Polynomial Interpolation

A commonly used approach is to use a properly crafted polynomial function

$$f(x) = P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n$$

to interpolate the points $(x_0, y_0), \ldots, (x_k, y_k)$.

Some benefits:

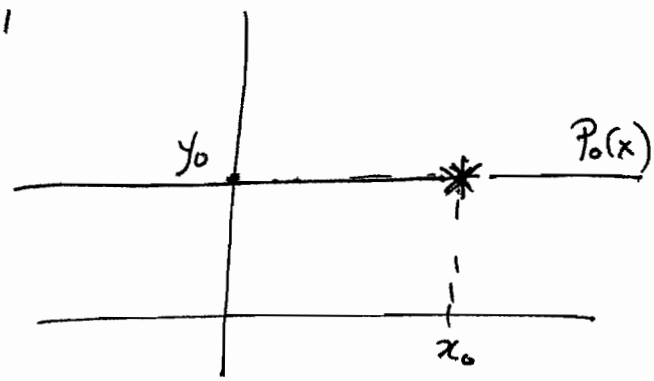
- Polynomials are relatively simple to evaluate
  (writing $P_n(x) = a_0 + x(a_1 + x(a_2 + x(a_3 \cdots + a_{n-1} + x a_n)))$
  we see that one needs n multiplications & n additions)
- We can easily compute derivatives P_n', P_n'' if desired.
- Reasonably established procedure, to determine the a_i's.
- Polynomial approximations are <u>familiar</u> from, e.g. Taylor series.

And, some disadvantages:

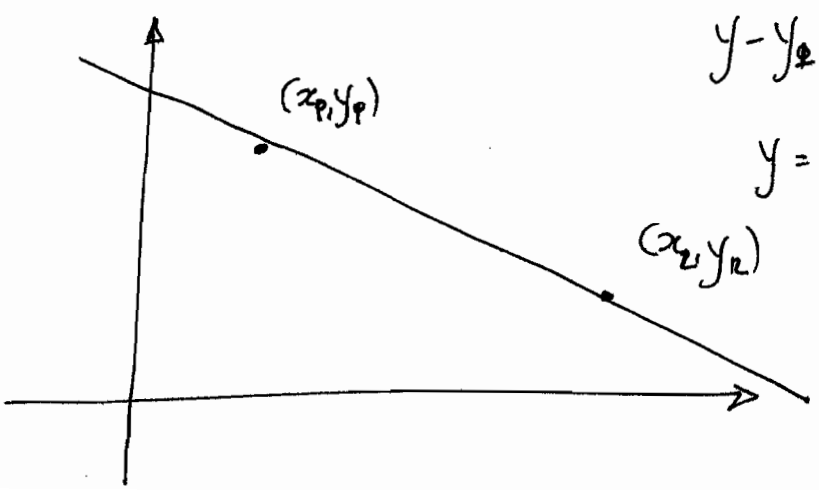
- Fitting polynomials can be problematic, when
  (i) we have <u>many</u> data points (k is large), or
  (ii) Some of the samples are too close together ($|x_i - x_j| = $ small

In the interest of simplicity (and not only) we try to find the most basic, yet adequate, $P_n(x)$ that interpolates $(x_1, y_1), \ldots, (x_k, y_k)$. For example.

- If $k=1$ (only one data sample) we have to interpolate through (x_1, y_1). A 0-degree polynomial (constant) will achieve that, if $a_0 = y_1$



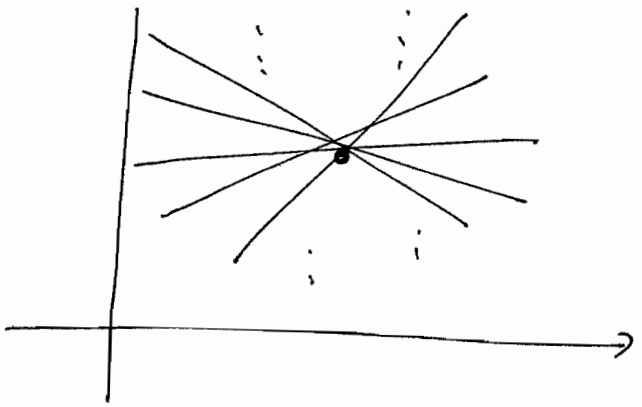
- If $k=2$ we have 2 points (x_1, y_1) & (x_2, y_2). A 0-degree polynomial $P_0(x) = a_0$ will not always be able to pass through both points (unless $y_1 = y_2$), but a degree-1 polynomial $P_1(x) = a_0 + a_1 x$ always can



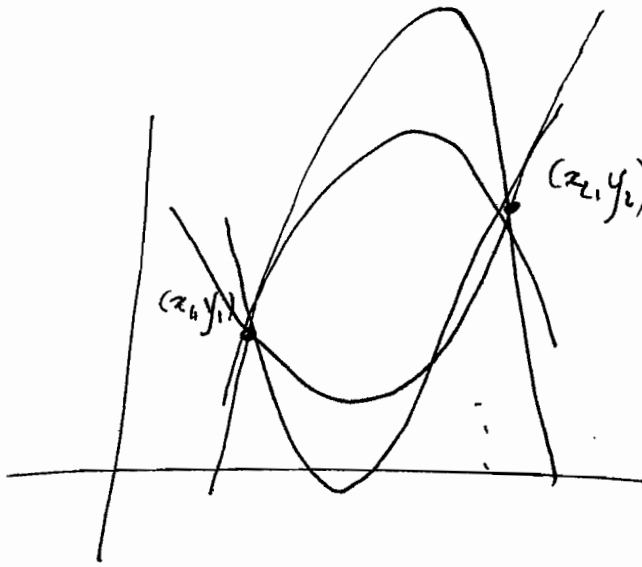
$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

$$y = \underbrace{y_1 - \frac{y_2 - y_1}{x_2 - x_1} x_1}_{a_0} + \underbrace{\frac{y_2 - y_1}{x_2 - x_1}}_{a_1} x$$

These are not the only polynomials that accomplish the task. e.g.:



or



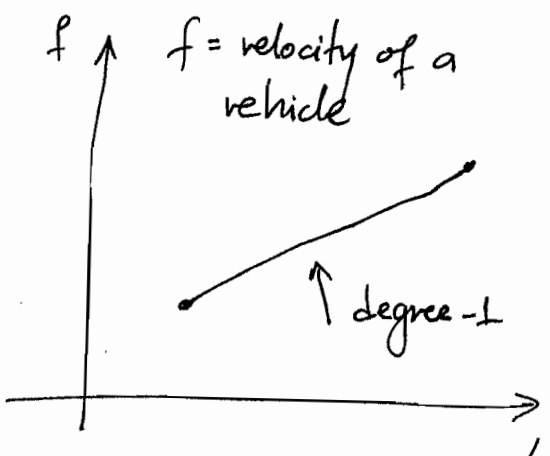
Interpolating (x_1, y_1) w/ 1-degree polynomials

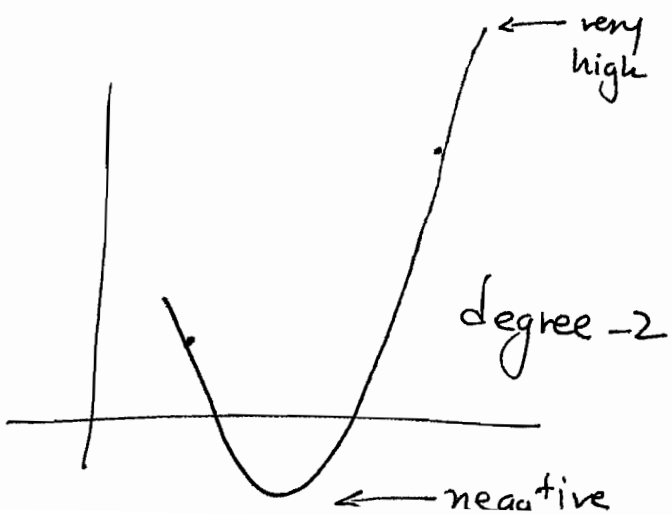
Using degree-2

The problem with using a degree higher than the minimum necessary is that:

- More than 1 solutions become available, with the "right" one being unclear
- Wildly varying curves become permissible, producing questionable approximations

e.g.





