Numerical Analysis and Computing

Lecture Notes #06

— Interpolation and Polynomial Approximation —

Piecewise Polynomial Approximation; Cubic Splines

Joe Mahaffy,
⟨mahaffy@math.sdsu.edu⟩

Department of Mathematics
Dynamical Systems Group
Computational Sciences Research Center
San Diego State University
San Diego, CA 92182-7720

http://www-rohan.sdsu.edu/∼jmahaffy

Spring 2010

Outline

Checking the Roadmap

Inspired by Weierstrass, we have looked at a number of strategies for approximating arbitrary functions using polynomials.

Taylor	Detailed information from one point, excellent locally, but not very successful for extended intervals.
Lagrange	$\leq n$th degree poly. interpolating the function in $(n+1)$ pts. **Representation:** Theoretical using the Lagrange coefficients $L_{n,k}(x)$; pointwise using Neville's method; and more useful/general using Newton's divided differences.
Hermite	$\leq (2n+1)$th degree polynomial interpolating the function, and matching its first derivative in $(n+1)$ points. **Representation:** Theoretical using two types of Hermite coefficients $H_{n,k}(x)$, and $\widehat{H}_{n,k}(x)$; and more useful/general using a modification of Newton's divided differences.

With $(n+1)$ points, and a uniform matching criteria of m derivatives in each point we can talk these in terms of the broader class of **osculating polynomials** with:

Taylor(m,n=0), Lagrange(m=0,n), Hermite(m=1,n); with resulting degree $d \leq (m+1)(n+1)-1$.

Admiring the Roadmap... Are We Done?

We even figured out how to modify Newton's divided differences to produce representations of arbitrary osculating polynomials...

We have swept a dirty little secret under the rug: —

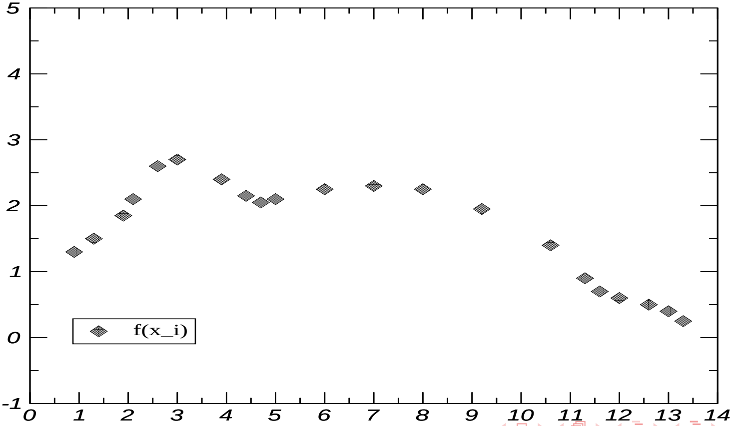
For all these interpolation strategies we get — provided the underlying function is smooth enough, *i.e.* $f \in C^{(m+1)(n+1)}([a,b])$ — errors of the form

$$\underbrace{\frac{\prod_{i=0}^{n}(x-x_i)^{(m+1)}}{((m+1)(n+1))!}}_{\eta(x)} f^{((m+1)(n+1))}(\xi(x)), \quad \xi(x) \in [a,b]$$

We have seen that with the x_i's dispersed (Lagrange / Hermite-style), the controllable part, $\eta(x)$, of the error term is better behaved than for Taylor polynomials. **However,** we have no control over the $((n+1)(m+1))$th derivative of f.

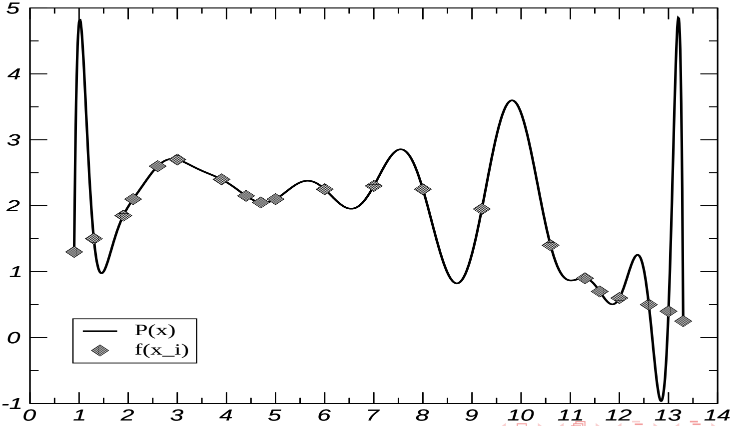
Problems with High Order Polynomial Approximation

We can force a polynomial of high degree to pass through as many points $(x_i, f(x_i))$ as we like. However, high degree polynomials tend to fluctuate wildly **between** the interpolating points.



