How to Plot a Smooth Curve in Matplotlib

Authored by

stats writer

December 22, 2025

The Matplotlib library is the cornerstone of Python's data visualization ecosystem, offering robust tools for generating static, animated, and interactive plots. While its basic functions are excellent for representing discrete data points or standard linear trends, achieving a truly smooth curve--one that suggests an underlying continuous process--often requires mathematical intervention beyond simple line drawing.

A standard line plot connects data points using straight segments, which can result in a jagged or angular appearance, especially when the input data points are sparse or irregularly spaced. To transition from these sharp angles to a fluid, continuous curve, we must employ techniques that mathematically estimate the values between the measured points. This process, known as interpolation, is crucial for improving the visual aesthetics and interpretability of data that is inherently continuous in nature, such as time series or physical measurements.

When generating smooth curves from discrete datasets, the ideal approach involves leveraging the advanced numerical capabilities provided by the SciPy library, which works seamlessly alongside Matplotlib and NumPy. SciPy offers sophisticated algorithms for spline interpolation, which are far superior to linear connections for creating aesthetically pleasing and mathematically sound curves.

scipy.interpolate.make_interp_spline(): This function constructs a piecewise polynomial interpolating spline, passing directly through the given data points.
scipy.interpolate.BSpline(): While often used internally, understanding B-splines is key to controlling the smoothness and degree of the generated curve.

This comprehensive guide details the practical application of these SciPy functions to transform raw, jagged line charts into perfectly smooth visualizations, enhancing both the visual quality and the underlying mathematical representation.

The Necessity of Interpolation for Curve Smoothing

Plotting raw data points connected by straight lines can sometimes be misleading, suggesting abrupt shifts where the underlying phenomenon is likely gradual. Interpolation solves this by fitting a mathematical curve--specifically a spline--through the existing data points. This curve then allows us to calculate an infinite number of intermediate points, effectively resampling the dataset at a much higher resolution.

A major advantage of using splines, particularly those generated by SciPy's interpolation module, is that they ensure the curve is continuous not only in position but also often in its first and second derivatives (velocity and acceleration, metaphorically speaking). This continuity guarantees that the resulting line does not contain sudden, unnatural kinks or sharp turns, providing a realistic representation of the continuous data being modeled.

Furthermore, proper smoothing via interpolation is essential when preparing data for publications or presentations where visual clarity is paramount. A smooth curve can reveal trends and overall shapes that might be obscured by the noise or sparseness of the original data points, facilitating easier interpretation by the audience.

Setting Up the Initial Matplotlib Plot

Before applying advanced smoothing techniques, it is instructive to observe the output of a standard Matplotlib line plot when using discrete, non-linear data. This baseline helps illustrate the problem that spline interpolation is designed to solve. We begin by importing the necessary libraries, NumPy for array handling and Matplotlib for plotting, and defining a simple dataset that exhibits a non-linear relationship.

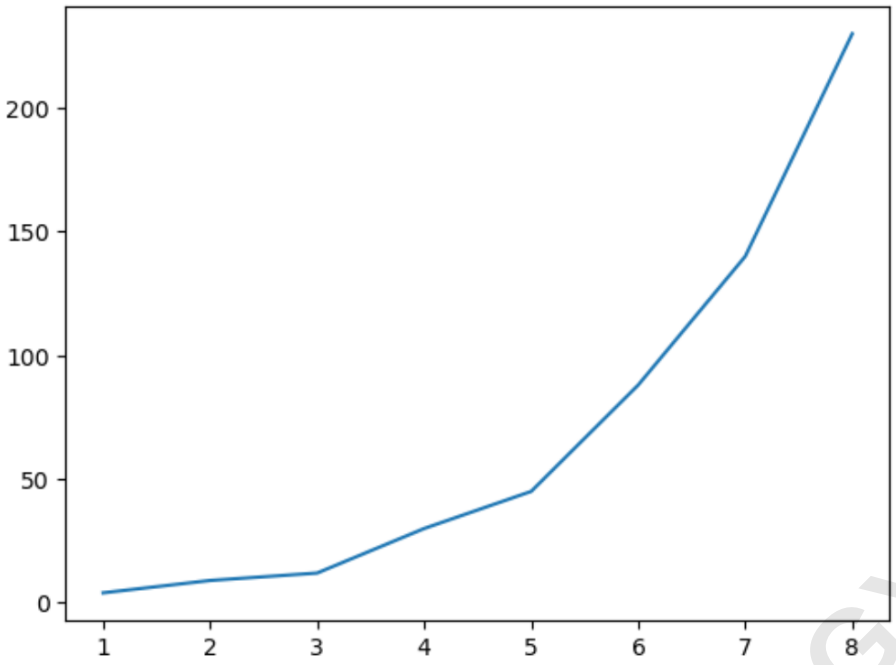
The resulting visualization, created using the standard `plt.plot(x, y)` command, simply draws a straight segment between each adjacent pair of (x, y) coordinates. Although mathematically correct for connecting the measured points, the visual effect is often disjointed and fails to convey the impression of a continuous curve that typically underlies such exponential or growth-related data patterns.

