2018-08-23

Cubic Interpolation with Irregularly-Spaced Points in Julia 1.4

R. Steven Turley
Brigham Young University, turley@byu.edu

Cubic Interpolation with Irregularly Spaced Points

Version 1.2

R. Steven Turley

November 19, 2020

Contents

1. Introduction

This is the mathematics and some implementation details behind a derivation of 1d cubic piece-wise continuous interpolation with regularly and irregularly spaced points. I will

explore two ways to compute this cubics: splines and Hermite polynomials. Both are continuous and have continuous derivatives at the knots. Splines also have continuous second derivatives.

1.1. Versions

Version 1.2: I added some insights about the scaling of the intervals with irregularly-spaced splines and fixed a bug in the code for finding the interpolation region for irregularly-spaced splines.

Version 1.1: I made minor revisions in the derivations and updated to code to reflect changes needed to fix bugs discovered which I was using splines for calculations of reflections with rough surfaces.

Version 1.0: Original document used for finding peaks in U161 reflectance data and spline integrations for roughness calculations.

2. Cubic Splines

I will use the article on splines for a regularly-spaced grid in MathWorld[1] as a basis for my derivations and generalizations. I have also relied on the Dierckx book[2] for information about B-splines, smoothing splines, and the routines in the FITPACK library.

Splines are piece-wise cubic polynomials which are continuous and have continuous first and second derivatives. In each interval it takes four coefficients to define a cubic polynomial. If there are $n+1$ points, there are n intervals requiring $4n$ coefficients for the splines. Let the knots on the spline (the data points that match exactly) be (x_i, y_i). Let $Y_i(x)$ be the cubic polynomial for the interval i where $x_i \le x \le x_{i+1}$. Then the $4n-4$ conditions for matching the points and having continuous first and second derivatives are for $2 \le n \le n$

$$Y_{i-1}(x_i) = y_i \tag{1}$$
$$Y_i(x_i) = y_i \tag{2}$$
$$Y'_{i-1}(x_i) = Y'_i(x_i) \tag{3}$$
$$Y''_{i-1}(x_i) = Y''_i(x_i)\ . \tag{4}$$

In addition to these equations, the spline also needs to match at the two endpoints.

$$Y_1(x_1) = y_1 \tag{5}$$
$$Y_n(x_{n+1}) = y_{n+1} \tag{6}$$

This gives a total of $4n-2$ equations and $4n$ unknowns. There are several ways to choose the last two conditions. I will use the specification that the second derivative be zero at the two endpoints.

$$Y''_1(x_1) = 0 \tag{7}$$
$$Y''_n(x_{n+1}) = 0 \tag{8}$$

2.1. Regularly-Spaced Points

The spline equations can be solved with a particularly elegant form for the case of equally spaced knots. It is useful to put the origin of each cubic at the beginning of the interval and transform to a variable t which goes from 0 to 1 in each interval i.

$$x = x_i + \alpha t \qquad \text{for} \quad x_i \le x \le x_{i+1} \tag{9}$$

$$\alpha = x_{i+1} - x_i \quad \forall i \tag{10}$$

If the intervals are of equal length, the conditions of continuity of a derivative with respect to x is the same as a derivative with respect to t. Equations 1 through 4 are then

$$Y_{i-1}(1) = y_i \tag{11}$$

$$Y_i(0) = y_i \tag{12}$$

$$Y'_{i-1}(1) = Y'_i(0) \tag{13}$$

$$Y''_{i-1}(1) = Y''_i(0). \tag{14}$$

Let the four coefficients of the cubic for interval i be given by

$$Y_i(t) = a_i + b_i t + c_i t^2 + d_i t^3. \tag{15}$$

Then these coefficients can be solved for in terms of the values y_i and the derivatives $D_i = Y'_i(0)$.

$$Y_i(0) = y_i = a_i \tag{16}$$

$$Y_i(1) = y_{i+1} = a_i + b_i + c_i + d_i \tag{17}$$

$$Y'_i(0) = D_i = b_i \tag{18}$$

$$Y'_i(1) = D_{i+1} = b_i + 2c_i + 3d_i \tag{19}$$

These equations can be solved for the cubic coefficients in terms of y_i and D_i.

$$a_i = y_i \tag{20}$$

$$b_i = D_i \tag{21}$$

$$c_i = 3(y_{i+1} - y_i) - 2D_i - D_{i+1} \tag{22}$$

$$d_i = 2(y_i - y_{i+1}) + D_i + D_{i+1} \tag{23}$$

Weisstein shows that these equations can be rewritten as the matrix equation

$$\begin{pmatrix} 2 & 1 & & & & & & \\ 1 & 4 & 1 & & & & & \\ & 1 & 4 & 1 & & & & \\ & & 1 & 4 & 1 & & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & & & 1 & 4 & 1 & \\ & & & & & 1 & 2 & \end{pmatrix} \begin{pmatrix} D_1 \\ D_2 \\ D_3 \\ D_4 \\ \vdots \\ D_n \\ D_{n+1} \end{pmatrix} = \begin{pmatrix} 3(y_2 - y_1) \\ 3(y_3 - y_1) \\ 3(y_4 - y_2) \\ \vdots \\ 3(y_n - y_{n-2}) \\ 3(y_{n+1} - y_{n-1}) \\ 3(y_{n+1} - y_n) \end{pmatrix}. \tag{24}$$

My derivation of this is in Appendix A. Equation 24 can be solved with an efficient symmetric tridiagonal solver in Julia for the unknown values D_i. Once those are known, Equations 20 through 23 can be used to solve for a_i, b_i, c_i, and d_i.

2.2. Irregularly-Spaced Points

If the points x_i are not regularly spaced, α in Equations 9 and 10 needs to be replaced with $\alpha_i = x_{i+1} - x_i$ which will vary in each interval. It also becomes more sensible to define Y_i as a function of x instead of a function of t.

$$Y_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3. \tag{25}$$

Letting D_i be a derivative of x instead of t,

$$Y_i(x_i) = y_i = a_i \tag{26}$$
$$Y_i(x_{i+1}) = y_{i+1} = a_i + b_i\alpha_i + c_i\alpha_i^2 + d_i\alpha_i^3 \tag{27}$$
$$Y_i'(x_i) = D_i = b_i \tag{28}$$
$$Y_i'(x_{i+1}) = D_{i+1} = b_i + 2c_i\alpha_i + 3d_i\alpha_i^2 \tag{29}$$

These can be solved for a_i, b_i, c_i and d_i as before.

$$a_i = y_i \tag{30}$$
$$b_i = D_i \tag{31}$$
$$c_i = 3\frac{y_{i+1} - y_i}{\alpha_i^2} - 2\frac{D_i}{\alpha_i} - \frac{D_{i+1}}{\alpha^i} \tag{32}$$
$$d_i = 2\frac{y_i - y_{i+1}}{\alpha_i^3} + \frac{D_i}{\alpha_i^2} + \frac{D_{i+1}}{\alpha_i^2} \tag{33}$$

With the change of variables, the first and second derivative equations with respect to x are the same as the conditions on the derivatives with respect to t. Appendix B derives the following matrix equation as a solution for D_i in terms of y_i and α_i.