Problems with High Order Polynomial Approximation

We can force a polynomial of high degree to pass through as many points $(x_i, f(x_i))$ as we like. However, high degree polynomials tend to fluctuate wildly **between** the interpolating points.



Alternative Approach to Interpolation

The oscillations tend to be extremely bad close to the **end points** of the interval of interest, and (in general) the more points you put in, the wilder the oscillations get!

Clearly, we need some new tricks!

Idea: Divide the interval into smaller sub-intervals, and construct different low degree polynomial approximations (with small oscillations) on the sub-intervals.

This is called **Piecewise Polynomial Approximation**.

Simplest continuous variant: **Piecewise Linear Approximation**:

Piecewise Linear Approximation

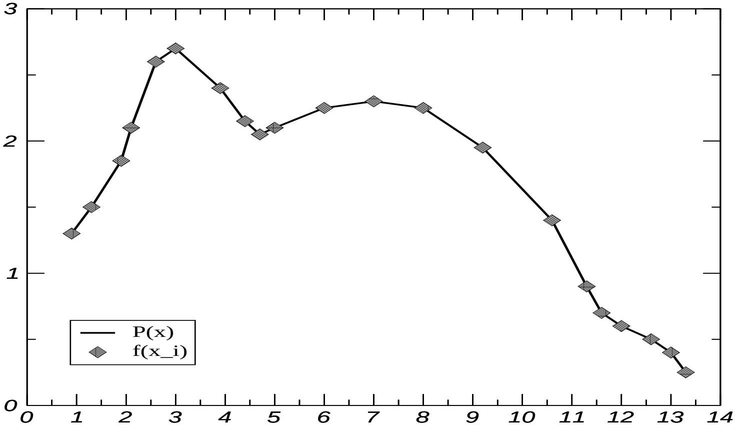


Figure: Piecewise linear approximation of the same data as on slide 5. Is this the end of excessive oscillations?!?

Problem with Piecewise Linear Approximation

The piecewise linear interpolating function is **not differentiable** at the **"nodes,"** *i.e.* the points x_i. (Typically we want to do more than just plot the polynomial... and even plotting shows sharp corners!)

Idea: Strengthened by our experience with Hermite polynomials, why not generate piecewise polynomials that match both the function value and some number of derivatives in the nodes!

Problem with Piecewise Linear Approximation

The piecewise linear interpolating function is **not differentiable** at the **"nodes,"** *i.e.* the points x_i. (Typically we want to do more than just plot the polynomial... and even plotting shows sharp corners!)

Idea: Strengthened by our experience with Hermite polynomials, why not generate piecewise polynomials that match both the function value and some number of derivatives in the nodes!

The Return of the Cubic Hermite Polynomial!

If, for instance $f(x)$ and $f'(x)$ are known in the nodes, we can use a collection of **cubic Hermite polynomials** $H_j^3(x)$ to build up such a function.

Problem with Piecewise Linear Approximation

The piecewise linear interpolating function is **not differentiable** at the **"nodes,"** *i.e.* the points x_i. (Typically we want to do more than just plot the polynomial... and even plotting shows sharp corners!)

Idea: Strengthened by our experience with Hermite polynomials, why not generate piecewise polynomials that match both the function value and some number of derivatives in the nodes!

The Return of the Cubic Hermite Polynomial!

If, for instance $f(x)$ and $f'(x)$ are known in the nodes, we can use a collection of **cubic Hermite polynomials** $H_j^3(x)$ to build up such a function.

But... what if $f'(x)$ is **not** known (in general getting measurements of the derivative of a physical process is much more difficult and unreliable than measuring the quantity itself), can we still generate an interpolant with continuous derivative(s)???