The following code block demonstrates the setup of this initial, unsmoothed chart. Notice how the default behavior of `plt.plot()` emphasizes the exact location of the original data points rather than the overall shape of the trend.

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

#create data
x = np.array()
y = np.array()

#create line chart
plt.plot(x,y)
plt.show()
```



Upon reviewing the generated image, it is evident that the line chart isn't completely smooth since the underlying data points are connected linearly, creating sharp angles at each coordinate. This lack of fluidity motivates the need for spline interpolation.

Implementing SciPy's make_interp_spline for Smoothing

To achieve the desired smoothness, we must first generate a new, much denser set of x-coordinates (often called `xnew`) and then use a spline function to calculate the corresponding y-values (`y_smooth`) that lie on the continuous curve passing through the original data. The `scipy.interpolate.make_interp_spline()` function is the primary tool for this task, constructing a smooth polynomial representation of the data.

The critical first step is defining `xnew`. We use NumPy's `linspace` function to create an array of values that span the exact range of our original x-data, but with a significantly larger number of points (e.g., 200 or more). This high resolution is what allows Matplotlib to draw a visually continuous and smooth line when plotting the interpolated values.

Once `xnew` is defined, we call `make_interp_spline(x, y, k=3)`, where `k` specifies the degree of the polynomial spline. A value of `k=3` indicates a cubic spline, which is the most common choice as it provides excellent smoothness while generally avoiding excessive oscillation. The output, `spl`, is a callable function that represents the interpolated curve, which we then evaluate using `spl(xnew)` to get our final smooth y-values.

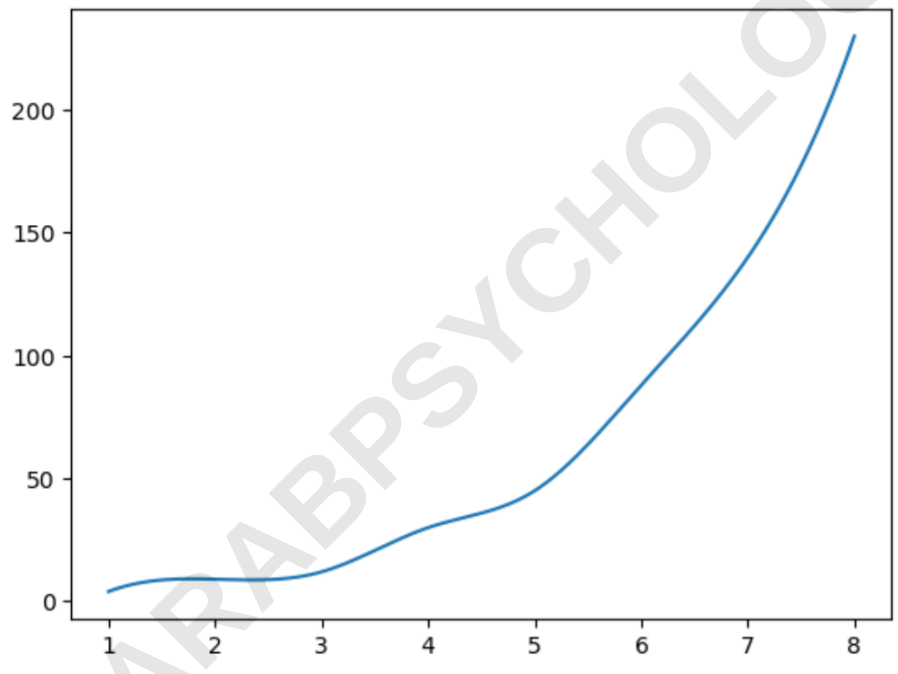
from scipy.interpolate import make_interp_spline, BSpline