$$\begin{pmatrix} 2\alpha_1^{-1} & \alpha_1^{-1} & & & & \\ \alpha_1^{-1} & 2(\alpha_1^{-1} + \alpha_2^{-1}) & \alpha_2^{-1} & & & \\ & \alpha_2^{-1} & 2(\alpha_2^{-1} + \alpha_3^{-1}) & \alpha_3^{-1} & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \\ & & \alpha_{n-1}^{-1} & 2(\alpha_{n-1}^{-1} + \alpha_n^{-1}) & \alpha_n^{-1} & \\ & & & \alpha_n^{-1} & 2\alpha_n^{-1} \end{pmatrix} \begin{pmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \\ D_n \\ D_{n+1} \end{pmatrix}$$
$$= \begin{pmatrix} 3(y_2 - y_1)\alpha_1^{-2} \\ 3[y_3\alpha_2^{-2} + y_2(\alpha_1^{-2} - \alpha_2^{-2}) - y_1\alpha_1^{-2}] \\ 3[y_4\alpha_3^{-2} + y_3(\alpha_2^{-2} - \alpha_3^{-2}) - y_2\alpha_2^{-2}] \\ \vdots \\ 3[y_{n+1}\alpha_n^{-2} + y_n(\alpha_{n-1}^{-2} - \alpha_n^{-2}) - y_{n-1}\alpha_{n-1}^{-2}] \\ 3(y_{n+1} - y_n)\alpha_n^{-2} \end{pmatrix}. \tag{34}$$

Equation (34) can be seen to be a special case of Equation (24) by letting $\alpha_i = \alpha$.

$$\frac{1}{\alpha}\begin{pmatrix} 2 & 1 & & & & & & \\ 1 & 4 & 1 & & & & & \\ & 1 & 4 & 1 & & & & \\ & & 1 & 4 & 1 & & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & & & 1 & 4 & 1 & \\ & & & & & 1 & 2 & \end{pmatrix}\begin{pmatrix} D_1 \\ D_2 \\ D_3 \\ D_4 \\ \vdots \\ D_n \\ D_{n+1} \end{pmatrix} = \frac{1}{\alpha^2}\begin{pmatrix} 3(y_2 - y_1) \\ 3(y_3 - y_1) \\ 3(y_4 - y_2) \\ \vdots \\ 3(y_n - y_{n-2}) \\ 3(y_{n+1} - y_{n-1}) \\ 3(y_{n+1} - y_n) \end{pmatrix} \tag{35}$$

$$\begin{pmatrix} 2 & 1 & & & & & & \\ 1 & 4 & 1 & & & & & \\ & 1 & 4 & 1 & & & & \\ & & 1 & 4 & 1 & & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & & & 1 & 4 & 1 & \\ & & & & & 1 & 2 & \end{pmatrix}\alpha\begin{pmatrix} D_1 \\ D_2 \\ D_3 \\ D_4 \\ \vdots \\ D_n \\ D_{n+1} \end{pmatrix} = \begin{pmatrix} 3(y_2 - y_1) \\ 3(y_3 - y_1) \\ 3(y_4 - y_2) \\ \vdots \\ 3(y_n - y_{n-2}) \\ 3(y_{n+1} - y_{n-1}) \\ 3(y_{n+1} - y_n) \end{pmatrix}. \tag{36}$$

Since D_i in Equation (24) are derivatives with respect to t and the D_i in Equation (34) are with respect to x, the factor of α is expected.

Even through the previous equations are mathematically correct, there is a problem if the vales of α_i are very small or very large. This could make the terms in Equation 32 and 33 to be of very different magnitude and therefore difficult to evaluate accurately. I originally throught it could also cause problems with accurately evaluating Equation 25, but changed my mind. In each term, the value of b_i, c_i, or d_i could become very big or very small, but it is offset by an equal factor in the other direction in $(x-x_i)^n$. To address this issue, I scaled the values of $alpha_i$ by their average values in the implementation.

2.3. Cubic Spline Results

2.3.1. Checking Validity

Figure 1 is my interpolation of a spline curve for regularly spaced points using my spline routine. Figure 2 is the same calculation using irregularly spaced points. Doing the same calculation in Matlab gives similar results. Figure 3 is the same calculation done in Matlab with irregularly spaced points. I also did a range of unit tests on the routine checking for satisfying the spline equations and for continuity of the function and its first two derivatives at the knots. All units tests were passed successfully.

2.3.2. Extra Oscillations

If there are jumps in the data making the curve look discontinuous, a spline can oscillate near the gap. Consider the data in Figure 4 as an example. In this case, a piecewise hermite polynomial may be a better way to go. Matlab recommends the pchip function[3, 4] based on hermite polynomials to address this issue. The spline interpolation in Matlab is identical to the Julia one. Figure 5 is what the pchip routine produces.

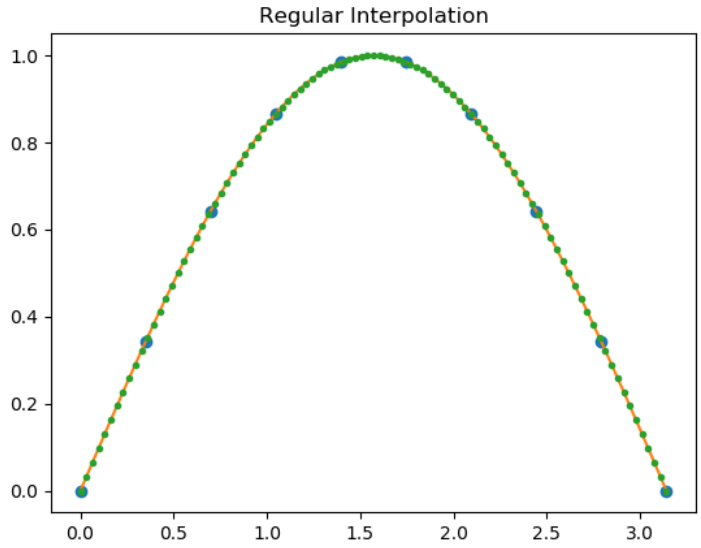

Figure 1: Spline interpolation with regularly spaced points for $\sin(x), x = 0, \pi$. The large circles are the data, the dotted line the exact curve and the orange line the spline curve.

3. Piece-wise Hermite Polynomials

I will follow Fritsch's notation[3]

$$f(x) = y_i H_1(x) + y_{i+1} H_2(x) + d_i H_3(x) + d_{i+1} H_4(x) \tag{37}$$

$$y_i = f(x_i) \tag{38}$$

$$d_i = f'(x_i) \tag{39}$$

$$h_i = x_{i+1} - xi \tag{40}$$

$$H_1(x) = \phi\left(\frac{x_{i+1} - x}{h_i}\right) \tag{41}$$

$$H_2(x) = \phi\left(\frac{x - x_i}{h_i}\right) \tag{42}$$

$$H_3(x) = -h_i \psi\left(\frac{x_{i+1} - x}{h_i}\right) \tag{43}$$

$$H_4(x) = h_i \psi(\left(\frac{x - x_i}{h_i}\right) \tag{44}$$

$$\phi(t) = 3t^2 - 2t^3 \tag{45}$$

$$\psi(t) = t^3 - t^2 \tag{46}$$

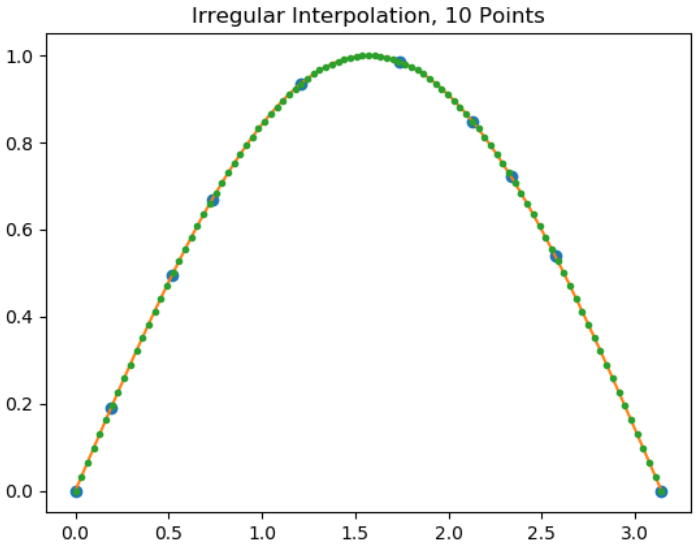

Figure 2: Spline interpolation with irregularly spaced points for $\sin(x), x = 0, \pi$. The large circles are the data, the dotted line the exact curve and the orange line the spline curve.