An Old Idea: Splines

(Edited for Space, and "Content") Wikipedia Definition: Spline —

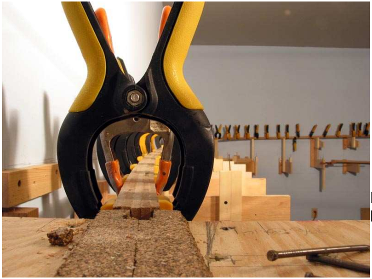
A spline consists of a long strip of wood (a lath) fixed in position at a number of points. Shipwrights often used splines to mark the curve of a hull. The lath will take **the shape which minimizes the energy required for bending it between the fixed points**, and thus adopt the smoothest possible shape.

Later craftsmen have made splines out of rubber, steel, and other elastomeric materials.

Spline devices help bend the wood for pianos, violins, violas, etc. The Wright brothers used one to shape the wings of their aircraft.

In 1946 mathematicians started studying the spline shape, and derived the **piecewise polynomial formula known as the spline curve or function.** This has led to the widespread use of such functions in **computer-aided design, especially in the surface designs of vehicles.**

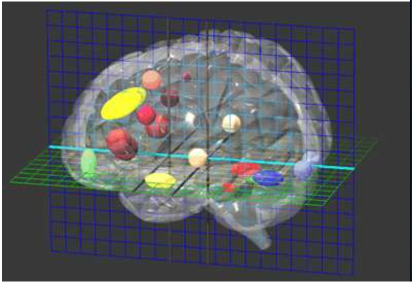
Modern Spline Construction: — A Model Railroad

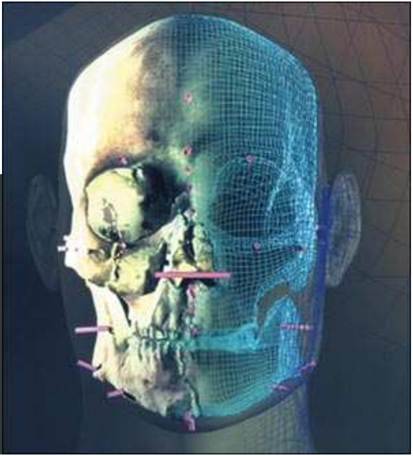


Pictures from Charlie Comstock's webpage http://s145079212.onlinehome.us/rr/

Applications & Pretty Pictures

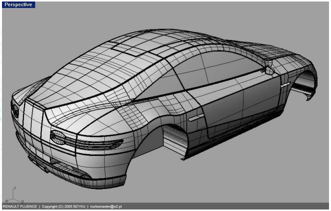
Provided by “Uncle Google”

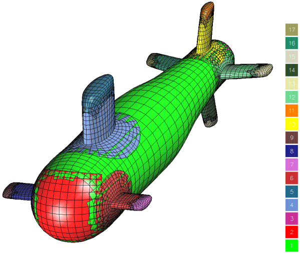


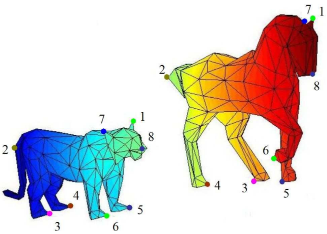


Applications & Pretty Pictures

Provided by "Uncle Google"







Cubic Splines to the Rescue!!! 1D-version

Given a function f defined on $[a, b]$ and a set of nodes $a = x_0 < x_1 < \cdots < x_n = b$, a **cubic spline interpolant** *S for f is a function that satisfies the following conditions:*

a. *$S(x)$ is a cubic polynomial, denoted $S_j(x)$, on the sub-interval $[x_j, x_{j+1}]$ $\forall j = 0, 1, \ldots, n-1$.*

Cubic Splines to the Rescue!!! 1D-version

Given a function f defined on $[a, b]$ and a set of nodes $a = x_0 < x_1 < \cdots < x_n = b$, a **cubic spline interpolant** *S for f is a function that satisfies the following conditions:*

a. *$S(x)$ is a cubic polynomial, denoted $S_j(x)$, on the sub-interval* $[x_j, x_{j+1}]$ $\forall j = 0, 1, \ldots, n-1$.

