Tension Spline

MATH 212 - Numerical Analysis

In alphabetical order,
Mahdi Boulila, Meherab Hossain, Cat Mai, Daniel Wright

1 Abstract

Interpolation of data is one of many procedures in physical sciences, data sciences, and applied mathematics, and is accomplished typically using an averaging or finite difference scheme on an equidistant grid. The approach is to divide a given interval of points $x_1, x_2, \ldots, x_n$ into a collection of sub-intervals and construct an approximating polynomial on each sub-interval. Approximation by functions of this type is called piece-wise-polynomial approximation. Simplest approximation is to calculate a quadratic polynomial on each sub-interval (i.e. between every two successive nodes). For better smoothing, cubic splines comes in handy; They are popular because of their smooth appearances of the function passing through all the nodes, and the accessible deployment in many environments. Cubic splines ensure that not only the function is continuous, but its first and second derivatives are also continuous. However, these functions can have undesirable oscillations between data points. This is where adding a tension parameter helps getting rid of unnecessary oscillations for the polynomial approximation. This stability ensures a delicate and precise creation of any curve on any interval without worrying about its effect on other nodes. Modified versions of tension spline are implemented everywhere, from game development to graphics design. This paper explains the mathematical foundations behind the tension spline, and presents the results of graphic design attempts with the interpolation method.

2 Introduction

The tension spline consists of a rather complicated hyperbolic function between every pair of consecutive nodes. Similar to the cubic spline, the tension spline passes through the nodes and two functions on either side of a node has the same first and second derivative values. One could also set the same boundary conditions on the tension spline (ex. Natural boundary, clamped boundary, etc). However, unlike the cubic spline, the tension spline function has its fourth derivative set to a function of its second derivative multiplied by a factor τ, the desired tension factor. One may notice that the cubic spline function is of order 3. By setting τ to zero we can set the fourth derivative of our tension spline to zero as well and have ourselves the cubic spline. In our project, we explore the tension spline interpolation by coding its functionality and using its results to draw simple figures of a car . One may note that drawing figures would require us to draw in sections as to not confuse the algorithm with points that may seem to be in consecutive order but are

required to draw different parts. The function between two points n_i and n_{i+1} also requires two coefficients, τ and z_i. z_i can be found as a matrix calculation of a system of tridiagonal equations. Here, the coefficients α and β and equation values γ are functions of y_i, x_i and τ. With 20-30 points, we notice the calculation and plotting were relatively fast. The higher the τ values, the straighter/'smoother' our plots were. With lower values, the interpolation seems to curve more and tends to oscillate like the cubic spline. Sections of our image were relatively straight plotted well. Plotting other parts which have varying smoothness is rather difficult and one would have to play around with τ values or increase the number of points in the areas that require more curvature. However, adding more and more points was computationally expensive since the complexity of our algorithm is of $O(n^3)$. In our case, this wasn't an issue since we added a few more intermediary nodes to ease curvature. One could split sections up further for varying smoothness with a natural boundary conditions although this would require more work to code sections of points and add to the plotting complexity. Although there are more implications of the τ values, we will discuss this in a later section.

3 Theory

3.1 Interpolation

In order to analyze discrete data points we often wish to interpolate them into a function. The standard form of that function is a polynomial of degree $n-1$ which interpolates n points. However when dealing with a large number of points that function will be inaccurate due to the function dipping and rising $n-1$ times (The drawback to polynomial interpolation is that the more points it take, the more oscillatory it becomes). It therefore might be more accurate to interpolate the function with a series of piece-wise functions along an area of our data. The simplest way to do that is connecting each data point to the next by a line -which is referred to as linear spline. It is often better to modify the spline into a cubic spline where each set of points are connected by a cubic polynomial, instead of an $n-1$ degree polynomial.