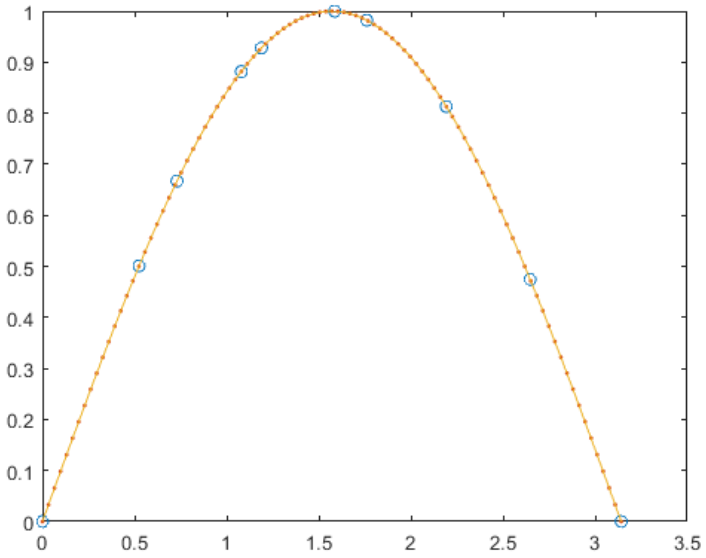

Figure 3: Spline interpolation using Matlab spline routine with irregularly spaced points. Compare to Figure 2.

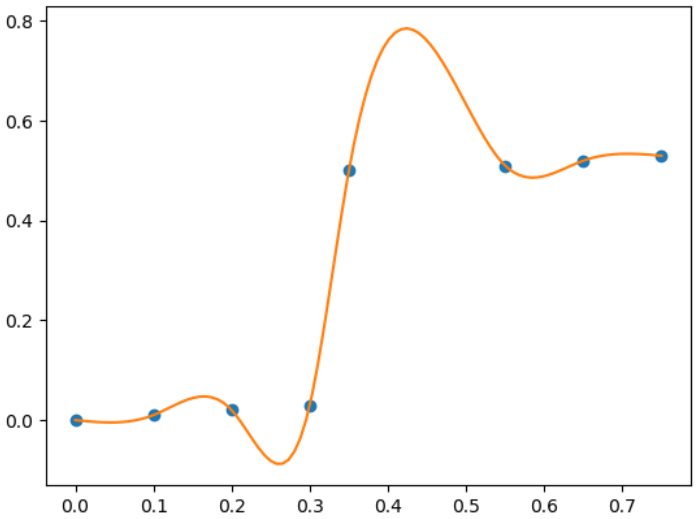

Figure 4: Cubic spline interpolation for data with a sudden bump.

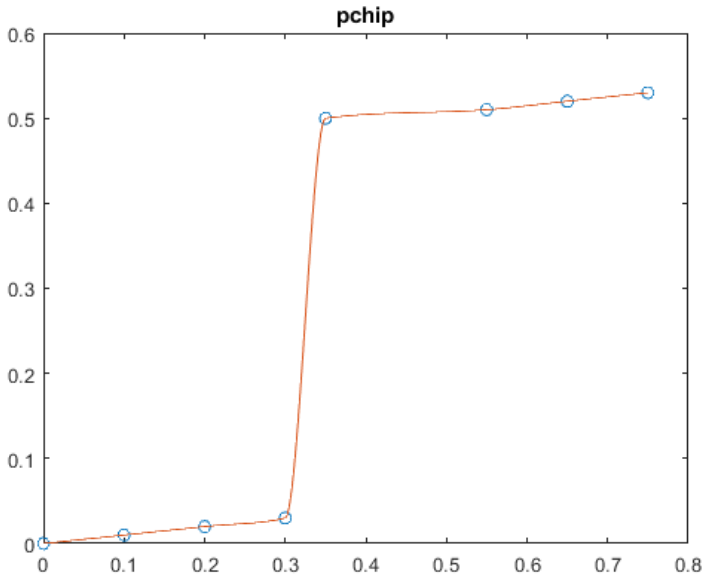

Figure 5: Hermite polynomial interpolation for data with a sudden bump using pchip in MATLAB.

If speed is important, the above nested formulas can undoubtedly be simplified to make he code more efficient.

The prescription in Fritsch[3] for producing cubic hermite interpolants requires the derivatives at the knots. It is not clear from the documentation which choice the current version of MATLAB makes in the pchip routine. However, I found looked through the MATLAB source code to see how this choice was made.

3.1. Derivatives

The formulas in the previous section require the computation of numerical derivatives at each knot.

3.1.1. Equally-Spaced Points

With equally-spaced knots, this is probably best done using the straightforward "three point formula."

$$y_i' \approx \frac{y_{i+1} - y_{i-1}}{2h}, \tag{47}$$

where $h = y_2 - y_1 = y_3 - y_2$. This formula is accurate to second order in h as can be seen from a Taylor series expansion of $f(x)$ about the point x_i.

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2}f''(x_i)(x - x_i)^2 + \frac{1}{6}f'''(x_i)(x - x_i)^3 \tag{48}$$

Substituting Equation 48 into Equation 47 with $y_{i+1} = f(x_i + h)$, $y_{i-1} = f(x_i - h)$ and $h = x_i - xi - 1 = x_{i+1} - x_i$ yields

$$y_i' = f'(x_i) = f'(x_i) + \frac{1}{6}f'''(x_i)h^2. \tag{49}$$

3.1.2. Unequally-Spaced Points

If the points are not equally spaced, a somewhat less accurate formula can be found from the average of the forward and backward difference formulas.

$$f'(x_i) \approx \frac{f_b'(x_i) + f_f'(x_i)}{2} \tag{50}$$

$$= \frac{y_i - y_{i-1}}{2h_b} + \frac{y_{i+1} - y_i)}{2h_f} \tag{51}$$

$$= \frac{y_{i+1}}{2h_f} + y_i\left(\frac{1}{2h_b} - \frac{1}{2h_f}\right) - \frac{y_{i-1}}{2h_b} \tag{52}$$

$$= \frac{y_{i+1}}{2h_f} + y_i\frac{h_f - h_b}{2h_f h_b} - \frac{y_{i-1}}{2h_b} \tag{53}$$

where $h_f = y_{i+1} - yi$ and $h_b = y_i - y_{i-1}$. Substituting the Equation 48 for the terms in Equation 52 shows the error in this formula.

$$\frac{f(x_{i+1})}{2h_f} = \frac{f(x_i)}{2h_f} + \frac{f'(x_i)}{2} + \frac{1}{4}h_f f''(x_i) + \cdots \tag{54}$$

$$\frac{f(x_{i-1})}{2h_b} = \frac{f(x_i)}{2h_b} - \frac{f'(x_i)}{2} + \frac{1}{4}h_b f''(x_i) + \cdots \tag{55}$$

$$\frac{f(x_{i+1})}{2h_f} + f(x_i)\left[\frac{1}{2h_b} - \frac{1}{2h_f}\right] - \frac{f(x_{i-1})}{2h_b} = f'(x_i) + \frac{1}{4}f''(x_i)(h_f - h_b) + \cdots \tag{56}$$

Thus, the error in this case in linear in h and proportional to $f''(x_i)$ instead of $f'''(x_i)$ as is the case for equally spaced points.