b. $S_j(x_j) = f(x_j)$, $\forall j = 0, 1, \ldots, (n-1)$. **"Left" Interpolation**

Cubic Splines to the Rescue!!! 1D-version

Given a function f defined on $[a,b]$ and a set of nodes $a = x_0 < x_1 < \cdots < x_n = b$, a **cubic spline interpolant** *S for f is a function that satisfies the following conditions:*

a. *$S(x)$ is a cubic polynomial, denoted $S_j(x)$, on the sub-interval $[x_j, x_{j+1}]$ $\forall j = 0, 1, \ldots, n-1$.*

b. $S_j(x_j) = f(x_j),\ \forall j = 0, 1, \ldots, (n-1).$ **"Left" Interpolation**

c. $S_j(x_{j+1}) = f(x_{j+1}),\ \forall j = 0, 1, \ldots, (n-1).$ **"Right" Interpolation**

Cubic Splines to the Rescue!!! 1D-version

Given a function f defined on $[a,b]$ and a set of nodes $a = x_0 < x_1 < \cdots < x_n = b$, a **cubic spline interpolant** *S for f is a function that satisfies the following conditions:*

a. *$S(x)$ is a cubic polynomial, denoted $S_j(x)$, on the sub-interval $[x_j, x_{j+1}]$ $\forall j = 0, 1, \ldots, n-1$.*

b. $S_j(x_j) = f(x_j), \ \forall j = 0, 1, \ldots, (n-1).$ **"Left" Interpolation**

c. $S_j(x_{j+1}) = f(x_{j+1}), \ \forall j = 0, 1, \ldots, (n-1).$ **"Right" Interpolation**

d. $S_j'(x_{j+1}) = S_{j+1}'(x_{j+1}), \ \forall j = 0, 1, \ldots, (n-2).$ **Slope-match**

Cubic Splines to the Rescue!!! 1D-version

Given a function f defined on $[a, b]$ and a set of nodes $a = x_0 < x_1 < \cdots < x_n = b$, a **cubic spline interpolant** *S for f is a function that satisfies the following conditions:*

a. *$S(x)$ is a cubic polynomial, denoted $S_j(x)$, on the sub-interval $[x_j, x_{j+1}]$ $\forall j = 0, 1, \ldots, n-1$.*

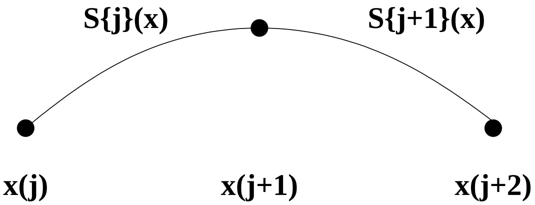
b. $S_j(x_j) = f(x_j), \;\forall j = 0, 1, \ldots, (n-1)$. **"Left" Interpolation**

c. $S_j(x_{j+1}) = f(x_{j+1}), \;\forall j = 0, 1, \ldots, (n-1)$. **"Right" Interpolation**

d. $S_j'(x_{j+1}) = S_{j+1}'(x_{j+1}), \;\forall j = 0, 1, \ldots, (n-2)$. **Slope-match**

e. $S_j''(x_{j+1}) = S_{j+1}''(x_{j+1}), \;\forall j = 0, 1, \ldots, (n-2)$. **Curvature-match**

A Spline Segment

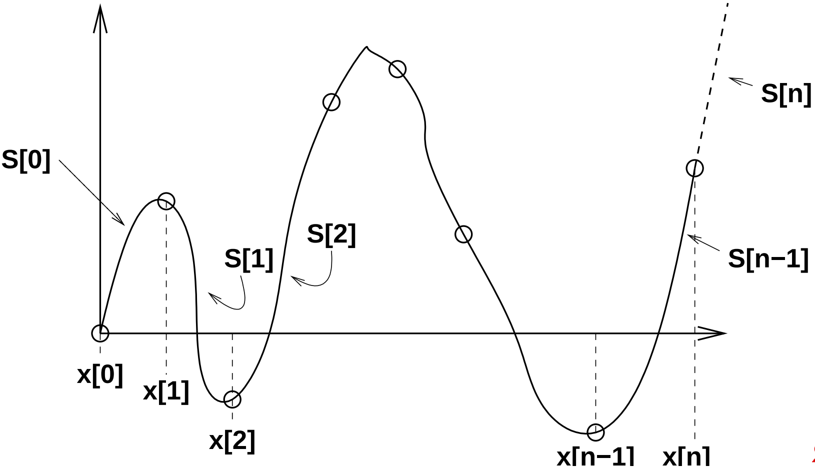


The spline segment $S_j(x)$ "lives" on the interval $[x_j, x_{j+1}]$.
The spline segment $S_{j+1}(x)$ "lives" on the interval $[x_{j+1}, x_{j+2}]$.