```
#create data
x = np.array()
y = np.array()

#define x as 200 equally spaced values between the min and max of original x
xnew = np.linspace(x.min(), x.max(), 200)

#define spline (k=3 means cubic spline)
spl = make_interp_spline(x, y, k=3)
y_smooth = spl(xnew)

#create smooth line chart
plt.plot(xnew, y_smooth)
plt.show()
```



The resulting plot, visualized using the newly generated `xnew` and `y_smooth` arrays, displays a dramatically smoother curve. This transformation demonstrates the power of interpolation in enhancing the visual representation of underlying data trends, making the visualization far more impactful and easier to interpret.

The Significance of the Spline Degree (k)

A critical parameter in spline interpolation is the degree, represented by the argument `k` in

`make_interp_spline()`. This value determines the order of the polynomial used to fit the curve segments between data points. Common choices include `k=1` (linear interpolation, resulting in straight lines), `k=2` (quadratic), and `k=3` (cubic, which is the standard for maximum smoothness).

The choice of `k` directly influences the trade-off between smoothness and fidelity to the local curvature of the data. Lower degrees (like `k=1` or `k=2`) result in less smooth transitions but are computationally simpler. Conversely, higher degrees introduce greater flexibility and smoothness but carry the risk of "Runge's phenomenon," where the curve exhibits unwanted oscillations or "wiggliness" between data points, especially near the boundaries.

Note that the higher the degree you use for the **k** argument, the more pronounced the oscillations or "wiggly" nature of the curve will be. While a higher degree ensures that the curve passes through all points with high order continuity, excessive degrees can lead to mathematical artifacts that do not truly reflect the underlying physical process. For most data visualization tasks, a cubic spline (`k=3`) offers the best balance of visual appeal and mathematical stability.

Demonstrating Excessive Degree (k=7)

To illustrate the effect of using an overly high degree, we can re-run the interpolation using `k=7`. This seventh-degree polynomial forces the curve to conform very tightly to the local neighborhood of each data point, leading to exaggerated curvature and potential overshoot, which deviates significantly from a visually simple, smooth trend.

Although the mathematical constraint requires the curve to pass perfectly through the original eight points, the path taken between these points becomes highly convoluted. This "wiggliness" is a clear example of overfitting, where the interpolating function is too sensitive to the specific measurements and potentially captures noise rather than the signal.

The following example demonstrates this phenomenon. Observe how the curve bends sharply around the data points compared to the gentle transitions provided by the cubic spline (`k=3`).

from scipy.interpolate import make_interp_spline, BSpline

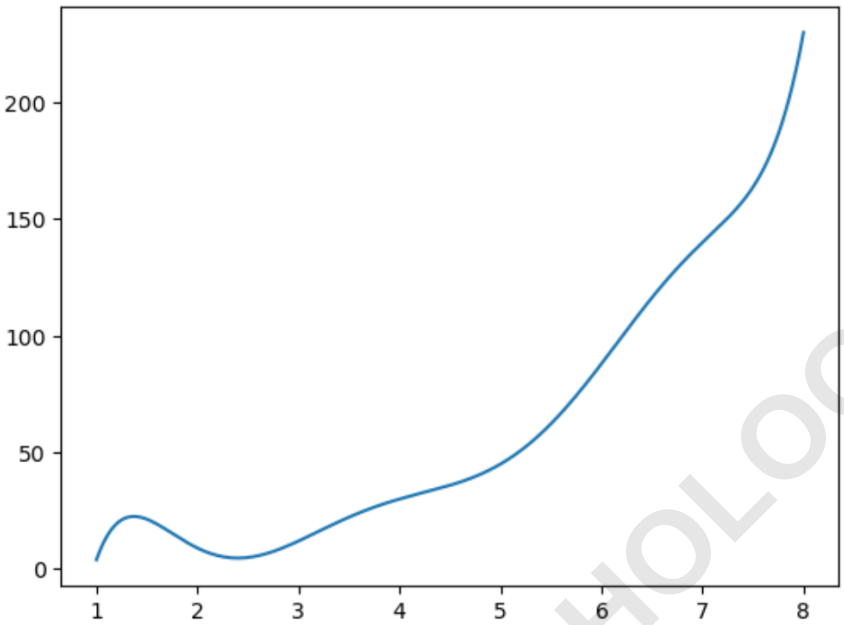
#create data
x = np.array()
y = np.array()

#define x as 200 equally spaced values between the min and max of original x
xnew = np.linspace(x.min(), x.max(), 200)

#define spline with degree k=7
spl = make_interp_spline(x, y, k=7)