3.2 Cubic Spline

First, we limited the degree of the polynomial. Second, we imposed conditions on each partial function so that the entire function is continuous as is its first and second derivative. We used these conditions to create system of equations for the coefficients for each piece (i.e. $[x_i, x_{i+1}]$). Since each sub-function has four coefficients, we need to have at least $4n$ conditions. We define our n functions as $S_1, S_2, \ldots, S_n$ with

$$S_n = a_n x^3 + b_n x^2 + c_n x + d_n \tag{1}$$

Our first condition, the value condition can be defined on the i^{th} given point as

$$S_n(x_i) = f(x_i) = y_i \tag{2}$$

meaning that at each point where we have a value for $f(x)$, our interpolation function, must have the same value. This applies for every discrete point we have $(x_0, x_1, \ldots, x_n)$, giving us $n+1$ conditions. Our next condition states that

$$S_i(x_{i+1}) = S_{i+1}(x_{i+1}) \tag{3}$$

What that means is that each piece of the function S_i must match the following function at the point where they meet x_i.In order to ensure that the function is continuous everywhere (in $[x_0, x_n]$), we must apply the same condition to each S_n's first and second derivative. this gives us the following three conditions:

$$S_i(x_{i+1}) = S_{i+1}(x_{i+1}) \tag{4}$$
$$S_i'(x_{i+1}) = S_{i+1}'(x_{i+1}) \tag{5}$$
$$S_i''(x_{i+1}) = S_{i+1}''(x_{i+1}) \tag{6}$$

Since these conditions do not apply to the first or last points each of these three types conditions give $n-1$ conditions, for a total of $3n-3$. If we add on the first set of conditions, we have $4n-2$ conditions. Therefore, we impose one condition each on the first and last point to make sure that the interpolation stays continuous.

There are two different pairs of conditions we can apply to the boundaries: First we have natural boundary conditions, where the (2^{nd} derivatives of the boundary points are 0). This gives us the last two conditions and the ability to solve the linear system we currently have. Second, we have clamped boundary conditions: for every value of $f'(x_0)$ and $f'(x_n)$, we set them equals to $S'(x_0)$ and $S'(x_n)$ accordingly. Now, With the resulting $4n$ conditions, we are able to solve the system of equations and get a unique continuous interpolation across our discrete data set.

3.3 Tension Spline

In some cases the cubic spline does not provide an accurate picture of the shape of the graph. In those cases we often use a tension spline. A tension spline is a cubic spline which has had tension factor applied to it, to stretch the graph closer to the given points. A tension factor τ is a number that we use to generate different conditions for our spline. When our tension factor is 0 we generate a normal cubic spline. With higher tension values the function looks more and more like a linear spline. The limit of the function as tension approaches infinity is the linear spline. We calculate Tension Splines similarly to the way we calculate cubic splines, but using different conditions to generate our formulas. The most important condition is the tension condition where along the entirety of our function $f^4 - \tau^2 f'' = 0$ We can see through this condition why a tension of 0 gives a cubic spline. Because when $\tau = 0$ that condition is $f^4 = 0$ which all cubic functions will satisfy. We set $f''(x_i) = z_1$ which allows us to apply the tension condition to get us the formula for the interval $[t_i, t_{i+1}]$:

$$\begin{aligned} f(x) = \{z_i \sinh(\tau(t_{i+1} - x)) + z_{i+1} \sinh(\tau(x - t_i))\}/(\tau^2 \sinh(\tau h_i)) \\ +(y_i - z_i/\tau^2)(t_{i+1} - x)/h_i + (y_{i+1} - z_{i+1}/\tau^2)(x - t_i)/h_i \end{aligned} \tag{7}$$

Applying that formula for the entire range of our interpolation gives us a tension spline, but we first must solve for the values of $z_0, z_1, \ldots, z_n$. We have n+1 unknowns meaning we need to have n+1 conditions. We use the derivative condition that we used on cubic splines:

$$f_i'(x_{i+1}) = f_{i+1}'(x_{i+1}) \tag{8}$$

which gives us $n-1$ conditions. We get our final two conditions the same way as in a natural cubic spline, by setting $S_0'' = S_n'' = 0$. That gives us two conditions for a total of n+1 conditions allowing us to solve the system of equations.

4 Numerical Computations

In this section, we present our numerical computation of the tension spline interpolants. First, we present the necessary machinery for the setup and our pseudo-code. Then, we discuss runtime complexity and error estimation. Finally, we show visualizations of the resulting interpolation with corresponding abscissas and ordinates.