Their function values:	$S_j(x_{j+1}) = S_{j+1}(x_{j+1}) = f(x_{j+1})$
derivatives:	$S_j'(x_{j+1}) = S_{j+1}'(x_{j+1})$
and second derivatives:	$S_j''(x_{j+1}) = S_{j+1}''(x_{j+1})$

... are required to match in the **interior** point x_{j+1}.

Example "Cartoon": Cubic Spline.



Building Cubic Splines, I. — Applying the Conditions

We start with

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$$
$$\forall j \in \{0, 1, \dots, n-1\}$$

and apply all the conditions to these polynomials...

For convenience we introduce the notation $h_j = x_{j+1} - x_j$.

b. $S_j(x_j) = a_j = f(x_j)$

c. $S_{j+1}(x_{j+1}) = \mathbf{a_{j+1} = a_j + b_j h_j + c_j h_j^2 + d_j h_j^3} = S_j(x_{j+1})$

d. Notice $S_j'(x_j) = b_j$, hence we get $\mathbf{b_{j+1} = b_j + 2c_j h_j + 3d_j h_j^2}$

e. Notice $S_j''(x_j) = 2c_j$, hence we get $\mathbf{c_{j+1} = c_j + 3d_j h_j}$.

— We got a whole lot of equations to solve!!! (How many???)

Cubic Splines, II. — Solving the Resulting Equations.

We solve **[e]** for $d_j = \dfrac{c_{j+1} - c_j}{3h_j}$, and plug into **[c]** and **[d]** to get

Cubic Splines, II. — Solving the Resulting Equations.

We solve **[e]** for $d_j = \dfrac{c_{j+1} - c_j}{3h_j}$, and plug into **[c]** and **[d]** to get

[c'] $a_{j+1} = a_j + b_j h_j + \dfrac{h_j^2}{3}(2c_j + c_{j+1})$,

[d'] $b_{j+1} = b_j + h_j(c_j + c_{j+1})$.

Cubic Splines, II. — Solving the Resulting Equations.

We solve **[e]** for $d_j = \dfrac{c_{j+1} - c_j}{3h_j}$, and plug into **[c]** and **[d]** to get

[c'] $a_{j+1} = a_j + b_j h_j + \dfrac{h_j^2}{3}(2c_j + c_{j+1})$,

[d'] $b_{j+1} = b_j + h_j(c_j + c_{j+1})$.

We solve for b_j in **[c']** and get

[*] $b_j = \dfrac{1}{h_j}(a_{j+1} - a_j) - \dfrac{h_j}{3}(2c_j + c_{j+1})$.

Cubic Splines, II. — Solving the Resulting Equations.

We solve [**e**] for $d_j = \dfrac{c_{j+1} - c_j}{3h_j}$, and plug into [**c**] and [**d**] to get

[**c'**] $a_{j+1} = a_j + b_j h_j + \dfrac{h_j^2}{3}(2c_j + c_{j+1})$,

[**d'**] $b_{j+1} = b_j + h_j(c_j + c_{j+1})$.

We solve for b_j in [**c'**] and get

[*****] $b_j = \dfrac{1}{h_j}(a_{j+1} - a_j) - \dfrac{h_j}{3}(2c_j + c_{j+1})$.

Reduce the index by 1, to get

[***'**] $b_{j-1} = \dfrac{1}{h_{j-1}}(a_j - a_{j-1}) - \dfrac{h_{j-1}}{3}(2c_{j-1} + c_j)$.

Cubic Splines, II. — Solving the Resulting Equations.