In fact we can show that using a polynomial $P_n(x)$ of degree n is the best choice when interpolating $n+1$ points. In this case the following properties are assured:

- Existence: Such a polynomial always exists (assuming that all the x_i's are different! it would be impossible for a function to pass through 2 points on the same vertical line). We will show this later, by constructing such a function.

- Uniqueness: We can sketch a proof:

  Assume that $P_n(x) = p_0 + p_1 x + \cdots + p_n x^n$

  $Q_n(x) = q_0 + q_1 x + \cdots + q_n x^n$

  both interpolate every (x_i, y_i), i.e. $P_n(x_i) = Q_n(x_i) = y_i \quad \forall i$

  Define another n-degree Polynomial

  $$r_0 + r_1 x + \cdots + r_n x^n = R_n(x) = P_n(x) - Q_n(x).$$

  Apparently $R_n(x_i) = 0 \quad \forall i = 1, 2, \ldots, n+1$.

  From algebra we know that every polynomial of degree n has at most n real roots, unless it is the zero polynomial, i.e. $r_0 = r_1 = \cdots = r_n = 0$. Since we have $R_n(x) = 0$ for $n+1$ distinct values, we must have $R_n(x) \equiv 0$

  or $P_n(x) \equiv Q_n(x)$.

In the previous lecture we addressed the problem of interpolation: We assumed that we only know a certain function $f(x)$ through samples of its values $y_1 = f(x_1), y_2 = f(x_2), \ldots, y_N = f(x_N)$ at the N locations $x_1, x_2, \ldots, x_N$.

We want to reconstruct (or guess) a function $f(x)$, defined for arbitrary values of x (beyond the ones at $x_1, \ldots, x_N$). Knowing such a function enables us to:

- Estimate values of f in between the samples $\{x_i\}$.
- Make predictions for the value of even beyond the range spanned by the $\{x_i\}$ (i.e, beyond the maximum x_i, or below the minimum one).
- Make an estimate for the derivative(s) of f at arbitrary locations

Polynomial interpolation Performs this task by passing an appropriate polynomial through data points $(x_1, y_1) \ldots (x_N, y_N)$. In particular, we choose to employe a degree-n polynomial

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n$$

when we have $(n+1)$ data points

$$(x_1, y_1), (x_2, y_2), \ldots, (x_{n+1}, y_{n+1})$$

to interpolate through. This choice is made in order to ensure existence (such a polynomial exists) and uniqueness (only <u>one</u> such polynomial exists). Let's examine these properties more closely:

<u>Existence</u>: We will later show that, with a degree-n polynomial it is always possible to match $n+1$ data points. For now, we can at least show that such a task would have been impossible (in general) if we were only allowed to use degree-$(n-1)$ polynomials. In fact, consider the points

$$(x_1, y_1=0), (x_2, y_2=0), \ldots, (x_n, y_n=0) \text{ and } (x_{n+1}, y_{n+1}=1).$$

Thus, if a $(n-1)$-degree polynomial was able to interpolate these points, we would have:

$$P_{n-1}(x_1) = P_{n-1}(x_2) = \cdots = P_{n-1}(x_n) = 0.$$

P_{n-1} can only equal zero at exactly $n-1$ locations (max) <u>unless</u> it is the zero polynomial $P_{n-1}(x) \equiv 0$. This is a contradiction as $P_{n-1}(x_{n+1}) \neq 0$.

Uniqueness: Assume that both $P_n(x)$ & $Q_n(x)$ interpolate all data points $(x_1, y_1), \ldots, (x_{n+1}, y_{n+1})$. Thus.

$$P_n(x_1) = Q_n(x_1)\ (=y_1),\ \ldots\ldots,\ P_n(x_{n+1}) = Q_n(x_{n+1})\ (=y_{n+1})$$

The polynomial $R_n(x) = P_n(x) - Q_n(x)$ thus satisfies

$$R_n(x_i) = 0 \quad \forall i \in \{1, 2, \ldots, n+1\}.$$

R_n is of degree n, thus it can only be zero at n points unless (which is the case here) $R_n(x) \equiv 0$, i.e. $P_n(x) \equiv Q_n(x)$.

The most basic procedure to determine the coefficients $a_0, a_1, \ldots, a_n$ of the interpolating polynomial $P_n(x)$, is to write a linear system of equations as follows:

$$\left.\begin{aligned} P_n(x_1) &= y_1 \\ P_n(x_2) &= y_2 \\ &\vdots \\ P_n(x_n) &= y_n \\ P_n(x_{n+1}) &= y_{n+1} \end{aligned}\right\} \Rightarrow$$

$$\begin{aligned} a_0 + a_1 x_1 + a_2 x_1^2 + \cdots + a_{n-1} x_1^{n-1} + a_n x_1^n &= y_1 \\ a_0 + a_1 x_2 + a_2 x_2^2 + \cdots + a_{n-1} x_2^{n-1} + a_n x_2^n &= y_2 \\ &\vdots \\ a_0 + a_1 x_n + a_2 x_n^2 + \cdots + a_{n-1} x_n^{n-1} + a_n x_n^n &= y_n \\ a_0 + a_1 x_{n+1} + a_2 x_{n+1}^2 + \cdots + a_{n-1} x_{n+1}^{n-1} + a_n x_{n+1}^n &= y_{n+1} \end{aligned}$$

or, in matrix form:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} & x_2^n \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} & x_n^n \\ 1 & x_{n+1} & x_{n+1}^2 & \cdots & x_{n+1}^{n-1} & x_{n+1}^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \\ a_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \\ y_{n+1} \end{bmatrix}$$

$(n+1)\times(n+1)$ matrix $\quad$ $(n+1)$-vector $\quad$ $(n+1)$-vector

$$V \cdot \underset{\sim}{a} = \underset{\sim}{y}$$

The matrix V is called a Vandermonde matrix. We will see that V is non-singular, thus we can solve the system $V\underset{\sim}{a} = \underset{\sim}{y}$ to obtain the coefficients $\underset{\sim}{a} = (a_0, a_1, \ldots, a_n)$. Let's evaluate the merits and drawbacks of this approach:

- Cost to determine the polynomial $P_n(x)$: VERY COSTLY, since a dense $(n+1)\times(n+1)$ linear system has to be solved. This will generally require time proportional to n^3, making large interpolation problems intractable. In addition, the Vandermonde matrix is notorious for being challenging to solve (esp. with Gauss elimination) and prone to large errors in the computed coefficients $\{a_i\}$, when n is large and/or $x_i \approx x_j$.

- Cost to evaluate $f(x)$ (x = arbitrary) if coefficients are known: VERY CHEAP. Using Horner's scheme:

$$a_0 + a_1 x + \cdots + a_n x^n = a_0 + x\left(a_1 + x\left(a_2 + \cdots + x\left(a_{n-1} + x a_n\right)\right)\right)$$

- Availability of derivatives: VERY EASY. e.g.