4.1 Tridiagonal System of Equations

Using the above conditions, we can derive a tridiagonal system of equations for the unknown interpolants $z_0, \ldots, z_n$ that can be written as:

$$\alpha_{i-1} z_{i-1} + (\beta_{i-1} + \beta_i) z_i + \alpha_i z_{i+1} = \gamma_i - \gamma_{i-1} \tag{9}$$

where

$$\begin{gathered} 1 \leq i \leq n-1 \\ h_i = x_{i+1} - x_i \\ \alpha_i = 1/h_i - \tau/\sinh(\tau h_i) \\ \beta_i = \tau \cosh(\tau h_i)/\sinh(\tau h_i) - 1/h_i \\ \gamma_i = \tau^2 (y_{i+1} - y_i)/h_i \end{gathered}$$

From Equation (8), we obtain $n-1$ conditions. We then apply two boundary conditions, $z_0 = z_n = 0$, which gives us $n+1$ conditions allowing us to solve the system of equations.

$$\mathbf{A} \cdot \mathbf{Z} = \mathbf{Y}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\ \alpha_0 & \beta_0 + \beta_1 & \alpha_1 & 0 & 0 & 0 & \ldots & 0 \\ 0 & \alpha_1 & \beta_1 + \beta_2 & \alpha_2 & 0 & 0 & \ldots & 0 \\ 0 & 0 & \alpha_1 & \beta_1 + \beta_2 & \alpha_2 & 0 & \ldots & 0 \\ & & & \vdots & & & & \\ 0 & 0 & \ldots & 0 & \alpha_{n-2} & \beta_{n-2} + \beta_{n-1} & \alpha_{n-1} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \ldots & 1 \end{bmatrix} \begin{bmatrix} z_0 = 0 \\ z_1 \\ \vdots \\ \\ \\ z_{n-1} \\ z_n = 0 \end{bmatrix} = \begin{bmatrix} \gamma_0 \\ \gamma_1 \\ \vdots \\ \\ \\ \gamma_{n-1} \\ \gamma_n \end{bmatrix}$$

Solving for $\mathbf{Z}$ giving z_i and plugging z_i in Equation (7), we obtain our tension spline interpolating polynomial.

4.2 Pseudo-Code

We present our pseudo-code at next page for calculating the coefficients. Actual implementation is in Section 7.

4.3 Time Complexity

4.3.1 Convergence Rate

For an interpolation polynomial, we can investigate for which class of functions or interpolating nodes that our polynomial S_n converges to the actual function f as $n \to \infty$. There

Algorithm 1: Computing Tension Spline Interpolants

Data: $(x_0, y_0), \ldots, (x_n, y_n)$ where $x_0 < x_1 < \cdots < x_n$ and $\tau \in \mathbb{Z}^+$
Result: $\mathbf{Z} = z_0, \ldots, z_n$
begin
for *i = 0, . . . , n-1* **do**
  compute $h_i, \alpha_i, \beta_i, \gamma_i$;
end
Let $\mathbf{A} = (n+1) \times (n+1)$ matrix, $\mathbf{Z}$ and $\mathbf{Y}$ both $(n+1) \times 1$ matrices.
set $\mathbf{A}_{0,0} = \mathbf{A}_{n,n} = 1$
for *i = 1, . . . , n-1* **do**
  $\mathbf{A}_{i,i-1} = \alpha_{i-1}$;
  $\mathbf{A}_{i,i} = \beta_{i-1} + \beta_i$;
  $\mathbf{A}_{i,i+1} = \alpha_i$;
  $\mathbf{Y}_{i,0} = \gamma_i - \gamma_{i-1}$;
end
Solve for $\mathbf{Z}$ in $\mathbf{A} \cdot \mathbf{Z} = \mathbf{Y}$.
end

is a very close relationship between tension splines and cubic splines as mentioned, however, there are more results pertaining to the latter. Pruess [1] exploited many known results about convergence rate for cubic splines to show a tighter bound on tension splines, which has $O(h^4)$ uniform convergence rate for uniformly bounded parameters, where $h = \max_i(x_{i+1} - x_i)$.