We solve [**e**] for $d_j = \dfrac{c_{j+1} - c_j}{3h_j}$, and plug into [**c**] and [**d**] to get

[**c'**] $a_{j+1} = a_j + b_j h_j + \dfrac{h_j^2}{3}(2c_j + c_{j+1})$,

[**d'**] $b_{j+1} = b_j + h_j(c_j + c_{j+1})$.

We solve for b_j in [**c'**] and get

[*] $b_j = \dfrac{1}{h_j}(a_{j+1} - a_j) - \dfrac{h_j}{3}(2c_j + c_{j+1})$.

Reduce the index by 1, to get

[*'] $b_{j-1} = \dfrac{1}{h_{j-1}}(a_j - a_{j-1}) - \dfrac{h_{j-1}}{3}(2c_{j-1} + c_j)$.

Plug [*] (lhs) and [*'] (rhs) into the index-reduced-by-1 version of [**d'**], *i.e.*

[**d"**] $b_j = b_{j-1} + h_{j-1}(c_{j-1} + c_j)$.

Cubic Splines, III. — A Linear System of Equations

After some "massaging" we end up with the linear system of equations for $j \in \{1, 2, \ldots, n-1\}$ (the interior nodes).

$$h_{j-1}c_{j-1} + 2(h_{j-1} + h_j)c_j + h_j c_{j+1} = \frac{3}{h_j}(a_{j+1} - a_j) - \frac{3}{h_{j-1}}(a_j - a_{j-1}).$$

Notice: The only unknowns are $\{c_j\}_{j=0}^{n}$, since the values of $\{a_j\}_{j=0}^{n}$ and $\{h_j\}_{j=0}^{n-1}$ are given.

Cubic Splines, III. — A Linear System of Equations

After some "massaging" we end up with the linear system of equations for $j \in \{1, 2, \ldots, n-1\}$ (the interior nodes).

$$h_{j-1}c_{j-1} + 2(h_{j-1} + h_j)c_j + h_j c_{j+1} = \frac{3}{h_j}(a_{j+1} - a_j) - \frac{3}{h_{j-1}}(a_j - a_{j-1}).$$

Notice: The only unknowns are $\{c_j\}_{j=0}^{n}$, since the values of $\{a_j\}_{j=0}^{n}$ and $\{h_j\}_{j=0}^{n-1}$ are given.

Once we compute $\{c_j\}_{j=0}^{n-1}$, we get

$$b_j = \frac{a_{j+1} - a_j}{h_j} - \frac{h_j(2c_j + c_{j+1})}{3}, \quad \text{and} \quad d_j = \frac{c_{j+1} - c_j}{3h_j}.$$

Cubic Splines, III. — A Linear System of Equations

After some "massaging" we end up with the linear system of equations for $j \in \{1, 2, \ldots, n-1\}$ (the interior nodes).

$$h_{j-1}c_{j-1} + 2(h_{j-1} + h_j)c_j + h_j c_{j+1} = \frac{3}{h_j}(a_{j+1} - a_j) - \frac{3}{h_{j-1}}(a_j - a_{j-1}).$$

Notice: The only unknowns are $\{c_j\}_{j=0}^{n}$, since the values of $\{a_j\}_{j=0}^{n}$ and $\{h_j\}_{j=0}^{n-1}$ are given.

Once we compute $\{c_j\}_{j=0}^{n-1}$, we get

$$b_j = \frac{a_{j+1} - a_j}{h_j} - \frac{h_j(2c_j + c_{j+1})}{3}, \quad \text{and} \quad d_j = \frac{c_{j+1} - c_j}{3h_j}.$$

We are **almost** ready to solve for the coefficients $\{c_j\}_{j=0}^{n-1}$, but we only have $(n-1)$ equations for $(n+1)$ unknowns...

Cubic Splines, IV. — Completing the System

We can complete the system in many ways, some common ones are...

Natural boundary conditions:

$$\textbf{[n1]} \quad 0 = S_0''(x_0) = 2c_0 \quad \Rightarrow \quad c_0 = 0$$
$$\textbf{[n2]} \quad 0 = S_n''(x_n) = 2c_n \quad \Rightarrow \quad c_n = 0$$

Cubic Splines, IV. — Completing the System

We can complete the system in many ways, some common ones are...