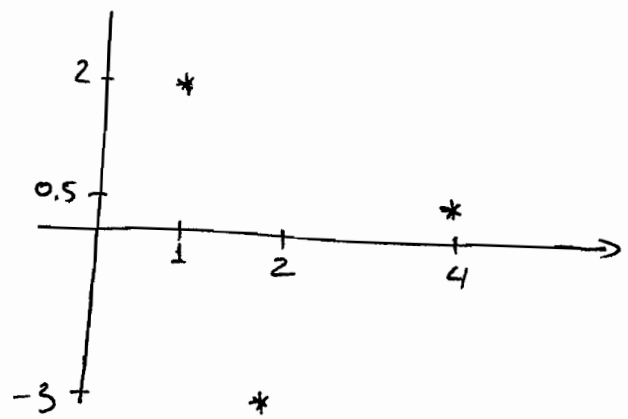
$$P_n'(x) = a_1 + 2a_2 x + 3a_3 x^2 + \cdots + (n-1)a_{n-1} x^{n-2} + n a_n x^{n-1}$$

- Allows incremental interpolation: NO!
This property examines if interpolating through $(x_1, y_1), \ldots, (x_{n+1}, y_{n+1})$ is <u>easier</u> if we already know a polynomial (of degree $= n$) that interpolates through $(x_1, y_1), \ldots, (x_n, y_n)$. In our case the system $V\underline{a} = \underline{y}$ would have to be solved from scratch for the $(n+1)$ data points.

<u>Lagrange interpolation</u> (§4.3) is an alternative way to define $P_n(x)$, without having to solve expensive systems of equations.

We shall explain how Lagrange interpolation works, with an example.

Example: Pass a ~~cubic~~ quadratic polynomial through $(1,2)$, $(2,-3)$, $(4,0.5)$.

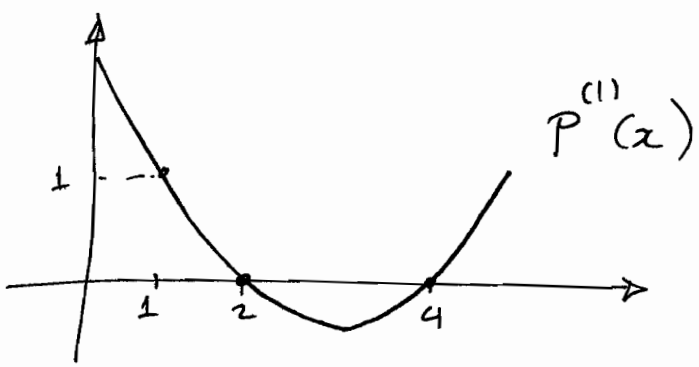


Assume we have somehow constructed 3 ~~cubic~~ quadratic polynomials $P^{(1)}(x)$, $P^{(2)}(x)$, $P^{(3)}(x)$, such that:

$P^{(1)}(1) = 1$

$P^{(1)}(2) = 0$

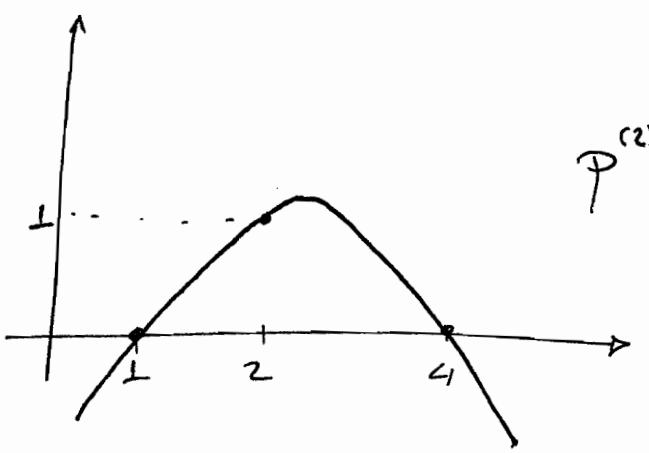
$P^{(1)}(4) = 0$



$P^{(2)}(1) = 0$

$P^{(2)}(2) = 1$

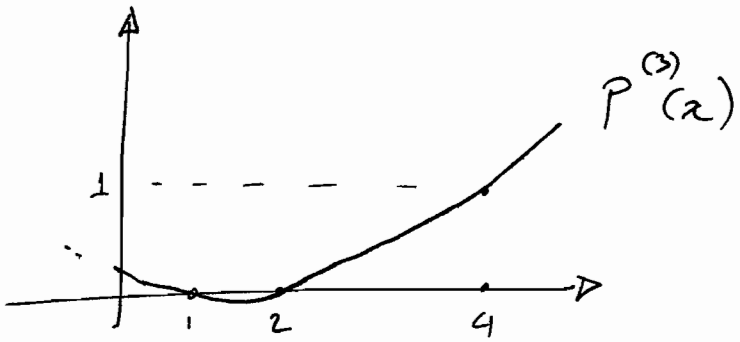
$P^{(2)}(4) = 0$



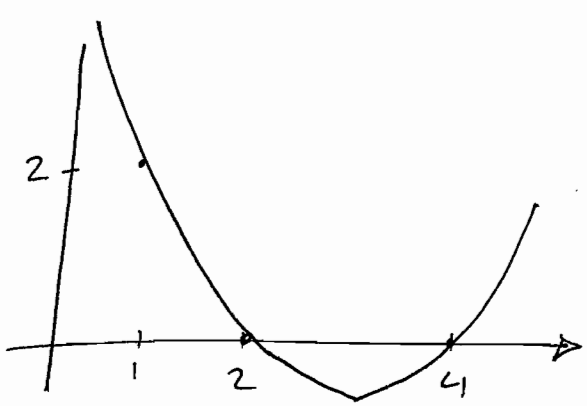
$P^{(3)}(1) = 0$

$P^{(3)}(2) = 0$

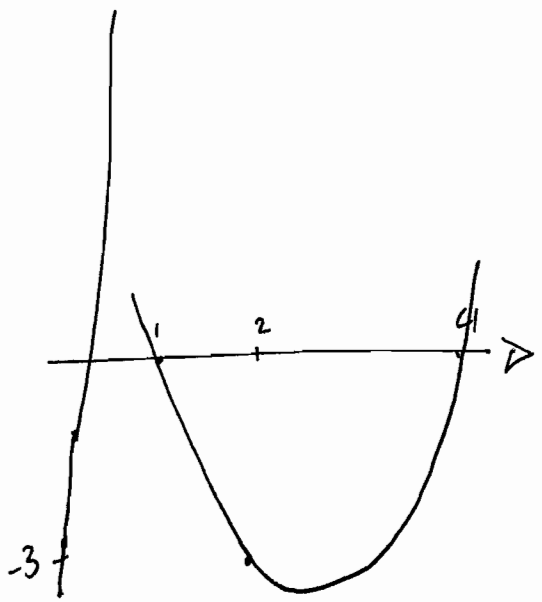
$P^{(3)}(4) = 1$



Now the idea is to <u>scale</u> each $P^{(i)}$, such that $P^{(i)}(x_i) = y_i$ and add them all together