Another way to find a derivative is to find the quadratic polynomial which goes through the three points and take the derivative of that. Let

$$f(x) = a + b(x - x_i) + c(x - x_i)^2. \tag{57}$$

At the point $f(x_{i-1}) = y_{i-1}$

$$y_{i-1} = a - bh_b + ch_b^2. \tag{58}$$

At the point $f(x_i) = y_i$

$$y_i = a. \tag{59}$$

At the point $f(x_{i+1}) = y_{i+1}$

$$y_{i+1} = a + bh_f + ch_f^2. \tag{60}$$

Substituting Equation 59 into Equation 58 and Equation 59, solving the remaining equations for c and then equating them to solve for b yields

$$b = f'(x_i) = \frac{(y_i - y_{i-1})h_f}{h_b(h_f + h_b)} + \frac{(y_{i+1} - y_i)h_b}{h_f(h_f + h_b)}. \tag{61}$$

Figure 6 compares the cubic Hermite interpolations using Equation 53 and Equation 61 to the cubic spline interpolation. Note that both choices of slopes still result in overshooting on the interpolation curve, but not as sever as with the spline. The slope computed using Equation 53 has less overshoot than the one computed using Equation 61.

3.2. Piece-wise Monotonic Curves

Fritsch[3, 5] describe a way to make the piece-wise cubic curve locally monotonic. This should be the same as the pchip method illustrated in Figure 5.

The first step is to compute the linear slopes between the data points.

$$\Delta_i = \frac{y_{i+1} - y_i}{h_i} \tag{62}$$

$$h_i = x_{i+1} - x_i \tag{63}$$

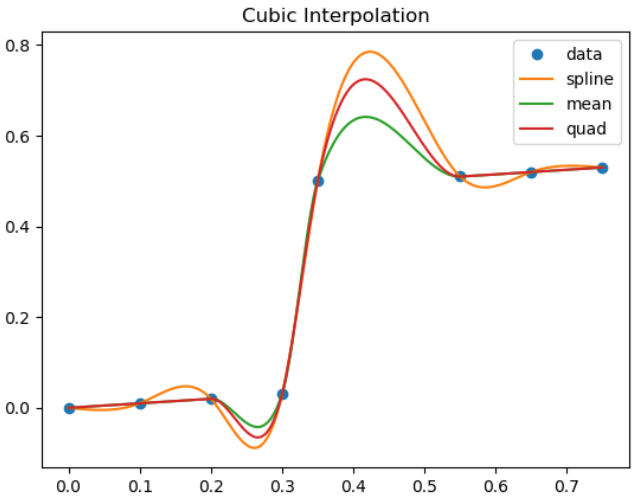

Figure 6: Comparison of spline and hermite polynomial interpolations for data with a discontinuity. The circles are the original data points. The curve labeled "mean" uses Equation 53 for the derivatives in the hermite interpolation. The curve labeled "quad" uses Equation 61 for the slopes.

The slopes at the data points are initially chosen to be the average of the linear slopes (as in Equation 50).

$$d_i = \frac{\Delta_{i-1} + \Delta_i}{2} \qquad \text{for } i = 2, \ldots, n-1 \tag{64}$$

$$d_1 = \Delta_1 \tag{65}$$

$$d_n = \Delta_n \tag{66}$$

If Δ_i and Δ_{i-1} have opposite signs, then $d_i = 0$. For $i = 1, \ldots, n-1$, if $\Delta_i = 0$

$$d_i = d_{i+1} = 0 \tag{67}$$

In this case, the following steps are ignored. The next step is to define the variables

$$\alpha_i = \frac{d_i}{\Delta_i} \tag{68}$$

$$\beta_i = \frac{d_{i+1}}{\Delta i} \tag{69}$$

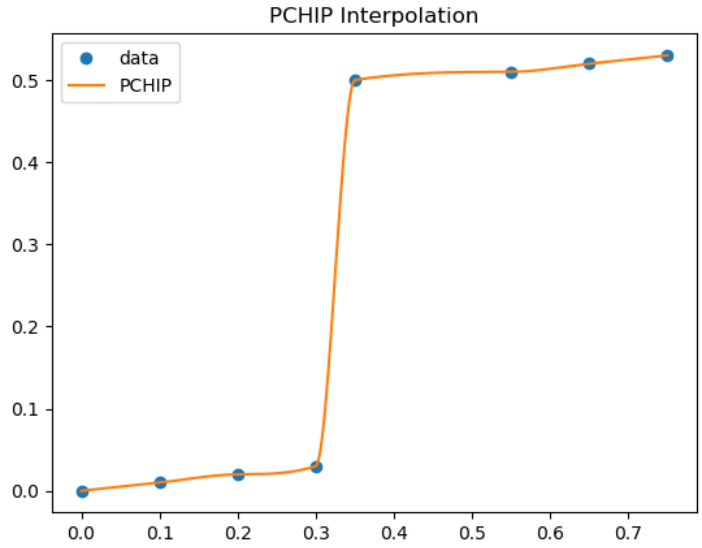

Figure 7: PCHIP interpolation using my Julia routine. Compare to the MATLAB calculation in Figure 5.

The vector (α_i, β_i) must have a radius less than 3. Therefore, if $\alpha_i^2 + \beta_i^2 > 9$,

$$\tau_i = \frac{3}{\sqrt{\alpha_i^2 + \beta_i^2}} \tag{70}$$

$$d_i = \tau_i \alpha_i \Delta_i \tag{71}$$

$$d_{i+1} = \tau_i \beta_i \Delta_i \tag{72}$$

Figure 7 is the interpolation using this algorithm for the slopes.

A. Spline Solution for Regularly-Spaced Points

The goal is to solve Equations 16 through 19 by eliminating the cubic coefficients and only having equations in terms of the y_i and D_i variables. We first need to add two more equations.

$$Y_i''(0) = 2c_i \tag{73}$$

$$Y_i''(1) = Y_{i+1}''(0) = 2c_i + 6d_i \tag{74}$$

$$c_{i+1} = c_i + 3d_i. \tag{75}$$

Substituting in the values for c_i from Equation 22 and d_i from Equation 23, Equation 75 becomes

$$3(y_{i+2} - y_{i+1}) - 2D_{i+1} - D_{i+2} = 3(y_{i+1} - y_i) - 2D_i - D_{i+1} + 3[2(y_i - y_{i+1}) + D_i + D_{i+1}]. \quad (76)$$

Grouping the y variables on one side of the equation and the D variables on the other,

$$3(y_{i+2} - y_i) = D_{i+2} + 4D_{i+1} + D_i. \quad (77)$$

This accounts for the middle rows of Equation 24. The top and bottom rows come from the initial and final conditions in Equations 7 and 8. Combining Equations 7, 73, and 22,

$$2c_1 = 0 \quad (78)$$

$$3(y_2 - y_1) = 2D_1 + D_2, \quad (79)$$

which is the first row of Equation 24. Combining Equations 8, 74, 22, and 23,

$$2c_n + 6d_n = 0 \quad (80)$$

$$3(y_{n+1} - y_n) - 2D_n - D_{n+1} + 3[2(y_n - y_{n+1}) + D_n + D_{n+1}] = 0 \quad (81)$$

$$3(y_{n+1} - y_n) = D_n + 2D_{n+1}, \quad (82)$$

which is the bottom row of Equation 24.

