Spline Interpolation

Chorobek Sheranov, Olle Rehnfeldt, Jakob Hutter
Github Repository

November 26, 2023

Abstract

While Linear Splines are trivial and prove to be inaccurate this paper focuses on Cubic Splines. Cubic Splines represent a periodic interpolation method generating third-order polynomial functions between adjacent points for interpolation purposes. These polynomials adhere to specific conditions: 1. Ensuring a continuous graph along the x-axis from the initial to the final point, 2. Maintaining consistent first derivatives for the starting and ending splines at each data point, and 3. They are establishing that the second derivative for the starting and ending splines at every data point is zero. 4. Different Versions of Endpoint Conditions. Additionally, our investigation demonstrates that this interpolation technique 1. exhibits stability by avoiding oscillation in contrast to single polynomial strategies, 2. proves effective in handling volatile functions when applied judiciously, and 3. is significantly more time efficient than single polynomial strategies (when implemented right)

1 Basic Idea

In contrast to other single polynomial interpolation methods, such as the Lagrange Polynomial, which seeks a singular function facilitating interpolation across all points, spline interpolation adopts a periodic approach. In this method, each data point is connected to its nearest neighbor through a distinct function, commonly referred to as a "spline." Unlike single polynomial interpolation techniques, where the polynomial order is contingent on the number of points, spline interpolation allows for the selection of the order of the "functions between points" or splines.

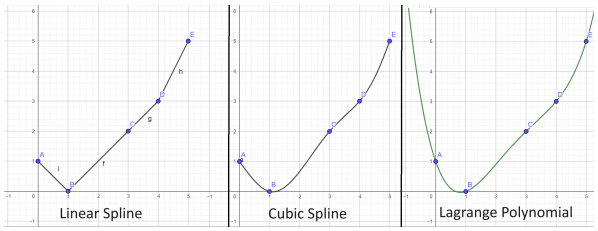


Figure 1: Comparison Interpolation Techniques

The accompanying figure illustrates three distinct cases: the left graph presents a linear spline example, where the splines form simple linear vectors connecting adjacent points, making the function calculation straightforward. Adjacent to it is the scenario of cubic spline interpolation, where each spline possesses an order of 3. The computation of these splines involves satisfying specific conditions, detailed under the heading "Cubic Conditions." Higher-order splines are not typically calculated due to the diminishing returns on effectiveness and the associated decrease in calculation speed. The final graph in the figure illustrates the visualization of the Lagrange polynomial for the given set of points.

In the following sections, this paper will delve into the intricacies of cubic splines, exploring their construction and analyzing the advantages they offer compared to other interpolation methods. As a result, when referring to "splines" or "spline interpolation" in the subsequent discussion, the focus will be on the specific model of cubic splines.

2 Cubic Conditions

There are many ways to find a cubic spline, these conditions and the definition of the single splines g_i are chosen such that they represent the mathematics used by *scipy.interpolate.CubicSpline*.
In the pursuit of establishing continuous and smooth functions between sets of given points, denoted as $P = \{(x_0, y_0), (x_1, y_1), (x_2, y_2) \ldots, (x_n, y_n)\}$ the chosen mathematical framework involves the definition of cubic splines. These splines, expressed as third-order polynomials:

$$g_i = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i, i \in [1, n] \subset \mathbb{N}, x \in [x_{i-1}, x_i]$$

The selection of third-order polynomial functions facilitates differentiability in the first and second order, a crucial characteristic for reliable interpolation results. The following constraints in cubic spline interpolation ensure a seamless and visually coherent representation of the data, enhancing reliability by preventing abrupt jumps or discontinuities. Without these constraints, the interpolation could exhibit unnatural fluctuations in slope and curvature, compromising the fidelity of the interpolated function to the original data. The constraints are articulated as follows:

1. Interpolation Continuity

Ensuring the creation of an uninterrupted graph formed by connected splines, this constraint dictates that at each connection point x_i, the spline function g_i matches the given data point (x_i, y_i). Moreover, it mandates the equality of the function values at the end of each spline segment, ensuring a seamless transition:

$$g_i(x_{i+1}) = y_{i+1} = g_{i+1}(x_{i+1}), i \in [1, n-1] \subset \mathbb{N}$$

2. Tangent Continuity

To eliminate hard edges and promote a visually coherent, single graph formed by connected splines, the curvature continuity constraint stipulates that the first derivatives of adjacent spline segments are equal at their connection points:

$$g_i'(x_i) = g_{i+1}'(x_i), i \in [1, n-1] \subset \mathbb{N}$$

3. Curvature Continuity

Ensuring a seamless junction of starting and ending splines at each point, the second derivative continuity constraint emphasizes that the rate of change in slope remains consistent. This further refines the smoothness of the connected spline:

$$g_i''(x_i) = g_{i+1}''(x_i), i \in [1, n-1] \subset \mathbb{N}$$

4. Endpoint Smoothness

There are 4 common ways to define the endpoints in such a way that there are enough conditions to calculate the coefficients a_i, b_i, c_i, d_i in all instances. These are:

i. The clamped condition which assumes the first derivatives at the endpoints to be 0.