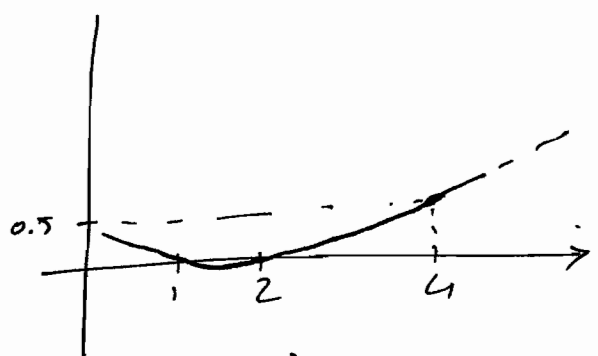


$2 \times P^{(1)}(x)$ +



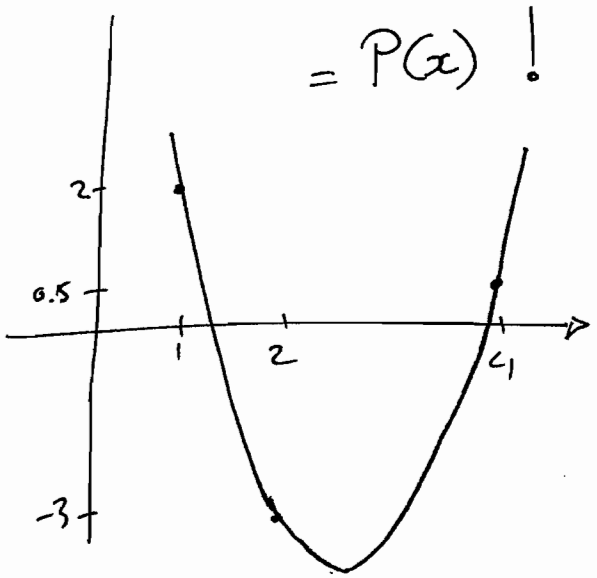
$-3 \times P^{(1)}(x)$

+



$0.5 \times P^{(3)}(x)$

=



$= P(x)$!

In summary, if we have a total of $(n+1)$ data points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, define the Lagrange polynomials of n-degree $l_0(x), l_1(x), \ldots, l_n(x)$ as:

$$l_i(x_j) = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j. \end{cases}$$

Then, the interpolating polynomial is simply

$$P(x) = y_0 l_0(x) + y_1 l_1(x) + \cdots + y_n l_n(x) = \sum_{i=0}^{n} y_i l_i(x).$$

No solution of a linear system is necessary here. We just have to explain what every $l_i(x)$ looks like ...

Since $l_i(x)$ is an n-degree polynomial, with the n-roots $x_0, x_1, x_2, \ldots, x_{i-1}, x_{i+1}, x_{i+2}, \ldots, x_n$, it must have the form

$$l_i(x) = \underbrace{C_i}_{\text{const}} (x-x_0)(x-x_1)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_n)$$

$$= C_i \prod_{j \neq i} (x - x_j)$$

Now, we require $l_i(x_i) = 1$, thus: $1 = C_i \cdot \prod_{j \neq i} (x_i - x_j)$

$$\Rightarrow C_i = \frac{1}{\prod_{i \neq i} (x_i - x_j)}$$

Thus, for every i, we have

$$l_i(x) = \frac{(x-x_0)(x-x_1)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_n)}{(x_i-x_0)(x_i-x_1)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_n)}$$

$$= \prod_{j\neq i}\left(\frac{x-x_j}{x_i-x_j}\right). = \frac{\prod_{j\neq i}(x-x_j)}{\prod_{j\neq i}(x_i-x_j)}$$

Note This result essentially proves existence of a polynomial interpolant of degree $=n$ that passes through $(n+1)$ data points

We can also use it to prove that the Vandermonde matrix V is non-singular; if it were singular, a right-hand-side $\underline{y} = (y_0, \ldots, y_n)$ would have existed such that $V\underline{a} = \underline{y}$ would have no solution, which is a contradiction

Let's evaluate the same 4 quality metrics we saw before, for the Vandermonde matrix approach.

- Cost of determining $P(x)$: VERY EASY, we are essentially able to write a formula for $P(x)$ without solving any systems.

However, if we want to write $P(x) = a_0 + a_1x + \cdots + a_nx^n$, the cost of evaluating the a_i's would be very high!

each l_i would need to be expanded $\Rightarrow$ approx. N^2 operations for each l_i; N^3 operations for $P(x)$.

- Cost of evaluating $P(x)$ [x: arbitrary]. SIGNIFICANT.

We do not really need to compute the a_i's beforehand, if we only need to evaluate $P(x)$ at select few locations. for each $l_i(x)$ the evaluation requires N a subtraction & N multiplications $\Rightarrow$ total = about N^2 operations (better than N^3 for computing the a_i's).

- Availability of derivatives : NOT READILY AVAILABLE

Differentiating each l_i (since $P'(x) = \sum y_i l_i'(x)$) is not trivial $\Rightarrow$ yields N terms each, with $(N-1)$ products per term.

- Incremental interpolation. NOT ACCOMMODATED.

Still, Lagrange interpolation is a good quality method, if we can accept its limitations.

Newton interpolation is yet another alternative, which enables both efficient evaluation and allows for incremental construction. Additionally (as we will see in next lecture) it allows both the coefficients $\{a_i\}$ as well as the derivative $P'(x)$ to be evaluated efficiently.

Lagrange interpolation

We seek an n-degree polynomial $P_n(x)$ which interpolates the data points $(x_0, y_0), (x_1, y_1), \ldots (x_n, y_n)$.

Lagrange interpolation constructs P_n as:

$$P_n(x) = y_0 l_0(x) + y_1 l_1(x) + \cdots + y_n l_n(x)$$

Each of the $l_i(x)$'s is an n-degree polynomial, which equals zero at every x_j $(j \neq i)$, while $l_i(x_i) = 1$.

We saw that this can be constructed as:

$$l_i(x) = \frac{(x-x_1)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_n)}{(x_i-x_1)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_n)} = \frac{\prod_{j\neq i}(x-x_j)}{\prod_{j\neq i}(x_i-x_j)}$$

from this definition it is obvious that

$$l_i(x_j) = \begin{cases} 1, & \text{if } i=j \\ 0, & \text{if } i\neq j \end{cases}$$

Let us evaluate this approach, as we did with the Vandermonde-system method:

- Cost of determining $P(x)$: VERY EASY. Essentially we can write a formula for $P(x) = y_0 l_0(x) + \cdots + y_n l_n(x)$ without solving any system.
  However: if we wanted to write $P(x)$ in the form $a_0 + a_1 x + \cdots + a_n x^n$ the cost for this would be very high! Even writing a single $l_i(x)$ in this form would require $\approx n^2$ operations (if we are careful how we do it), leading to a $O(n^3)$ cost for the entire $P(x)$.
- Cost of evaluating $P(x)$ for an arbitrary x: SIGNIFICANT
  If we don't want to precompute the a_i's, evaluating each $l_i(x)$ requires n subtractions & n multiplications. In total, we need about n^2 operations to compute $P(x)$. This is not as bad as the n^3 operations to find the a_i's, but still quite high.
- Availability of derivatives: NOT READILY AVAILABLE
  Differentiating each l_i (using product rule) yields n terms, each with $n-1$ factors $\Rightarrow$ expensive.

- Incremental construction: NOT SUPPORTED

  The construction of the l_i's assumes we know all the x_i's. However building $P(x)$ from scratch if we are given an extra data point is not all that expensive...

Still, Lagrange interpolation is a good quality method, if we can accept its limitations.

Newton interpolation (§4.4) is another alternative, which enables both efficient evaluation and allows for incremental construction. Additionally, it allows the a_i's to be evaluated efficiently, and from those we can easily obtain derivatives, too.

Here is the basic idea:

We want to interpolate $(x_0, y_0), \ldots, (x_n, y_n)$.