B. Spline Solution for Irregularly-Spaced Points

The goal is to solve Equations 16 through 19 by eliminating the cubic coefficients and only having equations in terms of the y_i, D_i, and α_i variables. We first need to add two more equations.

$$Y_i''(x_i) = 2c_i \quad (83)$$

$$Y_i''(x_{i+1}) = Y_{i+1}''(x_{i+1}) = 2c_i + 6d_i\alpha_i \quad (84)$$

$$c_{i+1} = c_i + 3d_i\alpha_i. \quad (85)$$

Substituting in the values for c_i from Equation 32 and d_i from Equation 33, Equation 85 becomes

$$3\frac{y_{i+2} - y_{i+1}}{\alpha_{i+1}^2} - 2\frac{D_{i+1}}{\alpha_i} - \frac{D_{i+2}}{\alpha_{i+1}} = 3\frac{y_{i+1} - y_i}{\alpha_i^2} - 2\frac{D_i}{\alpha_i} - \frac{D_{i+1}}{\alpha_i} + 3\left[2\frac{y_i - y_{i+1}}{\alpha_i^2} + \frac{D_i}{\alpha_i} + \frac{D_{i+1}}{\alpha_i}\right]. \quad (86)$$

Grouping the y variables on one side of the equation and the D variables on the other,

$$3\left[\frac{y_{i+2}}{\alpha_{i+1}^2} + y_{i+1}\left(\frac{1}{\alpha_i^2} - \frac{1}{\alpha_{i+1}^2}\right) - \frac{y_i}{\alpha_i^2}\right] = \frac{D_{i+2}}{\alpha_{i+1}} + 2D_{i+1}\left(\frac{1}{\alpha_i} + \frac{1}{\alpha_{i+1}}\right) + \frac{D_i}{\alpha_i}. \quad (87)$$

This accounts for the middle rows of Equation 34. The top and bottom rows come from the initial and final conditions in Equations 7 and 8. Combining Equations 7, 83, and 32,

$$2c_1 = 0 \tag{88}$$

$$3\frac{(y_2 - y_1)}{\alpha_1^2} = 2\frac{D_1}{\alpha_1} + \frac{D_2}{\alpha_1}, \tag{89}$$

which is the first row of Equation 34. Combining Equations 8, 84, 32, and 33,

$$2c_n + 6d_n\alpha_n = 0 \tag{90}$$

$$3\frac{y_{n+1} - y_n}{\alpha_n^2} - 2\frac{D_n}{\alpha_n} - \frac{D_{n+1}}{\alpha_n} + 3\left[2\frac{y_n - y_{n+1}}{\alpha_n^2} + \frac{D_n}{\alpha_n} + \frac{D_{n+1}}{\alpha_n}\right] = 0 \tag{91}$$

$$3\frac{y_n - y_{n+1}}{\alpha_n^2} = \frac{D_n}{\alpha_n} + 2\frac{D_{n+1}}{\alpha_n}, \tag{92}$$

which is the bottom row of Equation 24.

C. Julia Code

Here is the Julia 1.4 code I used to do the calculations in this article.