ii. Natural condition, setting the second derivative at the endpoint functions to 0.

iii. The periodic conditionsetsng the value of the functions equal to each other at the endpoints. $g_0(x) = g_n(x)$

iv. The "not-a-knot" condition. This is the default setting 'not-a-knot' used by *scipy.interpolate.CubicSpline*.

$$g_1'''(x_0) = 0, g_{n-1}'''(x_n) = 0$$

This is the method we will focus on in this paper due to it being the basic implementation in SciPy.

These four constraints are sufficient that for $n - 1$ of spline functions, $(n - 1) * 4$ conditions can be found, which makes this a solvable system. The following paragraph will focus on one way of computationally implementing it, as used by *scipy.interpolate.CubicSpline*.

3 Method for deriving a_i, b_i, c_i, d_i

Let's introduce the notation h_i and η_i $h_i = x_{i+1} - x - i$, $\eta_i = y_{i+1} - y_i$. Then

$$d_i = y_i \text{ Since: } g_i(x_i) = a_i(x_i - x_i)^3 + b_i(x_i - x_i)^2 + c_i(x_i - x_i) + d_i = d_i$$

To be able to calculate the remaining constants a_i, b_i, c_i we will need to solve a linear system using the constraints above. Thus, a_i and c_i we will be defined in terms of b_i.
To make this possible we will define the Tangent Continuity as such:

$$g_i'(x_i) = 3a_i(h_i)^2 + 2b_i(h_i) + c_i$$

The curvature continuity as such:

$$g_i''(x_i) = 6a_i(h_i) + 2b_i$$

this allows to rewrite b_{i+1} as:

$$6a_i(h_i) + 2b_i = b_{i+1}$$

$$\text{defived from: } g_{i+1}''(x_i) = 6a_i(x_{i+1} - x_{i+1}) + 2b_{i+1} = 2b_{1+i}$$

The interpolation continuity as such:

$$g_i(x_{i+1}) = a_i h_i^3 + b_i h_i^2 + c_i h_i + y_i = y_{i+1} \Rightarrow a_i h_i^3 + b_i h_i^2 + c_i h_i = \eta_i$$

$$\text{defived from: } d_i = y_i \text{ and } \eta_i = y_{i+1} - y_i$$

The tangent continuity can then be rewritten as:

$$g_i'(x_{i+1}) = 3a_i \cdot h_i^2 + 2b_i h_i + c_i = c_{i+1}$$

$$\text{defived from: } g_{i+1}(x_{i+1}) = 3a_{i+1}(x_{i+1} - x_{i+1})^2 + 2b_i(x_{i+1} - x_{i+1}) + c_i = c_{i+1}$$

We now have 4n - 2 constraints.
Then using the curvature continuity a_i can be rewritten in terms of b_i as such:

$$6a_i(h_i) + 2b_i = b_{i+1} \Rightarrow a_i = \frac{b_{i+1} - b_i}{3h_i}$$

Using this definition of a_i in b_i it is possible to rewrite c_i strictly in terms of b_i, as such:

$$a_i h_i^3 + b_i h_i^2 + c_i h_i = \eta_i \Rightarrow c_i = \frac{\eta_i - a_i h_i^3 - b_i h_i^2}{h_i} \Rightarrow c_i = \frac{\eta_i - \frac{b_{i+1} - b_i}{3h_i} h_i^3 - b_i h_i^2}{h_i} \Rightarrow c_i = \frac{\eta_i}{h_i} - \frac{1}{3} h_i (b_{i+1} - 2b_i)$$

Using the definition of the tangent continuity, an equation with the terms a_i b_i c_i can now, after some algebraic modification (Appendix Item 1), be rewritten in terms of only b_i as such:

$$3a_i h_i^2 + 2b_i h_i + c_i = c_{i+1} \Rightarrow \frac{1}{3} h_i b_i + \frac{2}{3}(h_i + h_{i+1}) b_{i+1} + \frac{1}{3} h_{i+1} b_{i+2} = \frac{\eta_{i+1}}{h_{i+1}} - \frac{\eta_i}{h_i}$$