Step 0: Define a 0-degree polynomial $P_0(x)$ that just interpolates (x_0, y_0). Obviously, we can achieve that by simply selecting

$$P_0(x) = y_0$$

Step 1 Define a 1st degree polynomial $P_1(x)$ that now interpolates both (x_0, y_0) and (x_1, y_1). We also want to take advantage of the previously defined $P_0(x)$, by constructing P_1 as:

$$P_1(x) = P_0(x) + M_1(x)$$

$M_1(x)$ is a 1st degree polynomial and it needs to satisfy:

$$\underbrace{P_1(x_0)}_{=y_0} = \underbrace{P_0(x_0)}_{=y_0} + M_1(x_0) \Rightarrow M_1(x_0) = 0$$

Thus $M_1(x) = c_1(x - x_0)$. We can determine c_1 using:

$$P_1(x_1) = P_0(x_1) + c_1(x_1 - x_0) \Rightarrow c_1 = \frac{P_1(x_1) - P_0(x_1)}{x_1 - x_0} = \frac{y_1 - P_0(x_1)}{x_1 - x_0}$$

Step 2: Now construct $P_2(x)$ which interpolates the three points (x_0, y_0), (x_1, y_1), (x_2, y_2). Define it as:

$$P_2(x) = P_1(x) + M_2(x) \qquad (M_2: \text{degree} = 2).$$

Once again we observe that

$$\left.\begin{aligned} \underbrace{P_2(x_0)}_{=y_0} &= \underbrace{P_1(x_0)}_{=y_0} + M_2(x_0) \\ \underbrace{P_2(x_1)}_{=y_0} &= \underbrace{P_1(x_1)}_{=y_0} + M_2(x_1) \end{aligned}\right\} \Rightarrow M_2(x_0) = M_2(x_1) = 0$$

Thus $M_2(x)$ must have the form:

$$M_2(x) = c_2(x-x_0)(x-x_1).$$

Substituting $x \leftarrow x_2$ we get an expression for c_2

$$y_2 = P_2(x_2) = P_1(x_2) + c_2(x_2-x_0)(x_2-x_1)$$

$$\Rightarrow c_2 = \frac{y_2 - P_1(x_2)}{(x_2-x_0)(x_2-x_1)}$$

...

Step k: In the previous step, we constructed a $(k-1)$ degree polynomial that interpolates $(x_0,y_0) \ldots (x_{k-1},y_{k-1})$. We will use this $P_{k-1}(x)$ and now define an n-degree polynomial $P_k(x)$ such that all of $(x_0,y_0), \ldots (x_k,y_k)$ are now interpolated.

Again $P_k(x) = P_{k-1}(x) + M_k(x)$ where M_k has degree $= k$

Now, we have:

For any $i \in \{0, 1, \dots, k-1\}$

$$\underbrace{P_k(x_i)}_{=y_i} = \underbrace{P_{k-1}(x_i)}_{=y_i} + M_k(x_i) \Rightarrow M_k(x_i) = 0 \quad \forall i = 0, 1, \dots, k-1$$

Thus, the k-degree polynomial M_k must have the form

$$M_k(x) = C_k (x - x_0) \cdots (x - x_{k-1})$$

Substituting $x \leftarrow x_k$ we get

$$y_k = P_k(x_k) = P_{k-1}(x_k) + C_k (x_k - x_0) \cdots (x_k - x_{k-1})$$

$$\Rightarrow C_k = \frac{y_k - P_{k-1}(x_k)}{\prod_{j=0}^{k-1} (x_k - x_j)}$$

Every polynomial $M_i(x)$ in this process is written as:

$$M_i(x) = C_i \cdot n_i(x) \quad \text{where } n_i(x) = \prod_{j=0}^{i-1} (x - x_j)$$

After n steps, the interpolating polynomial $P_n(x)$ is then written as:

$$P(x) = C_0 n_0(x) + C_1 n_1(x) + \cdots + C_n n_n(x).$$

where $n_0(x) = 1$

$n_1(x) = x - x_0$

$n_2(x) = (x-x_0)(x-x_1)$

$\vdots$

$n_k(x) = (x-x_0)(x-x_1)\cdots(x-x_{k-1})$

These are the Newton Polynomials (compare with the Lagrange polynomials $l_i(x)$). Note the x_i's are called the centers

Let us evaluate Newton interpolation, as we did with other methods:

- Cost of evaluating $P(x)$ for an arbitrary x: EASY

  This can be accellerated, using a modification of Horner's scheme

$$P(x) = c_0 + c_1(x-x_0) + c_2(x-x_0)(x-x_1) + \cdots$$

$$+ c_n(x-x_0)(x-x_1)\cdots(x-x_{n-1}) =$$

$$= c_0 + (x-x_0)\Big[c_1 + (x-x_1)\Big[c_2 + (x-x_2)\Big[c_3 + \cdots$$

$$+ c_{n-1} + (x-x_{n-1})c_n\Big]\Big]\cdots\Big]$$

e.g. for $n=3$

$$P(x) = c_0 + c_1(x-x_0) + c_2(x-x_0)(x-x_1) + c_3(x-x_0)(x-x_1)(x-x_2)$$

$$= c_0 + (x-x_0)\left[c_1 + (x-x_1)\left[c_2 + (x-x_2)\,c_3\right]\right]$$

- Cost of determining $P(x)$ (i.e. the coefficients $\{c_i\}$).

We saw one way of computing them, when describing the overall method. There is, however, another efficient and systematic way to compute them, called <u>divided differences</u>. A divided difference is a function defined over a set of sequentially indexed centers, e.g.

$$x_i, x_{i+1}, \ldots, x_{i+j-1}, x_{i+j}$$

The divided difference of these values is denoted by:

$$f[x_i, x_{i+1}, \ldots, x_{i+j-1}, x_{i+j}]$$

The value of this symbol is defined recursively, as follows:

For divided differences with 1 argument

$$f[x_i] := f(x_i) = y_i$$

With two arguments:

$$f[x_i, x_{i+1}] = \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i}$$

With three:

$$f[\underbrace{x_i, x_{i+1}}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i}$$

With j+1 arguments:

$$f[\underbrace{x_i, x_{i+1}, \ldots, x_{i+j-1}}, x_{i+j}] = \frac{f[x_{i+1}, \ldots, x_{i+j}] - f[x_i, \ldots, x_{i+j-1}]}{x_{i+j} - x_i}$$

The fact that makes divided differences so useful, is that $f[x_i, x_{i+1}, \ldots, x_{i+j-1}, x_{i+j}]$ can be shown to be the coefficient of the highest power of x in a polynomial that interpolates through

$$(x_i, y_i), (x_{i+1}, y_{i+1}), \ldots, (x_{i+j-1}, y_{i+j-1}), (x_{i+j}, y_{i+j})$$

Why is this useful ?

Remember, the polynomial that interpolates $(x_0, y_0), \ldots, (x_k, y_k)$ is

$$P_k(x) = \underbrace{P_{k-1}(x)}_{\text{highes power } = x^{k-1}} + \underbrace{c_k (x-x_0) \cdots (x-x_{k-1})}_{= c_k x^k + \text{lower powers.}}$$

Thus $c_k = f[x_0, x_1, x_2, \ldots, x_k]$!

or

$$\begin{aligned} P(x) = & f[x_0] \\ & + f[x_0, x_1](x-x_0) \\ & + f[x_0, x_1, x_2](x-x_0)(x-x_1) \\ & \vdots \\ & + f[x_0, x_1, \ldots, x_n](x-x_0) \cdots (x-x_{n-1}). \end{aligned}$$

So, if we can quickly evaluate the divided differences, we have determined $P(x)$!