C.1. Interp.jl

This is the module which does the actual calculations.

```
module Interp
# Code for interpolation for various orders
using LinearAlgebra
using Statistics
import Base.length

export CubicSpline, interp, slope, slope2, pchip, pchip2, pchip3

const eps = 1e-3 # rel error allowed on extrapolation

"""
    CubicSpline(x,a,b,c,d)

concrete type for holding the data needed
    to do a cubic spline interpolation
"""
abstract type AbstractSpline end

struct CubicSpline <: AbstractSpline
    x::Union{Array{Float64,1},
        StepRangeLen{Float64,
            Base.TwicePrecision{Float64},
            Base.TwicePrecision{Float64}}}
    a::Array{Float64,1}
    b::Array{Float64,1}
```

```
    c::Array{Float64,1}
    d::Array{Float64,1}
    alphabar::Float64
end

struct ComplexSpline <: AbstractSpline
    x::Union{Array{Float64,1},
        StepRangeLen{Float64,
            Base.TwicePrecision{Float64},
            Base.TwicePrecision{Float64}}}
    a::Array{Complex{Float64},1}
    b::Array{Complex{Float64},1}
    c::Array{Complex{Float64},1}
    d::Array{Complex{Float64},1}
    alphabar::Float64
end

"""
    PCHIP(x,a,b,c,d)

concrete type for holding the data needed
    to do a piecewise continuous hermite interpolation
"""
struct PCHIP
    x::Union{Array{Float64,1},
        StepRangeLen{Float64,
            Base.TwicePrecision{Float64},
            Base.TwicePrecision{Float64}}}
    y::Array{Float64,1}
    d::Array{Float64,1}
    h::Array{Float64,1}
end

"""
    CubicSpline(x,y)

Creates the CubicSpline structure needed for cubic spline
interpolation

# Arguments
- `x`: an array of x values at which the function is known
- `y`: an array of y values corresponding to these x values
"""
function CubicSpline(x::Array{Float64,1}, y::Array{Float64,1})
    len = size(x,1)
    if len<3
        error("CubicSpline requires at least three points for interpolation")
    end
    # Pre-allocate and fill columns and diagonals
    yy = zeros(typeof(x[1]),len)
    du = zeros(typeof(x[1]),len-1)
    dd = zeros(typeof(x[1]),len)
    # Scale x so that the alpha values are better
    alpha = x[2:len].-x[1:len-1]
    alphabar = Statistics.mean(alpha)
    alpha = alpha/alphabar
    yy[1] = 3*(y[2]-y[1])/alpha[1]^2
    du = 1 ./alpha
    dd[1] = 2/alpha[1]
    yy[2:len-1] = 3*(y[3:len]./alpha[2:len-1].^2
        .+y[2:len-1].*(alpha[1:len-2].^(-2).-alpha[2:len-1].^(-2))
        .-y[1:len-2]./alpha[1:len-2].^2)
```

```
    dd[2:len-1] = 2*(1 ./alpha[1:len-2] .+ 1 ./alpha[2:len-1])
    yy[len] = 3*(y[len]-y[len-1])/alpha[len-1]^2
    dd[len] = 2/alpha[len-1]
    # Solve the tridiagonal system for the derivatives D
    dm = Tridiagonal(du,dd,du)
    D = dm\yy
    # fill the arrays of spline coefficients
    a = y[1:len-1]  # silly but makes the code more transparent
    b = D[1:len-1]  # ditto
    c = 3 .*(y[2:len].-y[1:len-1])./alpha[1:len-1].^2 .-
        2*D[1:len-1]./alpha[1:len-1].-D[2:len]./alpha[1:len-1]
    d = 2 .*(y[1:len-1].-y[2:len])./alpha[1:len-1].^3 .+
        D[1:len-1]./alpha[1:len-1].^2 .+
        D[2:len]./alpha[1:len-1].^2
    CubicSpline(x, a, b, c, d, alphabar)
end

function CubicSpline(x::Array{Float64,1}, y::Array{Complex{Float64},1})
    len = size(x,1)
    if len<3
        error("CubicSpline requires at least three points for interpolation")
    end
    # Pre-allocate and fill columns and diagonals
    yy = zeros(typeof(y[1]),len)
    du = zeros(typeof(x[1]),len-1)
    dd = zeros(typeof(x[1]),len)
    alpha = x[2:len].-x[1:len-1]
    alphabar = Statistics.mean(alpha)
    alpha = alpha/alphabar
    yy[1] = 3*(y[2]-y[1])/alpha[1]^2
    du = 1 ./alpha
    dd[1] = 2/alpha[1]
    yy[2:len-1] = 3*(y[3:len]./alpha[2:len-1].^2
        .+y[2:len-1].*(alpha[1:len-2].^(-2).-alpha[2:len-1].^(-2))
        .-y[1:len-2]./alpha[1:len-2].^2)
    dd[2:len-1] = 2*(1 ./alpha[1:len-2] .+ 1 ./alpha[2:len-1])
    yy[len] = 3*(y[len]-y[len-1])/alpha[len-1]^2
    dd[len] = 2/alpha[len-1]
    # Solve the tridiagonal system for the derivatives D
    dm = Tridiagonal(du,dd,du)
    D = dm\yy
    # fill the arrays of spline coefficients
    a = y[1:len-1]  # silly but makes the code more transparent
    b = D[1:len-1]  # ditto
    c = 3 .*(y[2:len].-y[1:len-1])./alpha[1:len-1].^2 .-
        2*D[1:len-1]./alpha[1:len-1].-D[2:len]./alpha[1:len-1]
    d = 2 .*(y[1:len-1].-y[2:len])./alpha[1:len-1].^3 .+
        D[1:len-1]./alpha[1:len-1].^2 .+
        D[2:len]./alpha[1:len-1].^2
    ComplexSpline(x, a, b, c, d, alphabar)
end

function CubicSpline(x::StepRangeLen{Float64,Base.TwicePrecision{Float64},
    Base.TwicePrecision{Float64}}, y::Array{Float64,1})
    len = length(x)
    if len<3
        error("CubicSpline requires at least three points for interpolation")
    end
    # Pre-allocate and fill columns and diagonals
    yy = zeros(len)
    dl = ones(len-1)
    dd = 4.0 .* ones(len)
```

```
    dd[1] = 2.0
    dd[len] = 2.0
    yy[1] = 3*(y[2]-y[1])
    yy[2:len-1] = 3*(y[3:len].-y[1:len-2])
    yy[len] = 3*(y[len]-y[len-1])
    # Solve the tridiagonal system for the derivatives D
    dm = Tridiagonal(dl,dd,dl)
    D = dm\yy
    # fill the arrays of spline coefficients
    a = y[1:len-1]  # silly but makes the code more transparent
    b = D[1:len-1]  # ditto
    c = 3 .*(y[2:len].-y[1:len-1]).-2*D[1:len-1].-D[2:len]
    d = 2 .*(y[1:len-1].-y[2:len]).+D[1:len-1].+D[2:len]
    alpha = step(x);
    CubicSpline(x, a, b, c, d, alpha)
end

function CubicSpline(x::StepRangeLen{Float64,Base.TwicePrecision{Float64},
    Base.TwicePrecision{Float64}}, y::Array{Complex{Float64},1})
    len = length(x)
    if len<3
        error("CubicSpline requires at least three points for interpolation")
    end
    # Pre-allocate and fill columns and diagonals
    yy = zeros(Complex{Float64}, len)
    dl = ones(len-1)
    dd = 4.0 .* ones(len)
    dd[1] = 2.0
    dd[len] = 2.0
    yy[1] = 3*(y[2]-y[1])
    yy[2:len-1] = 3*(y[3:len].-y[1:len-2])
    yy[len] = 3*(y[len]-y[len-1])
    # Solve the tridiagonal system for the derivatives D
    dm = Tridiagonal(dl,dd,dl)
    D = dm\yy
    # fill the arrays of spline coefficients
    a = y[1:len-1]  # silly but makes the code more transparent
    b = D[1:len-1]  # ditto
    c = 3 .*(y[2:len].-y[1:len-1]).-2*D[1:len-1].-D[2:len]
    d = 2 .*(y[1:len-1].-y[2:len]).+D[1:len-1].+D[2:len]
    alpha = x.step;
    ComplexSpline(x, a, b, c, d)
end

# This version of pchip uses the mean value of the slopes
# between data points on either side of the interpolation point
"""
    pchip(x,y)

Creates the PCHIP structure needed for piecewise
    continuous cubic spline interpolation