After defining each a_i and c_i in terms of b_i we can now write all equations giving us b_i into the matrix form $Ax = b$, where A is now a tridiagonal matrix and $m :=$ missing value:

$$Ax = b$$

$$\begin{pmatrix} \cdots & \cdots & \cdots & \cdots & m & \cdots & \cdots & \cdots & \cdots \\ \frac{1}{3}h_0 & \frac{2}{3}(h_0 + h_1) & \frac{1}{3}h_1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{3}h_1 & \frac{2}{3}(h_1 + h_2) & \frac{1}{3}h_1 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \frac{1}{3}h_{n-2} & \frac{2}{3}(h_{n-3} + h_{n-2}) & \frac{1}{3}h_{n-2} & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & \frac{1}{3}h_{n-2} & \frac{2}{3}(h_{n-2} + h_{n-1}) & \frac{1}{3}h_{n-1} \\ \cdots & \cdots & \cdots & \cdots & m & \cdots & \cdots & \cdots & \cdots \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-2} \\ b_{n-1} \\ b_n \end{pmatrix}$$

$$= \begin{pmatrix} m \\ \frac{\eta_1}{h_1} - \frac{\eta_0}{h_0} \\ \frac{\eta_2}{h_2} - \frac{\eta_1}{h_1} \\ \vdots \\ \frac{\eta_{n-2}}{h_{n-2}} - \frac{\eta_{n-3}}{h_{n-3}} \\ \frac{\eta_{n-1}}{h_{n-1}} - \frac{\eta_{n-2}}{h_{n-2}} \\ m \end{pmatrix}$$

Solving for b_i and plugging the values back into our previously defined constraints will give us the values for all a_i, b_i, c_i, d_i. However, due to the condition of us needing $n-1$, we cannot find b_0 (since that requires us to know x_{-1} and y_{-1} which we do not know. Moreover we cannot find b_n , since we do not not the points x_n and y_n

The "not-a-knot" allows us to set up the equations:
To find b_0 of the $g_0(x)$ equation:

$$3h_0 + 2h_1 + h_2 \cdot 0 + h_1 b_1 + (h_1 - h_2) \cdot 0 + h_1 b_2 = 3h_i(a_2 - a_1) - 3h_0(a_1 - a_0)$$

To find the b_n of the $g_n(x)$ equation:

$$h_{n-2} - h_{2,n-1} h_{n-2} b_{n-2} + (3h_{n-1} + 2h_{n-2} + h_{2,n-1}) h_{n-2} b_{n-1} = 3h_{n-1}(a_n - a_{n-1}) - 3h_{n-2}(a_{n-1} - a_{n-2})$$

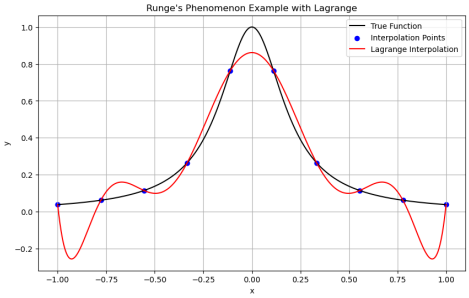
By first solving the matrix and then calculating the coefficient endpoints by the given "Not-a-knot" condition all the coefficients $a_i, b_i, c_i, d_i, i \in (0, n-1)$ can be found.

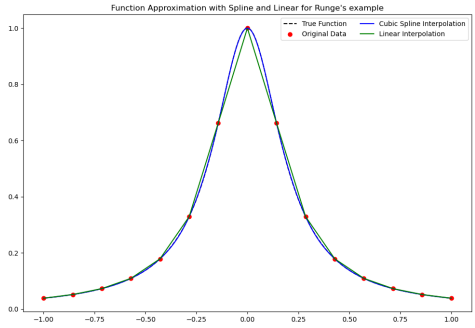
4 Perks

This chapter examines the perks of cubic spine interpolation alongside some examples.

4.1 Oscillation

Runge's phenomenon is a phenomenon in numerical analysis and interpolation theory that occurs when using high-degree polynomial interpolation to approximate a function, especially when using equidistant interpolation points. Named after the German mathematician Carl David Tolmé Runge, this phenomenon is characterized by oscillations and overshooting in the interpolated function, particularly near the edges of the interpolation interval. The general idea is that as the degree of the interpolating polynomial increases, the oscillations in the interpolated function become more pronounced, leading to poor approximation, even if the function being approximated is smooth. This is particularly evident when equally spaced interpolation points are used. Runge's phenomenon again underscores the importance of carefully choosing interpolation methods and a number of points depending on the case.