Let is see a specific example 2|10|2011

$(x_0, y_0) = (-2, -27)$

$(x_1, y_1) = (0, -1)$

$(x_2, y_2) = (1, 0)$

$f[x_0] = y_0 = -27$

$f[x_1] = y_1 = -1$

$f[x_2] = y_2 = 0$

$$f[x_0, x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0} = \frac{-1 - (-27)}{0 - (-2)} = 13$$

$$f[x_1, x_2] = \frac{f[x_2] - f[x_1]}{x_2 - x_1} = \frac{0 - (-1)}{1 - 0} = 1$$

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} = \frac{1 - 13}{1 - (-2)} = -4$$

Thus $P(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1)$

$= -27 + 13(x+2) - 4(x+2) \cdot x$

The Newton interpolation method for the data points

$$(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$$

describes the n-degree polynomial interpolant as

$$\begin{aligned} p(x) = \; & c_0 \cdot \underbrace{1}_{n_0(x)} \\ & + c_1 \cdot \underbrace{(x-x_0)}_{n_1(x)} \\ & + c_2 \underbrace{(x-x_0)(x-x_1)}_{n_2(x)} \\ & \vdots \\ & + c_n \underbrace{(x-x_0)(x-x_1)\cdots(x-x_{n-1})}_{n_n(x)} \end{aligned}$$

The coefficients $\{c_i\}$ are computed using the method of divided differences: For a set of x-values $x_i, x_{i+1}, \ldots, x_j$ we define the divided difference as $f[x_i, \ldots, x_j]$.

The value of this symbol is recursively defined as:

$$\rightarrow \quad f[x_i] = y_i \quad \text{for } i = 0, \ldots, n$$

$$\rightarrow \quad f[x_i, x_{i+1}, \ldots, x_j] = \frac{f[x_{i+1}, \ldots, x_j] - f[x_i, x_{i+1}, \ldots x_{j-1}]}{x_j - x_i}$$

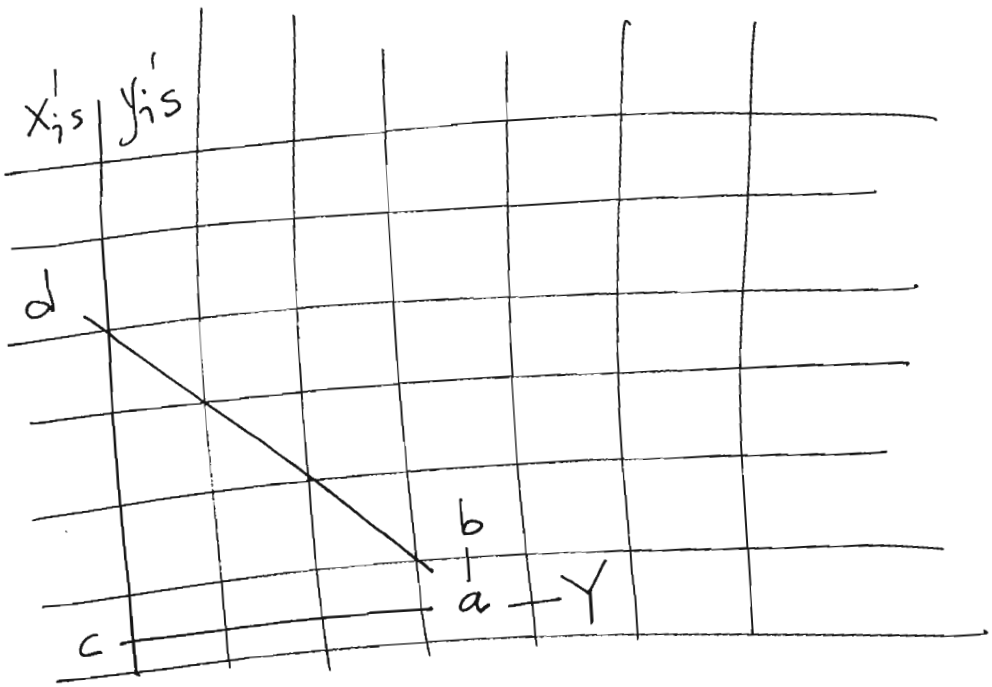
Once all relevant divided differences have been computed, the coefficients c_i are given by:

$$c_0 = f[x_0]$$
$$c_1 = f[x_0, x_1]$$
$$c_2 = f[x_0, x_1, x_2]$$
$$\vdots$$
$$c_n = f[x_0, x_1, \ldots, x_n]$$

Divided differences are usually tabulated as follows

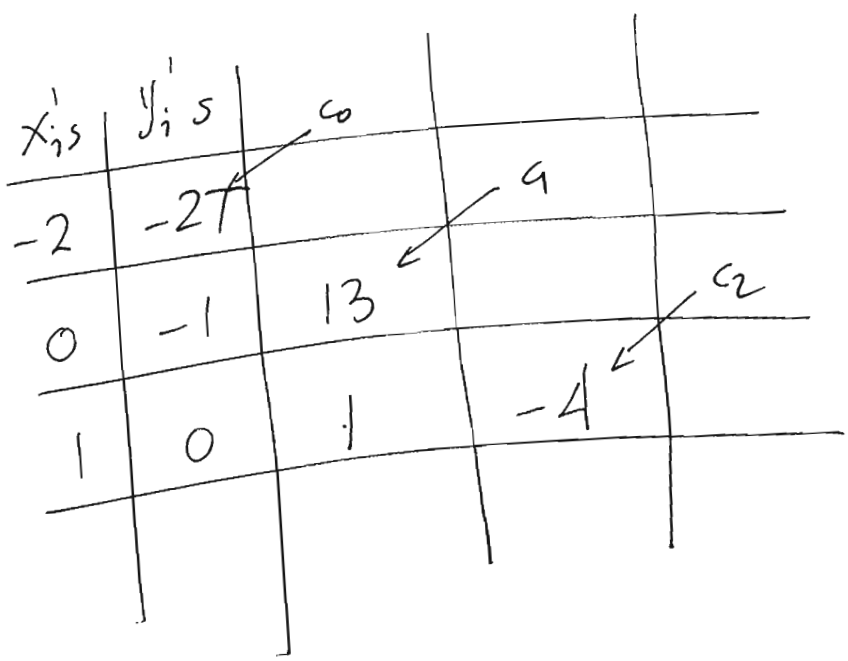
	$f[\cdot]$	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$	
x_0	$\boxed{f[x_0]}$	$\rightarrow c_0$		
x_1	$f[x_1]$	$\boxed{f[x_0,x_1]}$	$\rightarrow c_1$	
x_2	$f[x_2]$	$f[x_1,x_2]$	$\boxed{f[x_0,x_1,x_2]}$	$\rightarrow c_2$
x_3	$f[x_3]$	$f[x_2,x_3]$	$f[x_1,x_2,x_3]$	$\ldots$
x_4	$f[x_4]$	$f[x_3,x_4]$	$f[x_2,x_3,x_4]$	

The recursive definition can be implemented directly on the table as follows



$$Y = \frac{a-b}{c-d}$$

e.g $(x_0, y_0) = (-2, -27)$ $(x_1, y_1) = (0, -1)$ $(x_2, y_2) = (1, 0)$



Easy evaluation

e.g. $p(x) = c_0$
$+ c_1 (x - x_0)$
$+ c_2 (x - x_0)(x - x_1)$
$+ c_3 (x - x_0)(x - x_1)(x - x_2)$
$+ c_4 (x - x_0)(x - x_1)(x - x_2)(x - x_3) =$

$$= c_0 + (x - x_0)\Big[c_1 + (x - x_1)\Big[c_2 + (x - x_2)\Big[c_3 + (x - x_3)\,c_4\Big]\Big]\Big]$$

$Q_4(x)$
$Q_3(x)$
$Q_2(x)$
$Q_1(x)$
$Q_0(x)$

$$P(x) = Q_0(x)$$

recursively: $Q_n(x) = c_n$

$$Q_{n-1}(x) = c_{n-1} + (x - x_{n-1})\, Q_n(x)$$

The value of $P(x) = Q_0(x)$ can be evaluated (in linear time) by iterating this recurrence n times

We also have:

$$Q_{n-1}(x) = c_{n-1} + (x - x_{n-1})\, Q_n(x)$$

$$\Rightarrow Q_{n-1}'(x) = Q_n(x) + (x - x_{n-1}) Q_n'(x)$$

Thus, once we have computed all the Q_k's we can also compute all the derivatives too. Ultimately, $P'(x) = Q_0'(x)$.

