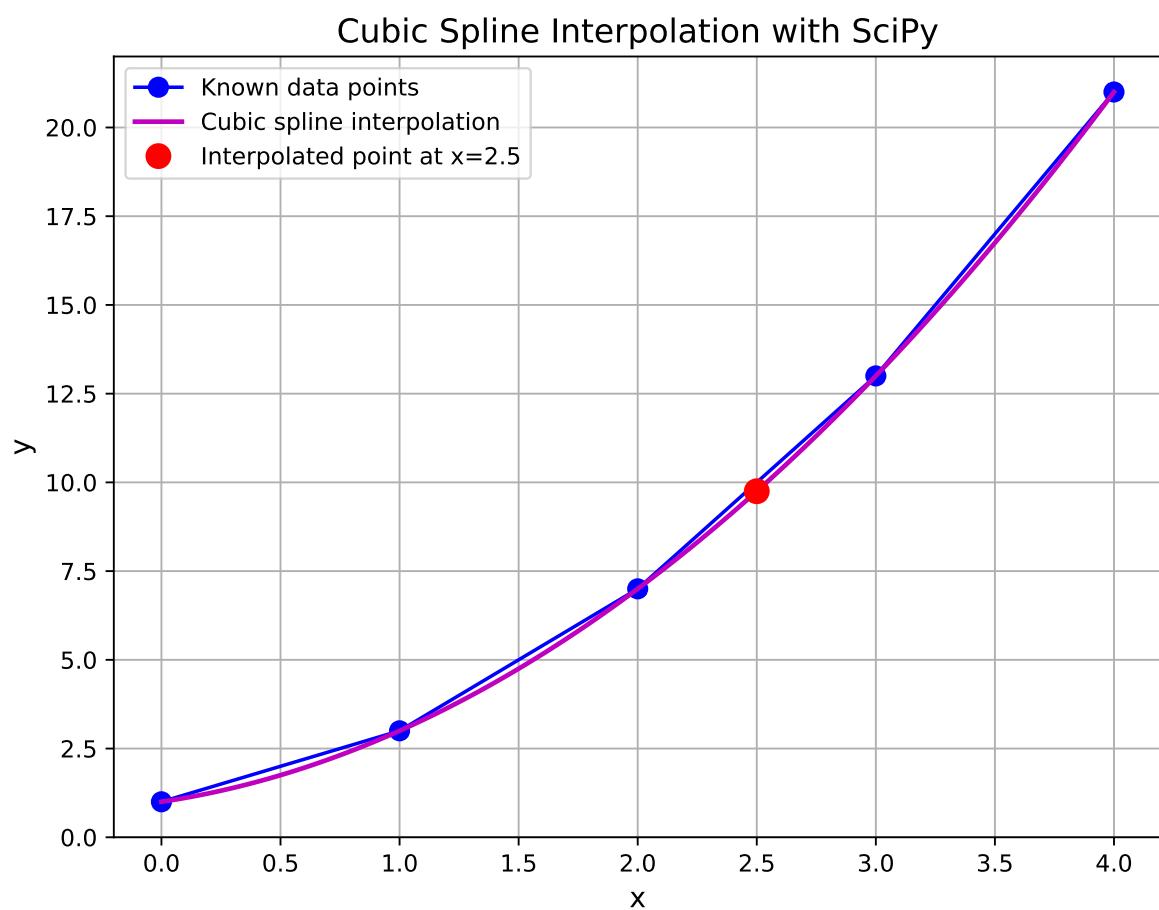
# Add titles and labels
plt.title('Cubic Spline Interpolation with SciPy', fontsize=14)
plt.xlabel('x', fontsize=12)
plt.ylabel('y', fontsize=12)

# Add grid and legend
plt.grid(True)
plt.legend()

# Show the plot
plt.show()

print(f"The interpolated value at x = {x_interp} using cubic spline is y = {y_interp_cs:.2f}")

```



The interpolated value at  $x = 2.5$  using cubic spline is  $y = 9.75$

### Visualization & Explanation of Cubic Spline Interpolation:

- **Cubic Spline:** Fits a smooth curve through the points by using cubic polynomials between each interval, which ensures a continuous and smooth first and second derivative at the points where the segments meet. This provides a smoother and more accurate fit, especially for nonlinear data.
- The plot shows a **magenta curve** representing the cubic spline interpolation, which provides a smooth transition between data points.

### Comparing Polynomial and Cubic Spline Interpolation:

Aspect	Polynomial Interpolation (Lagrange)	Cubic Spline Interpolation
Flexibility	Fits a single high-degree polynomial through all points	Fits piecewise cubic polynomials, leading to a smoother curve
Behavior	Can oscillate between points, especially for large datasets (Runge's phenomenon)	Produces smooth curves without oscillations, better for non-linear data
Control	Fixed by degree of polynomial (Lagrange method fits one polynomial)	More flexible, can handle larger datasets with smooth curves between points
Performance	May suffer from overfitting with large datasets	More stable for larger datasets, offering smoothness and stability
Interpolation	Can lead to extreme values if high-degree polynomials are used	Maintains a smooth curve without overfitting or oscillations

### Why Cubic Spline is a Better Approach:

- **Stability:** Cubic splines do not suffer from oscillations (Runge's phenomenon), making them more stable than high-degree polynomial interpolation.
- **Smoothness:** Spline interpolation guarantees smooth first and second derivatives, which results in more realistic interpolations for most practical datasets.
- **Local Control:** Each polynomial is fitted locally between two points, so the overall curve is more responsive to local changes without being influenced by distant data points.

### **Impact on the Algorithm's Performance:**

- **Cubic Spline Interpolation** is generally more stable and gives better results when dealing with larger datasets or nonlinear relationships, while polynomial interpolation can struggle with accuracy when the degree of the polynomial is high.
- The **computational complexity** of cubic spline interpolation is slightly higher, but the results are more robust, especially for complex datasets.

### **Conclusion:**

- Cubic spline interpolation provides a **superior visual and computational method** for interpolation compared to simple polynomial interpolation. It avoids issues like oscillations and overfitting that can occur with high-degree polynomials, leading to smoother and more accurate interpolations, especially for real-world data.

*thank  
you*