# Arguments
- `x`: an array of x values at which the function is known
- `y`: an array of y values corresonding to these x values
"""
function pchip(x::Array{Float64,1}, y::Array{Float64,1})
    len = size(x,1)
    if len<3
        error("PCHIP requires at least three points for interpolation")
    end
    h = x[2:len].-x[1:len-1]
```

```
    # Pre-allocate and fill columns and diagonals
    d = zeros(len)
    d[1] = (y[2]-y[1])/h[1]
    for i=2:len-1
        d[i] = (y[i+1]/h[i]+y[i]*(1/h[i-1]-1/h[i])-y[i-1]/h[i-1])/2
    end
    d[len] = (y[len]-y[len-1])/h[len-1]
    PCHIP(x,y,d,h)
end

# PCHIP with quadratic fit to determine slopes
function pchip2(x::Array{Float64,1}, y::Array{Float64,1})
    len = size(x,1)
    if len<3
        error("PCHIP requires at least three points for interpolation")
    end
    h = x[2:len].-x[1:len-1]
    # Pre-allocate and fill columns and diagonals
    d = zeros(len)
    d[1] = (y[2]-y[1])/h[1]
    for i=2:len-1
        d[i] = (y[i]-y[i-1])*h[i]/(h[i-1]*(h[i-1]+h[i])) +
            (y[i+1]-y[i])*h[i-1]/(h[i]*(h[i-1]+h[i]))
    end
    d[len] = (y[len]-y[len-1])/h[len-1]
    PCHIP(x,y,d,h)
end

# Real PCHIP
function pchip3(x::Array{Float64,1}, y::Array{Float64,1})
    len = size(x,1)
    if len<3
        error("PCHIP requires at least three points for interpolation")
    end
    for i = 2:length(x)
        if x[i] <= x[i-1]
            error("pchip3: array of x values is not monotonic at x = $(x[i+1])")
        end
    end
    h = x[2:len].-x[1:len-1]
    # test for monotonicty
    del = (y[2:len].-y[1:len-1])./h
    # Pre-allocate and fill columns and diagonals
    d = zeros(len)
    d[1] = del[1]
    for i=2:len-1
        if del[i]*del[i-1] < 0
            d[i] = 0
        else
            d[i] = (del[i]+del[i-1])/2
        end
    end
    d[len] = del[len-1]
    for i=1:len-1
        if del[i] == 0
            d[i] = 0
            d[i+1] = 0
        else
            alpha = d[i]/del[i]
            beta = d[i+1]/del[i]
            if alpha^2+beta^2 > 9
                tau = 3/sqrt(alpha^2+beta^2)
```

```
                d[i] = tau*alpha*del[i]
                d[i+1] = tau*beta*del[i]
            end
        end
    end
    PCHIP(x,y,d,h)
end

"""
    interp(cs::CubicSpline, v::Float)

Interpolate to the value corresonding to v

# Examples
```
x = cumsum(rand(10))
y = cos.(x);
cs = CubicSpline(x,y)
v = interp(cs, 1.2)
```
"""
function interp(cs::AbstractSpline, v::Float64)
    # Find v in the array of x's
    if (v<cs.x[1]) | (v>cs.x[length(cs.x)])
        error("Extrapolation not allowed")
    end
    segment = region(cs.x, v)
    if cs.x isa StepRangeLen
        # regularly spaced points
        t = (v-cs.x[segment])/step(cs.x)
    else
        # irregularly spaced points
        t = (v-cs.x[segment])/cs.alphabar
    end
    cs.a[segment] + t*(cs.b[segment] + t*(cs.c[segment] + t*cs.d[segment]))
end
# alias
(cs::AbstractSpline)(v::Float64) = interp(cs,v)

function interp(pc::PCHIP, v::Float64)

    if v*(1+eps)<first(pc.x)
        error("Extrapolation not allowed, $v<$(first(pc.x))")
    end
    if v*(1-eps)>last(pc.x)
        error("Extrapolation not allowed, $v>$(last(pc.x))")
    end
    i = region(pc.x, v)
    phi(t) = 3*t^2 - 2*t^3
    psi(t) = t^3 - t^2
    H1(x) = phi((pc.x[i+1]-v)/pc.h[i])
    H2(x) = phi((v-pc.x[i])/pc.h[i])
    H3(x) = -pc.h[i]*psi((pc.x[i+1]-v)/pc.h[i])
    H4(x) = pc.h[i]*psi((v-pc.x[i])/pc.h[i])
    yv = pc.y[i]*H1(v) + pc.y[i+1]*H2(v) + pc.d[i]*H3(v) + pc.d[i+1]*H4(v)
    # For reasons I have yet to understand completely, this can blow
    # up sometimes. Revert to linear interpolation in this case
    if isnan(yv)
        h = pc.x[i+1] - pc.x[i]
        if h > 0
            t = (pc.x[i+1]-v)/h
        else
```

```
            error("interp data is not monotonic, x[i] = $(x[i]), x[i+1]=$(x[i+1])")
        end
        println("warning, reverting to linear interpolation")
        yv = pc.y[i] + t*(pc.y[i+1]-pc.y[i])
    end
    yv
end
#alias
(pc::PCHIP)(v::Float64) = interp(pc,v)

"""
    slope(cs::CubicSpline, v::Float)

Derivative at the point corresonding to v

# Examples
```
x = cumsum(rand(10))
y = cos.(x);
cs = CubicSpline(x,y)
v = slope(cs, 1.2)
```
"""
function slope(cs::CubicSpline, v::Float64)
    # Find v in the array of x's
    if (v<cs.x[1]) | (v>cs.x[length(cs.x)])
        error("Extrapolation not allowed")
    end
    segment = region(cs.x, v)
    if cs.x isa StepRangeLen
        # regularly spaced points
        t = (v-cs.x[segment])/step(cs.x)
    else
        # irregularly spaced points
        t = (v-cs.x[segment])/cs.alphabar
    end
    cs.b[segment] + t*(2*cs.c[segment] + t*3*cs.d[segment])
end

"""
    slope(pc::PCHIP, v::Float)

Derivative at the point corresponding to v

# Examples
```
x = cumsum(rand(10))
y = cos.(x);
pc = pchip(x,y)
v = slope(pc, 1.2)
```
"""
function slope(pc::PCHIP, v::Float64)
    # Find v in the array of x's
    if (v<pc.x[1]) | (v>pc.x[length(pc.x)])
        error("Extrapolation not allowed")
    end
    i = region(pc.x, v)
    phip(t) = 6*t - 6*t^2
    psip(t) = 3*t^2 - 2*t
    H1p(x) = -phip((pc.x[i+1]-v)/pc.h[i])/pc.h[i]
    H2p(x) = phip((v-pc.x[i])/pc.h[i])/pc.h[i]
```

```
    H3p(x) = psip((pc.x[i+1]-v)/pc.h[i])
    H4p(x) = psip((v-pc.x[i])/pc.h[i])
    pc.y[i]*H1p(v) + pc.y[i+1]*H2p(v) + pc.d[i]*H3p(v) + pc.d[i+1]*H4p(v)
end

"""
    slope2(cs::CubicSpline, v::Float)

Second derivative at the point corresponding to v

# Examples
```
x = cumsum(rand(10))
y = cos.(x);
cs = CubicSpline(x,y)
v = slope2(cs, 1.2)
```
"""
function slope2(cs::CubicSpline, v::Float64)
    # Find v in the array of x's
    if (v<cs.x[1]) | (v>cs.x[length(cs.x)])
        error("Extrapolation not allowed")
    end
    segment = region(cs.x, v)
    if cs.x isa StepRangeLen
        # regularly spaced points
        t = (v-cs.x[segment])/step(cs.x)
    else
        # irregularly spaced points
        t = (v-cs.x[segment])/cs.alphabar
    end
    2*cs.c[segment] + 6*t*cs.d[segment]
end

function region(x::AbstractArray, v::Float64)
    # Binary search
    len = size(x,1)
    li = 1
    ui = len
    mi = div(li+ui,2)
    done = false
    while !done
        if v<x[mi]
            ui = mi
            mi = div(li+ui,2)
        elseif v>x[mi+1]
            li = mi
            mi = div(li+ui,2)
        else
            done = true
        end
        if mi == li
            done = true
        end
    end
    mi
end

function region(x::StepRangeLen{Float64,Base.TwicePrecision{Float64},
    Base.TwicePrecision{Float64}}, y::Float64)
    min(floor(Int,(y-first(x))/step(x)),length(x)-2) + 1
end
```

```
end # module Interp
```