Theorem 1 *(Pruess, 1976) If $f \in C^4[a, b]$ and S_n is our interpolating tension spline polynomial with natural boundary conditions, then $\exists K$ dependent on $\tau, ||D^2 f||, ||D^4 f||$ but independent of h, such that*

$$||D^i(f - S_n)|| \leq Kh^{4-i} \qquad \text{for } i = 0, 1, 2$$

where $D = \frac{d}{dx}$. Natural boundaries being:

$$S_n'(a) = f'(a) \qquad S_n'(b) = f'(b)$$

4.3.2 Computation Complexity

The overall algorithm runs in $O(n^3)$, derived from matrix decomposition runtime. Assuming constant lookup, the numerical setup should run in $O(n \log n)$, based on the runtime of polynomial evaluation of the hyperbolic functions, which is done through the polynomial expansion of e. Though Taylor expansion first comes to mind, most modern efficient libraries do not use Taylor expansion but other methods e.g. Chebyshev polynomials. Some polynomial approximation schemes include Estrin/Horner's method, Chebyshev/FFT and Remez/minimax approximation. The Numpy library we use calls on the underlying GNU C library which approximates e by evaluating a 7^{th}-degree polynomial using some form

of minimax algorithm per the source code[1]. This has satisfied our curiosity, however, the dominant time factor is the process of solving for the matrix $\mathbf{Z}$.

Generally, most matrix decomposition algorithm has $O(n^3)$ time complexity but actual flops[2] can vary. Gaussian elimination with partial pivoting for LU decomposition has $2n^3/3$ flops [5]. Using the fact that our matrix is positive semidefinite, we can use Cholesky factorization with $n^3/3$ flops [5]. For much faster speed as a trade-off for numerical stability, Thomas' Tridiagonal matrix algorithm is a simplified version of Gaussian elimination specifically for tridiagonal and diagonally dominant matrix, which runs in $O(n)$ [4]. Results in this report are produced using Numpy's built-in matrix inverse function[3], which factorizes a matrix using LU decomposition then uses backward-forward substitution.

4.4 Error Estimation

In general, the error of interpolation polynomial S_n with $f \in C^{n+1}[a, b]$ is

$$f(x) - S_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i) \qquad \xi \in (a, b).$$

Hall and Meyer [2] has established the error bound for cubic spline interpolation with natural boundary conditions and bounded ratio $\frac{h_{i+1}}{h_i}$. We let $h = \max_i(x_{i+1} - x_i)$.

Theorem 2 *(Hall and Meyer, 1976) For $f \in C^4[a, b]$ and $S_n'(a) = f'(a), S_n'(b) = f'(b)$ where S_n is the cubic spline interpolating polynomial, the error bound is*

$$||f(x) - S_n(x)|| \leq \frac{5}{384} h^4 ||f^{(4)}||$$

In order to bound the error for interpolatory tension spline, one approach is that we can establish the bounded difference between the cubic spline and tension spline, and apply Theorem 2. Pruess [1] has derived the bounded difference between the two splines.

Using the same interpolating nodes and boundary conditions, for a node x_i, let α_i be the interpolatory cubic spline and β_i be the interpolatory tension spline. Then,

Theorem 3 *(Pruess, 1976) Bounded difference between cubic and tension spline*

$$||\alpha - \beta|| \leq \frac{26}{3} \tau^2 h^4 \max_i |\alpha_i''|$$

Since there exists a second derivative for cubic spline interpolation in Theorem 3, we now use Hall and Meyer's result for bounding the error of α''.

Theorem 4 *(Hall and Meyer, 1976) For $f \in C^4[a, b]$ and $S_n'(a) = f'(a), S_n'(b) = f'(b)$ where S_n is the cubic spline interpolating polynomial, the error bound is*

$$||f''(x) - S_n''(x)|| \leq \frac{3}{8} h^2 ||f^{(4)}||$$

4.5 Visualizations

Here we present our result on two different shapes, the top of a car and a shark.