We saw 3 methods for polynomial interpolation (Vandermonde, Lagrange, Newton). It is important to understand that all 3 methods compute (in theory) the same exact interpolant $p(x)$, just following diffent paths which may be better or worse from a computational perspective.

The question however, remains:

→ How accurate is this interpolation

or, in other words

→ How close is $p(x)$ to the "real" function $f(x)$!

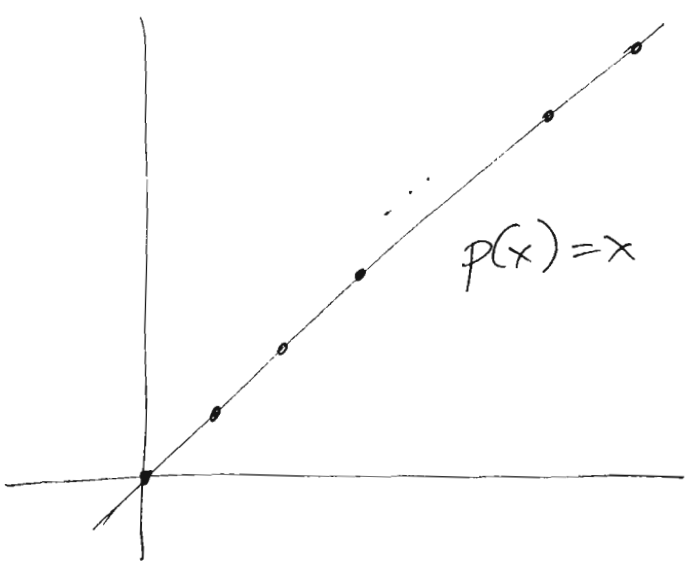
Example:

$(x_0, y_0) = (0,0)$

$(x_1, y_1) = (1,1)$

$\vdots$

$(x_n, y_n) = (n,n)$



Using Lagrange polynomials $P(x)\ (=x)$ is written as

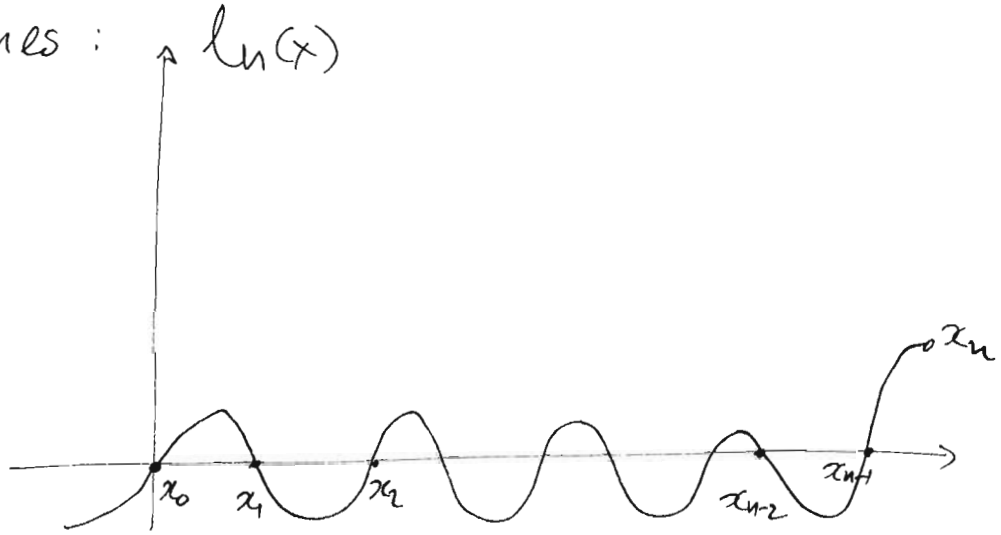
$$P(x) = \sum_{i=0}^{n} y_i \, l_i(x)$$

Let us "shift" y_n by a small amount δ. The new value is $y_n^* = y_n + \delta$. The updated interpolant $P^*(x)$ then becomes:

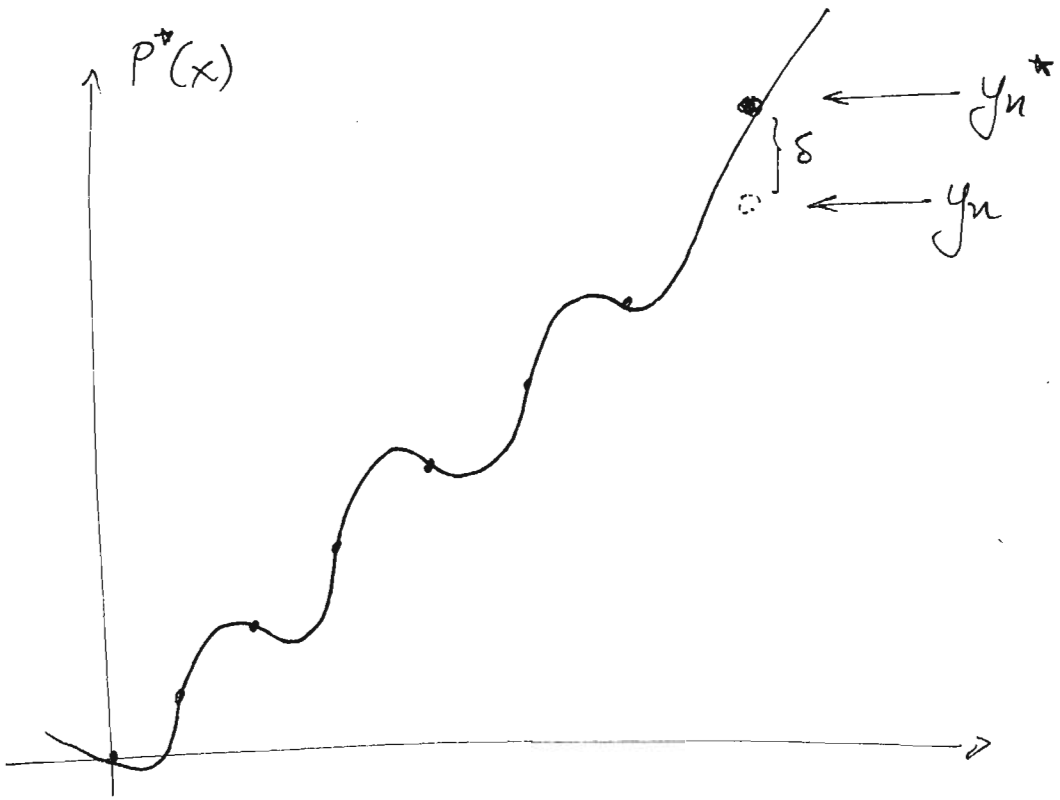
$$P^*(x) = \sum_{i=0}^{n-1} y_i \, l_i(x) + y_n^* \, l_n(x)$$

$$\text{Thus} \quad P^*(x) - P(x) = \delta \cdot l_n(x).$$

Note that l_n is a function that "oscillates" through zero several times:

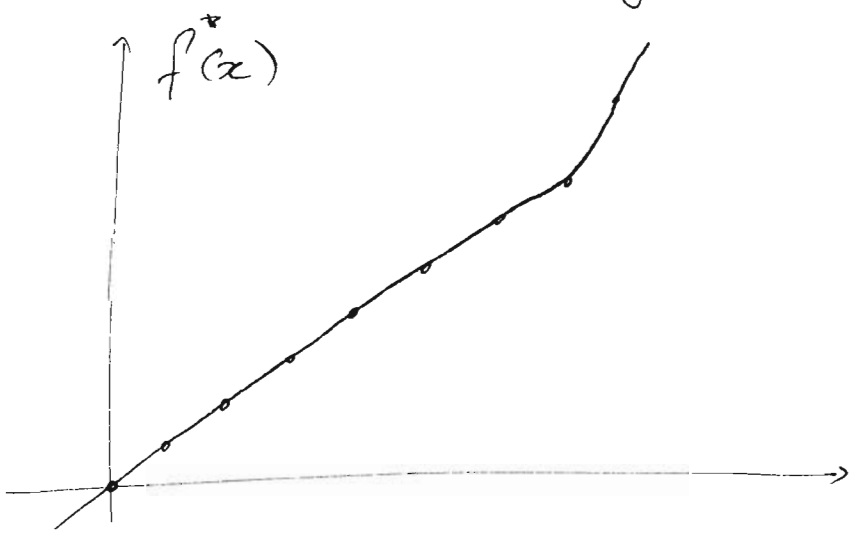


Thus P^* looks like



What we observe is that a <u>local</u> change in y-values caused a <u>global</u> (and drastic) change in $P(x)$.