C.2. TestInterp.jl

This is the code which does the unit testing.

```
using Test
try
    using Interp
catch
    push!(LOAD_PATH, pwd())
    using Interp
end
import Interp.region

const unitTests = true
const graphicsTests = false
const bumpTests = false

if graphicsTests | bumpTests
    import PyPlot # needed if graphicsTests is true
    const plt = PyPlot
end

function regular_tests()
    @testset "regular interpolation" begin
    # Test not enough points exception
    x = range(1.0, stop=2.0, length=2)
    y = [2.0, 4.0]
    @test_throws ErrorException CubicSpline(x,y)
    x = range(1.0, stop=3.25, length=4)
    y = [1.5, 3.0, 3.7, 2.5]
    cs = CubicSpline(x,y)
    @test_throws ErrorException interp(cs, 0.0)
    @test_throws ErrorException interp(cs, 4.0)
    # Check region
    @test region(x, 1.0) == 1
    @test region(x, 1.2) == 1
    @test region(x, 3.25) == 3
    @test region(x, 2.0) == 2
    @test region(x, 2.8) == 3
    # Check spline at knots
    @test interp(cs, 1.0) == 1.5
    @test interp(cs, 1.75) == 3.0
    @test isapprox(interp(cs, 3.25), 2.5, atol=1e-14)
    # Check spline with unit spacing of knots
    len = 5
    x = range(0.0, stop=4.0, length=len)
    y = sin.(x)
    cs = CubicSpline(x,y)
    dy = cos.(x)
    for i = 1:len-1
        @test cs.a[i] == y[i]
        @test isapprox(cs.a[i] + cs.b[i] + cs.c[i] + cs.d[i], y[i+1], atol=1.e-12)
        @test isapprox(cs.b[i], dy[i], atol=0.08)
    end
    for i=1:len-2
```

```
        @test isapprox(cs.b[i] + 2*cs.c[i] + 3*cs.d[i], dy[i+1], atol=0.08)
    end
    # Check second derivatives at end points
    @test isapprox(0., cs.c[1], atol=1e-14);
    @test isapprox(0., cs.c[len-1]+3*cs.d[len-1], atol=1e-14);
    end;
end

function irr_coef_tests()
    @testset "irregular interpolation coefficients test" begin
    x = [0.2, 1.4, 3.8, 5.7]
    y = [1.5, 3.0, 3.7, 2.5]
    n = length(x)
    csi = CubicSpline(x,y)
    alpha = (x[2:n] - x[1:n-1])/csi.alphabar
    for i = 1:length(x)-1
        ap = csi.a[i]+csi.b[i]*alpha[i]+csi.c[i]*alpha[i]^2+csi.d[i]*alpha[i]^3
        @test isapprox(ap,y[i+1])
    end
    for i = 1:length(x)-2
        bp = csi.b[i] + 2*csi.c[i]*alpha[i] + 3*csi.d[i]*alpha[i]^2
        @test isapprox(bp,csi.b[i+1])
    end
    end;
end


function irregular_tests()
    @testset "irregular interpolation" begin
    # Test not enough points exception
    x = [1.0, 2.0]
    y = [2.0, 4.0]
    @test_throws ErrorException CubicSpline(x,y)
    x = [0.2, 1.4, 3.8, 5.7]
    y = [1.5, 3.0, 3.7, 2.5]
    csi = CubicSpline(x,y)
    @test_throws ErrorException interp(csi, 0.0)
    @test_throws ErrorException interp(csi, 6.0)
    # Check region
    @test region(x, 0.3) == 1
    @test region(x, 0.2) == 1
    @test region(x, 5.7) == 3
    @test region(x, 2.1) == 2
    @test region(x, 4.0) == 3
    # Check spline at knots
    @test interp(csi, 0.2) == 1.5
    @test interp(csi, 1.4) == 3.0
    @test isapprox(interp(csi, 5.7), 2.5, atol=1e-14)
    # Check spline with unit spacing of knots
    x = range(0.,4.,step=1.)
    y = sin.(x)
    cs = CubicSpline(x,y)
    csi = CubicSpline(collect(x),y)
    for i = 1:4
        @test csi.a[i] == cs.a[i]
        @test csi.b[i] == cs.b[i]
        @test csi.c[i] == cs.c[i]
        @test csi.d[i] == cs.d[i]
        @test csi.a[i] == y[i]
        @test isapprox(csi.a[i] + csi.b[i] + csi.c[i] + csi.d[i], y[i+1], atol=1.e-12)
    end
    # Check meeting knot conditions
```

```
    for i = 1:3
        di = csi.b[i+1]
        dip = csi.b[i] + 2*csi.c[i] + 3*csi.d[i]
        @test isapprox(di, dip, atol=1.e-12)
    end
    for i = 1:3
        ddi = 2*csi.c[i+1]
        ddip = 2*csi.c[i]+6*csi.d[i]
        @test isapprox(ddi, ddip, atol=1.e-12)
    end
    # Second derivatives at end points
    @test isapprox(csi.c[1], 0.0, atol = 1.e-12)
    @test isapprox(2*csi.c[4]+6*csi.d[4], 0.0, atol = 1.e-12)
    # Test matching boundary conditions with unequally spaced knots
    x = [0.0, 0.7, 2.3, 3.0, 4.1]
    y = sin.(x)
    csi = CubicSpline(x,y)
    for i = 1:4
        @test csi.a[i] == y[i]
        alpha = x[i+1]-x[i]
        yend = csi.a[i] + csi.b[i]*alpha + csi.c[i]*alpha^2
         + csi.d[i]*alpha^3
        # @test isapprox(yend, y[i+1], atol=1.e-12)
    end
    # Check for continuity near knot 2
    eps = 0.0001
    vl = x[2] - eps
    vg = x[2] + eps
    yl = interp(csi, vl)
    yg = interp(csi, vg)
    @test abs(yl-yg) < 2*eps
    sl = slope(csi, vl)
    sg = slope(csi, vg)
    @test abs(sl-sg) < 2*eps
    sl2 = slope2(csi, vl)
    sg2 = slope2(csi, vg)
    @test abs(sl2-sg2) < 2*eps
    # Check meeting knot conditions
    for i = 1:3
        alpha = (x[i+1]-x[i])/csi.alphabar
        dip = csi.b[i+1]
        di = csi.b[i]+2*csi.c[i]*alpha+3*csi.d[i]*alpha^2
        @test isapprox(di, dip, atol=1.e-12)
    end
    for i = 1:3
        alpha = (x[i+1]-x[i])/csi.alphabar
        ddi = 2*csi.c[i+1]
        ddip = 2*csi.c[i]+6*csi.d[i]*alpha
        @test isapprox(ddi, ddip, atol=1.e-12)
    end
    # Second derivatives at end points
    @test isapprox(csi.c[1], 0.0, atol = 1.e-12)
    alpha = (x[5] - x[4])/csi.alphabar
    @test isapprox(2*csi.c[4]+6*csi.d[4]*alpha, 0.0, atol = 1.e-12)
    end;
end

function graphics_tests()
    x = range(0.0, stop=pi, length=10)
    y = sin.(x)
    cs = CubicSpline(x,y)
    xx = range(0.0, stop=pi, length=97)
```

```
    yy = [interp(cs,v) for v in xx]
    yyy = sin.(xx)
    plt.figure()
    plt.plot(x,y,"o",xx,yy,"-",xx,yyy,".")
    plt.title("Regular Interpolation")
    plt.show()

    x = cumsum(rand(10));
    x = (x.-x[1]).*pi/(x[10].-x[1])
    y = sin.(x)
    cs = CubicSpline(x,y)
    xx = range(0.0, stop=pi, length=97)
    yy = [interp(cs,v) for v in xx]
    yyy = sin.(xx)
    plt.figure()
    plt.plot(x,y,"o",xx,yy,"-",xx,yyy,".")
    plt.title("Irregular Interpolation, 10 Points")
    plt.show()
end

function bump_tests()
    x = [0.0, 0.1, 0.2, 0.3, 0.35, 0.55, 0.65, 0.75];
    y = [0.0, 0.01, 0.02, 0.03, 0.5, 0.51, 0.52, 0.53];
    xx = range(0.0,stop=0.75,length=400);
    sp = CubicSpline(x,y);
    yy = [interp(sp, v) for v in xx]
    pc = pchip(x,y)
    yyy = [interp(pc,v) for v in xx]
    pc2 = pchip2(x,y)
    yyy2 = [interp(pc2,v) for v in xx]
    plt.figure()
    plt.plot(x,y,"o",xx,yy,"-",xx,yyy,"-",xx,yyy2,"-")
    plt.title("Cubic Interpolation")
    plt.legend(("data", "spline", "mean","quad"))
    pc3 = pchip3(x,y)
    yyy3 = [interp(pc3,v) for v in xx]
    plt.figure()
    plt.plot(x, y, "o", xx, yyy3, "-")
    plt.title("PCHIP Interpolation")
    plt.legend(("data", "PCHIP"))
    plt.show()
end

function regular_pchip_tests()
    @testset "Regular pchip" begin
    end;
end

function irregular_pchip_tests()
    x=[1.0, 1.8, 2.5, 3.0, 3.9];
    y=cos.(x);
    pc=pchip(x,y)
    @testset "Irregular pchip" begin
    for i=1:5
        # Continuity
        @test interp(pc,x[i])==y[i]
    end
    for i = 2:4
        # Continuity of slope
        eps = 0.000001
        @test isapprox(slope(pc,x[i]-eps),slope(pc,x[i]+eps),atol=4*eps)
    end
```

```
    end;
end

if unitTests
    regular_tests()
    irr_coef_tests()
    irregular_tests()
    # regular_pchip_tests()
    # irregular_pchip_tests()
end

if graphicsTests
    graphics_tests()
end

if bumpTests
    bump_tests()
end
```

References

[1] Weisstein, Eric W. “Cubic Spline." From *MathWorld*–A Wolfram Web Resource. `http://mathworld.wolfram.com/CubicSpline.html` (accessed 8/17/2018). Citing: Bartels, R. H.; Beatty, J. C.; and Barsky, B. A. "Hermite and Cubic Spline Interpolation," Ch. 3 in *An Introduction to Splines for Use in Computer Graphics and Geometric Modelling*. San Francisco, CA: Morgan Kaufmann, pp. 9–17, 1998.

[2] Dierckx, Paul, “Curve and Surface Fitting with Splines,” Oxford Science Publications, Clarendon Press, 1993.

[3] Fritsch, F. N. and R. E. Carlson. “Monotone Piecewise Cubic Interpolation." SIAM Journal on Numerical Analysis. Vol. 17, 1980, pp.238-–246.

[4] Kahaner, David, Cleve Moler, Stephen Nash. Numerical Methods and Software. Upper Saddle River, NJ: Prentice Hall, 1988.

[5] Wikipedia, “Monotonic cubic interpolation," `https://en.wikipedia.org/wiki/Monotone\_cubic\_interpolation` (accessed 23 Aug 2018).