[1] GNU C source code for evaluating e

[2] floating point operations per second

[3] source code for numpy's inverse function

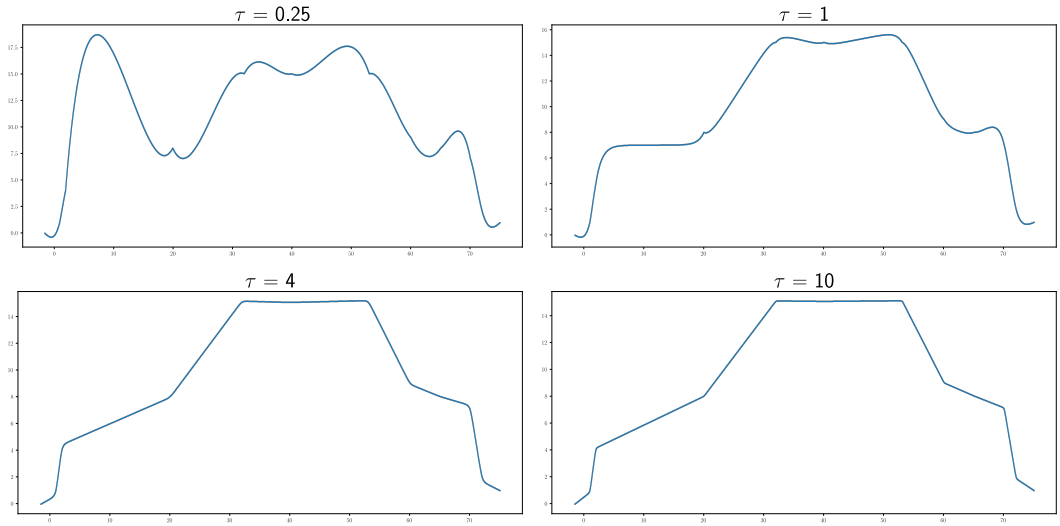

Figure 1: Tension spline interpolation for the same set of data points but with different τ, resembling a car. The lower τ is, the "softer" the curve looks at each tension point. For lower τ values, the plots of resulting functions have points that look non-differentiable. This is likely due to big jumps the abscissas at those nodes. Adding more nodes in-between will result in smoother-looking functions.

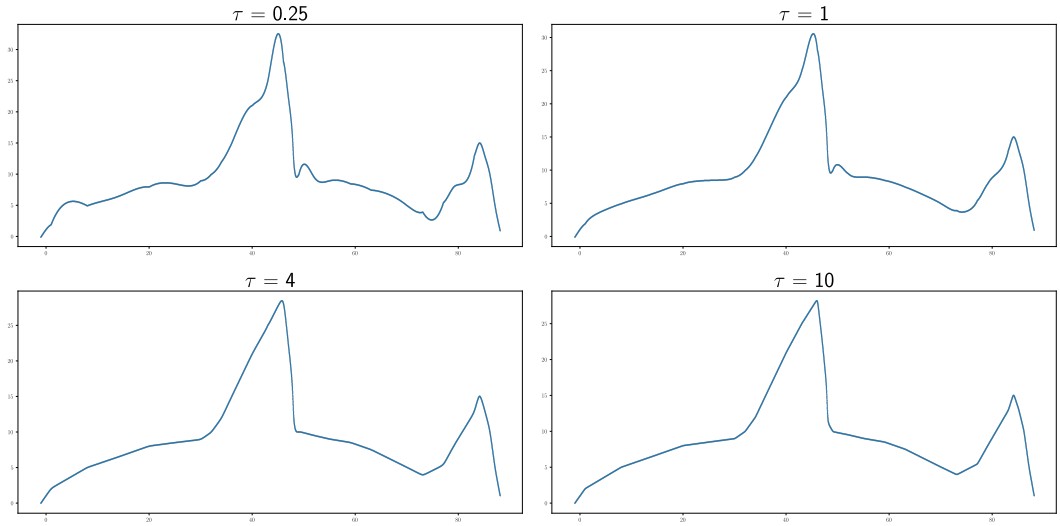

Figure 2: Tension spline interpolation for the same set of data points but with different τ, resembling a shark.

x_i	-1.5	1	2	20	32	40	53	60	65	70	72	75
y_i	0	1	4	8	15	15	15	9	8	7	2	1

Table 1: Nodes for the cars in Figure 1. $n = 11$ with 12 data points.

x_i	-1	1	8	14	20	25	30	32	34	36	40	43	46	47	47.75	48	49
y_i	0	2	5	6.5	8	8.5	9	10	12	15	21	25	28	22	16	12	10

Table 2: Nodes for the sharks in Figure 2, truncated for view. $n = 30$ with 31 data points.

5 Conclusion

From our results we see that the tension spline performs well when it comes to plotting simple figures. Even at low values τ, given enough nodes, the function does not oscillate as much. The algorithm is fast even with a large number of data points given the number of zeroes in our A matrix eases calculations. However,it has a tendency oscillate when the nodes are sparse and τ is relatively low. The function curves irregularly as seen in Figure 1. A benefit to this interpolation is that could theoretically approximate the cubic spline and natural spline interpolation. For the natural spline approximation, we need $\tau \to \infty$. When setting τ to high values, i.e 70 or 80, one could fairly approximate the natural spline interpolation. On the other hand, large values of τ hamper calculations and complexity to the point where the algorithm is unable to plot anything.