Above in the left figure, we are addressing Runge's famous counterexample for interpolation: $f(x) = \frac{1}{1+25x^2}$ with Lagrange's single polynomial technique. If this function is interpolated at equally spaced points in the interval [-1,1], the polynomials do not converge uniformly. The maximum error goes to infinity. In comparison, the right figure shows how spline solves the Problem of Runge's Phenomenon for the same function. We can conclude, based on the figure, that Runge's phenomenon of polynomials not converging can be avoided by spline interpolation instead of single polynomial.

4.2 Volatile Functions

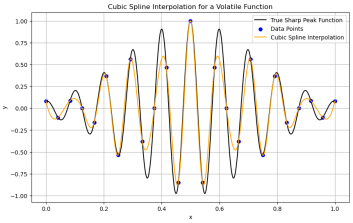
Spline interpolation is often considered a data-driven approach, where the shape of the interpolating function is influenced by the data distribution. This can lead to better generalization and less sensitivity to outliers.

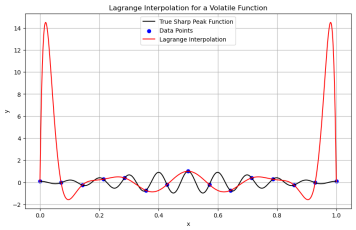
High oscillations in a true function can lead to overshooting and undershooting when using interpolation methods like Lagrange. The interpolating polynomial might try to fit the peaks and troughs of the oscillations, resulting in a polynomial that doesn't closely follow the true behavior of the function.

Lagrange interpolation constructs a global polynomial that aims to fit the entire interval. This global nature can be a limitation when dealing with functions that exhibit localized volatility. The interpolating polynomial might not respond well to rapid changes within small intervals.

Piecewise interpolation methods, such as spline interpolation, can be more effective for functions with high oscillations. By breaking the interval into smaller segments and using lower-degree polynomials for each segment, splines offer smoother transitions between data points and can better capture localized volatility.

Let's again compare the performance of interpolations on such highly oscillating or volatile functions: With a relatively high number of points to approximate a function with 30 in our case in the left

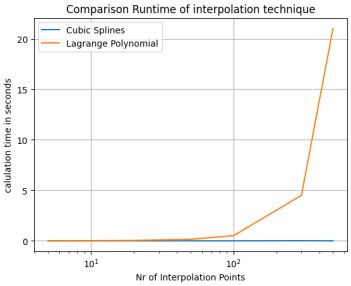


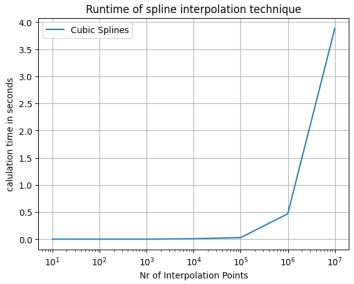


figure, Spline does a great job even for a highly oscillating function. However, as noted, we need more points which increases computation time. The right figure was one of the best results for a Lagrange over this interval with an optimal number of points chosen. We can see its generalization of the whole interval with one complex function fails to capture the local behavior of highly volatile functions.
In conclusion, it can be said that spline interpolation is highly suitable to handle highly volatile (local complex behavior) functions when Lagrange is too simple and generalizing.

4.3 Running Time

For this example, we compared the scipy implementation of CubicSpline against the Lagrange Polynomial approximation of scipy. We ran both on different types and sizes of point databases. Here is one representative finding:





The left figure compares the number of the points database against the calculation time of the scipy implementations to find the Lagrange Polynomial and Cubic Spline Polynomials. It is visible that Lagrange's running time increases faster, and has higher order. In the right figure above, it can be seen that Cubic Spline running increases significantly for a higher number of points in the database.

5 Conclusion

Cubic splines are found by setting certain conditions of continuity, which allows the creation of a solvable linear system for all $n-1$ spline polynomials. When this linear system is implemented and set up in a certain way (our example scipy with "knot-to-knot" endpoint condition) the calculation time for all spline polynomials is significantly faster, in comparison to single polynomial interpolation techniques as Lagrange. Furthermore, we also found that Cubic Splines are less prone to oscillate, in comparison to Lagrange Interpolation.