Clamped boundary conditions: (Derivative known at endpoints).

$$
\begin{array}{llcl}
[\mathbf{c1}] & S_0'(x_0) & = & b_0 = f'(x_0) \\
[\mathbf{c2}] & S_{n-1}'(x_n) & = & b_n = b_{n-1} + h_{n-1}(c_{n-1} + c_n) = f'(x_n)
\end{array}
$$

[c1] and **[c2]** give the additional equations

$$
\begin{array}{lrcl}
[\mathbf{c1}'] & 2h_0c_0 + h_0c_1 & = & \frac{3}{h_0}(a_1 - a_0) - 3f'(x_0) \\
[\mathbf{c2}'] & h_{n-1}c_{n-1} + 2h_{n-1}c_n & = & 3f'(x_n) - \frac{3}{h_{n-1}}(a_n - a_{n-1}).
\end{array}
$$

Cubic Splines, IV. — Completing the System

Given a function f defined on $[a, b]$ and a set of nodes $a = x_0 < x_1 < \cdots < x_n = b$, a **cubic spline interpolant** *S for f is a function that satisfies the following conditions:*

a. *$S(x)$ is a cubic polynomial, denoted $S_j(x)$, on the sub-interval* $[x_j, x_{j+1}]$ $\forall j = 0, 1, \ldots, n-1$.

b. $S_j(x_j) = f(x_j), \ \forall j = 0, 1, \ldots, (n-1)$. **"Left" Interpolation**

c. $S_j(x_{j+1}) = f(x_{j+1}), \ \forall j = 0, 1, \ldots, (n-1)$. **"Right" Interpolation**

d. $S_j'(x_{j+1}) = S_{j+1}'(x_{j+1}), \ \forall j = 0, 1, \ldots, (n-2)$. **Slope-match**

e. $S_j''(x_{j+1}) = S_{j+1}''(x_{j+1}), \ \forall j = 0, 1, \ldots, (n-2)$. **Curvature-match**

---

f. *One of the following sets of boundary conditions is satisfied:*

1. $S''(x_0) = S''(x_n) = 0$, – **free / natural boundary**
2. $S'(x_0) = f'(x_0)$ *and* $S'(x_n) = f'(x_n)$, – **clamped boundary**

Natural Boundary Conditions: Linear System, $A\tilde{\mathbf{x}} = \tilde{\mathbf{y}}$

We end up with a linear system of equations, $A\tilde{\mathbf{x}} = \tilde{\mathbf{y}}$, where

$$A = \begin{bmatrix} \mathbf{1} & 0 & 0 & \cdots & \cdots & 0 \\ h_0 & 2(h_0+h_1) & h_1 & \ddots & & \vdots \\ 0 & h_1 & 2(h_1+h_2) & h_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & \cdots & \cdots & 0 & 0 & \mathbf{1} \end{bmatrix},$$

Boundary Terms: marked in red-bold.

Natural Boundary Conditions: Linear System, $A\tilde{\mathbf{x}} = \tilde{\mathbf{y}}$

We end up with a linear system of equations, $A\tilde{\mathbf{x}} = \tilde{\mathbf{y}}$, where

$$\tilde{\mathbf{y}} = \begin{bmatrix} \mathbf{0} \\ \frac{3(a_2-a_1)}{h_1} - \frac{3(a_1-a_0)}{h_0} \\ \vdots \\ \frac{3(a_n-a_{n-1})}{h_{n-1}} - \frac{3(a_{n-1}-a_{n-2})}{h_{n-2}} \\ \mathbf{0} \end{bmatrix}, \qquad \tilde{\mathbf{x}} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix}$$

$\tilde{\mathbf{x}}$ are the unknowns (the quantity we are solving for!)

Boundary Terms: marked in red-bold.

Clamped Boundary Conditions: Linear System

We end up with a linear system of equations, $A\tilde{\mathbf{x}} = \tilde{\mathbf{y}}$, where

$$A = \begin{bmatrix} \mathbf{2h_0} & \mathbf{h_0} & 0 & \cdots & \cdots & 0 \\ h_0 & 2(h_0+h_1) & h_1 & \ddots & & \vdots \\ 0 & h_1 & 2(h_1+h_2) & h_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & \cdots & \cdots & 0 & \mathbf{h_{n-1}} & \mathbf{2h_{n-1}} \end{bmatrix},$$

Boundary Terms: marked in red-bold.