In terms of plotting, our tension spline interpolation can only plot simple silhouettes that pass the vertical line test. Forcing the algorithm to plot figures would result in gaps between nodes. This may cause issues with a lot of drawings and images may have to plotted section wise. Similarly, points have to added in order of connection (increasing x_i values). Another issue would be that varying curvature. When plotting the image of the lower half of the car, one would have to toy around the τ value to find one that plot the chassis straight but also keep the tyres curved. Thus for *nice* image, the tyres should be plotted separately with a good number of points. Plotting the different sections together also adds onto the intricacy of our algorithm and plotting time.

In terms of implementation, the tension spline interpolation could be used in games design to create low resolution backgrounds with a low τ value. Another usage could be modelling economic trends given it's nature to smooth out near nodes.

6 References

[1] Pruess, S. (1976). Properties of splines in tension. Journal of Approximation Theory, 17, 86-96.

[2] C.A Hall, W.W Meyer. Optimal error bounds for cubic spline interpolation. J. Approximation Theory, 16 (1976), pp. 105-122

[3] Costantini, Paolo & Kvasov, Boris & Manni, Carla. (1999). On discrete hyperbolic tension splines. Adv. Comput. Math.

[4] Tridiagonal matrix algorithm - TDMA (Thomas algorithm) – CFD-Wiki, the free CFD reference. (2021). Retrieved 2 December 2021, from CFD-Online

[5] Hunger, R. (2007) Floating Point Operations in Matrix-Vector Calculus.

7 Appendix

7.1 Python Implementation

```
t = [-1,1,8,14,20,25,30,32,34,36,40,43,46,47,47.75,48,49,52,55,59,63,70,73,
     77,79,83,84,85,86,87,88]
y = [0,2,5,6.5,8,8.5,9,10,12,15,21,25,28,22,16,12,10,9.5,9,8.5,7.5,5,4,5.5,
     8,13,15,13,10,5,1]
```

Listing 1: Coordinates for the shark plots in Figure 2

```
import numpy as np
#NOTICE: In Python, range(n) means [0,n)

t = [-1.5,1,2,20,32,40,53,60,65,70,72,75]
y = [0,1,4,8,15,15,15,9,8,7,2,1]

# check if t's are sorted and t's and y's have the same length
assert t == sorted(t)
assert len(t) == len(y)

tau = 10
n = len(t)-1

# corresponding to h, alpha, beta, gamma. Lists of n 0s
h, a, b, g = [0]*n, [0]*n, [0]*n, [0]*n

# calculating the variables
for i in range(n):
    h[i] = t[i+1] - t[i]
    g[i] = tau**2 *(y[i+1] - y[i])/h[i]
    a[i] = 1/h[i] - tau/np.sinh(tau*h[i])
    b[i] = tau*np.cosh(tau*h[i])/np.sinh(tau*h[i]) - 1/h[i]

# initializing A,Z,Y matrices
A = np.zeros((n+1,n+1))
for row in range(1,n):
    A[row][row-1] = a[row-1]
    A[row][row] = b[row-1] + b[row]
    A[row][row+1] = a[row]
A[0][0] = 1
A[n][n] = 1

Y = np.zeros((n+1,1))
for i in range(1, n):
    Y[i][0] = g[i] - g[i-1]

# convert python list to matrix
```

```
Y = np.asmatrix(Y)
A = np.asmatrix(A)

# use inverse of matrix to solve for Z
Z = np.linalg.inv(A)*Y

# calculating the actual polynomial
def f(x,i):
    t1 = (z[i]*np.sinh(tau*(t[i+1]-x))) + (z[i+1]*np.sinh(tau*(x-t[i])))
    t1 = t1/(tau**2 * np.sinh(tau*h[i]))
    t2 = (y[i]-z[i]/tau**2)*(t[i+1]-x)/h[i]
    t3 = (y[i+1]-z[i+1]/tau**2)*(x-t[i])/h[i]
    return float(t1 + t2 + t3)
```

Listing 2: Python implementation of finding tension spline interpolants