```
y_smooth = spl(xnew)

#create smooth line chart
plt.plot(xnew, y_smooth)
plt.show()
```



Depending on how curved you want the line to be, you can modify the value for **k**. However, for most robust graphical representations, a degree between 2 and 5 is recommended. The crucial insight here is that smoothness is not simply achieved by increasing the polynomial degree; it requires a mindful selection of the degree that best fits the natural characteristics of the data.

Alternative Interpolation Methods in SciPy

While B-spline interpolation using `make_interp_spline` is highly effective, SciPy provides other methods for smoothing curves, each suitable for different data types and requirements. One alternative is the use of Radial Basis Functions (RBF) interpolation, particularly useful when dealing with multi-dimensional data or irregularly spaced points, although it can be more computationally intensive.

Another popular method is using the `interp1d` function, which offers simpler methods like linear and cubic interpolation but does not provide the sophisticated smoothness controls inherent to splines. If performance is a major concern and the data points are not too sparse, `interp1d` might be sufficient, though it generally won't match the quality of a well-tuned B-spline fit.

It is important to differentiate between interpolation (which forces the curve through all given points) and curve fitting (which attempts to find a function that approximates the trend but does not necessarily pass through every single point, thus providing noise reduction). If the goal is pure visual smoothing where fidelity to every measured point is required, interpolation methods like splines are the correct choice. If the data is noisy and a generalized trend is desired, regression techniques would be more appropriate.

Best Practices for Smooth Curve Visualization

To ensure your smooth curve plots are accurate and professional, several best practices should be observed. First, always ensure that the resolution of your interpolated data (`xnew`) is sufficiently high. A minimum of 100 points, or ideally 200-500 points, between the minimum and maximum x-values will prevent the smooth curve from appearing segmented or pixelated when rendered by Matplotlib.

Second, consider plotting the original discrete data points alongside the smooth curve. This technique provides context, allowing the viewer to understand which points are actual measurements and which sections are mathematically interpolated. Using different colors or markers (e.g., small circles for the raw data and a solid line for the spline) clearly differentiates the original measurement from the generated trend.

Finally, always justify your choice of the spline degree **k**. While `k=3` is the standard, if you have theoretical reasons to believe the underlying curve has a simpler form, a lower degree might be more honest. Conversely, avoid using excessively high degrees (k > 5) unless your data guarantees rapid oscillation, as this often leads to visually misleading results, as demonstrated in our `k=7` example. The goal is to convey the trend smoothly without introducing mathematical artifacts.

Summary of Interpolation Requirements

Generating a smooth curve in Matplotlib is not a built-in feature of the plotting function itself; rather, it is achieved through pre-processing the data using advanced SciPy tools. This methodical approach ensures that the resulting visualization is not only visually pleasing but also mathematically sound, based on rigorous spline interpolation.

The key steps involve defining a dense array of new x-coordinates using NumPy's `linspace`, constructing the interpolating spline using `make_interp_spline`, and then plotting the newly calculated interpolated curve. By mastering the degree parameter **k**, you gain precise control over the smoothness and fidelity of the resulting visualization, transforming sparse data into continuous, professional-grade plots.

How to Show Gridlines on Matplotlib Plots
How to Remove Ticks from Matplotlib Plots
How to Create Matplotlib Plots with Log Scales