Clamped Boundary Conditions: Linear System

We end up with a linear system of equations, $A\tilde{\mathbf{x}} = \tilde{\mathbf{y}}$, where

$$\tilde{\mathbf{y}} = \begin{bmatrix} \boldsymbol{\frac{3(a_1 - a_0)}{h_0} - 3f'(x_0)} \\ \frac{3(a_2 - a_1)}{h_1} - \frac{3(a_1 - a_0)}{h_0} \\ \vdots \\ \frac{3(a_n - a_{n-1})}{h_{n-1}} - \frac{3(a_{n-1} - a_{n-2})}{h_{n-2}} \\ \boldsymbol{3f'(x_n) - \frac{3(a_n - a_{n-1})}{h_{n-1}}} \end{bmatrix}, \quad \tilde{\mathbf{x}} = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix}$$

Boundary Terms: marked in red-bold.

Cubic Splines, The Error Bound

No numerical story is complete without an error bound...

If $f \in C^4[a,b]$, *let*

$$M = \max_{a \le x \le b} |f^4(x)|.$$

If S *is the unique* **clamped cubic spline interpolant** *to* f *with respect to the nodes* $a = x_0 < x_1 < \cdots < x_n = b$, *then with*

$$h = \max_{0 \le j \le n-1} (x_{j+1} - x_j) = \max_{0 \le j \le n-1} h_j$$

$$\max_{a \le x \le b} |f(x) - S(x)| \le \frac{5Mh^4}{384}$$

Banded Matrices

[Reference]

We notice that the linear systems for both natural and clamped boundary conditions give rise to **tri-diagonal linear systems**.

Further, these systems are **strictly diagonally dominant** — the entries on the diagonal outweigh the sum of the off-diagonal elements (in absolute terms) —, so pivoting (re-arrangement to avoid division by a small number) is not needed when solving for $\tilde{\mathbf{x}}$ using Gaussian Elimination...

This means that these systems can be solved very quickly (we will revisit this topic later on, but for now the algorithm is on the next couple of slides), see also *"Computational Linear Algebra / Numerical Matrix Analysis."* Math 543

Algorithm: Solving $Tx = b$ in $\mathcal{O}(n)$ Time, I. [REFERENCE]

Given the $N \times N$ tridiagonal matrix T and the $N \times 1$ vector y:

`Step 1:` `The first row:`

$$\begin{aligned} l_{1,1} &= T_{1,1} \\ u_{1,2} &= T_{1,2}/l_{1,1} \\ z_1 &= y_1/l_{1,1} \end{aligned}$$

`Step 2:` `FOR` $i = 2 : (n-1)$

$$\begin{aligned} l_{i,i-1} &= T_{i,i-1} \\ l_{i,i} &= T_{i,i} - l_{i,i-1}u_{i-1,i} \\ u_{i,i+1} &= T_{i,i+1}/l_{i,i} \\ z_i &= (y_i - l_{i,i-1}z_{i-1})/l_{i,i} \end{aligned}$$

`END`

`Step 3:` `The last row:`

$$\begin{aligned} l_{n,n-1} &= T_{n,n-1} \\ l_{n,n} &= T_{n,n} - l_{n,n-1}u_{n-1,n} \\ z_n &= (y_n - l_{n,n-1}z_{n-1})/l_{n,n} \end{aligned}$$

Algorithm: Solving $Tx = b$ in $\mathcal{O}(n)$ Time, II. [Reference]

Step 4: $x_n = z_n$

Step 5: FOR $i = (n-1) : -1 : 1$

$$x_i = z_i - u_{i,i+1} x_{i+1}$$

END

Notes: The algorithm computes both the LU-factorization of T, as well as the solution $\tilde{\mathbf{x}} = T^{-1}\tilde{\mathbf{y}}$. Steps 1–3 computes $\tilde{\mathbf{z}} = L^{-1}\tilde{\mathbf{y}}$, and steps 4–5 computes $\tilde{\mathbf{x}} = U^{-1}\tilde{\mathbf{z}}$. (This will gain meaning later on, when we talk about Gaussian Elimination and Matrix Factorizations — Don't worry if it makes no sense at all right now!)