Perhaps the "real" function f would have exhibited a more graceful and localized change, e.g.:



We will use the following theorem to compare the "real" function f being sampled, and the reconstructed interpolant $p(x)$

Theorem 1 Let:

- $x_1 < x_2 < \cdots < x_{n-1} < x_n$
- $y_k = f(x_k)$ $k = 1, 2, \ldots, n$, where f is a function which is n-times differentiable with continuous derivatives
- $P(x)$ is a polynomial that interpolates $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$

Then for any $x \in (x_1, x_n)$ there exists a $\theta = \theta(x) \in (x_1, x_n)$ such that

$$f(x) - P(x) = \frac{f^{(n)}(\theta)}{n!}(x-x_1)(x-x_2)\cdots(x-x_n)$$

This theorem may be difficult to apply directly, since:

→ θ is not known

→ θ changes with x

→ The n-th derivative $f^{(n)}(x)$ may not be fully known.

However, we can use it to derive a conservative bound:

Theorem 2 If $M = \max_{x \in [x_1, x_n]} |f^{(n)}(x)|$

and $h = \max_{1 \le i \le n-1} |x_{i+1} - x_i|$

then $|f(x) - p(x)| \le \frac{M h^n}{4n}$ for all $x \in [x_1, x_n]$.

How good is this, especially when we keep adding more and more data points (e.g. $n \to \infty$ and $h \to 0$)

This really depends on the higher order derivatives of $f(x)$... For example

$f(x) = \sin x \quad x \in [0, 2\pi]$

All derivatives of f are $\pm \sin x$ or $\pm \cos x$

thus $|f^{(k)}(x)| \le 1$ for <u>any</u> k

In this case $M = 1$, and as we add more (and denser) data points we have $|f(x) - p(x)| \le \frac{M h^n}{4n} \xrightarrow[n \to \infty]{h \to 0} 0$

For some functions, however, the values of $|f^{(k)}(x)|$ grow vastly as $k \to \infty$ (i.e. when we introduce additional points). e.g.:

$$f(x) = \frac{1}{x} \implies |f^{(n)}(x)| = n! \frac{1}{x^{n+1}}$$

$x \in (0.5, 1)$

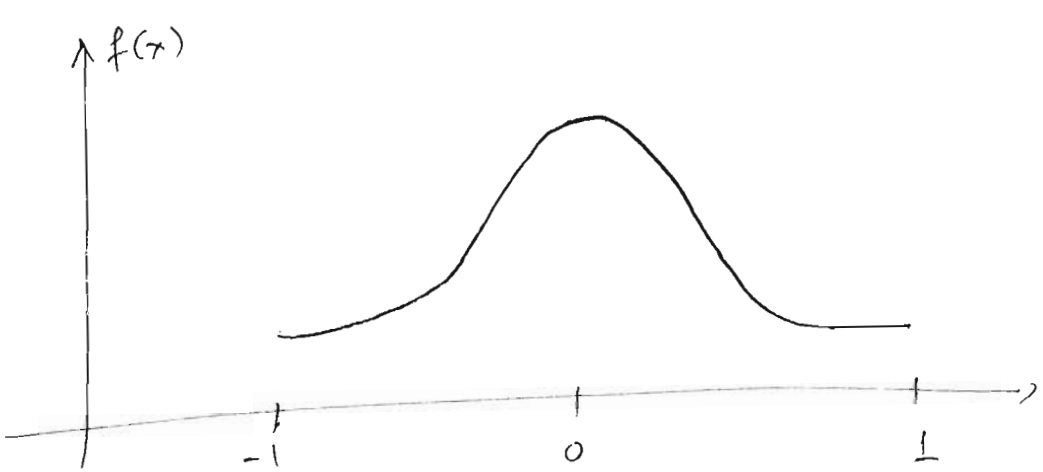
$$M_n = \max_{x \in (0.5,1)} |f^{(n)}(x)| = n! \cdot 2^n$$

in this case, as $n \to \infty$:

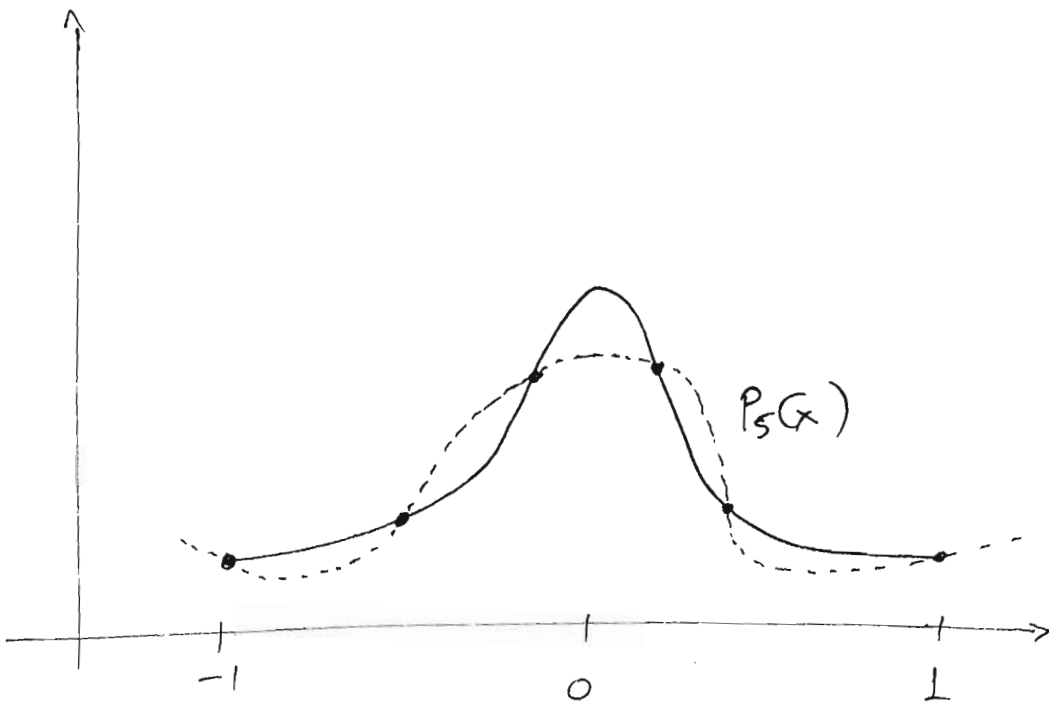
$$\frac{M_n h^n}{4n} = \frac{n! 2^n h^n}{4n} \xrightarrow[n \to \infty]{} +\infty \quad !$$

Another commonly cited counter-example is Runge's function:

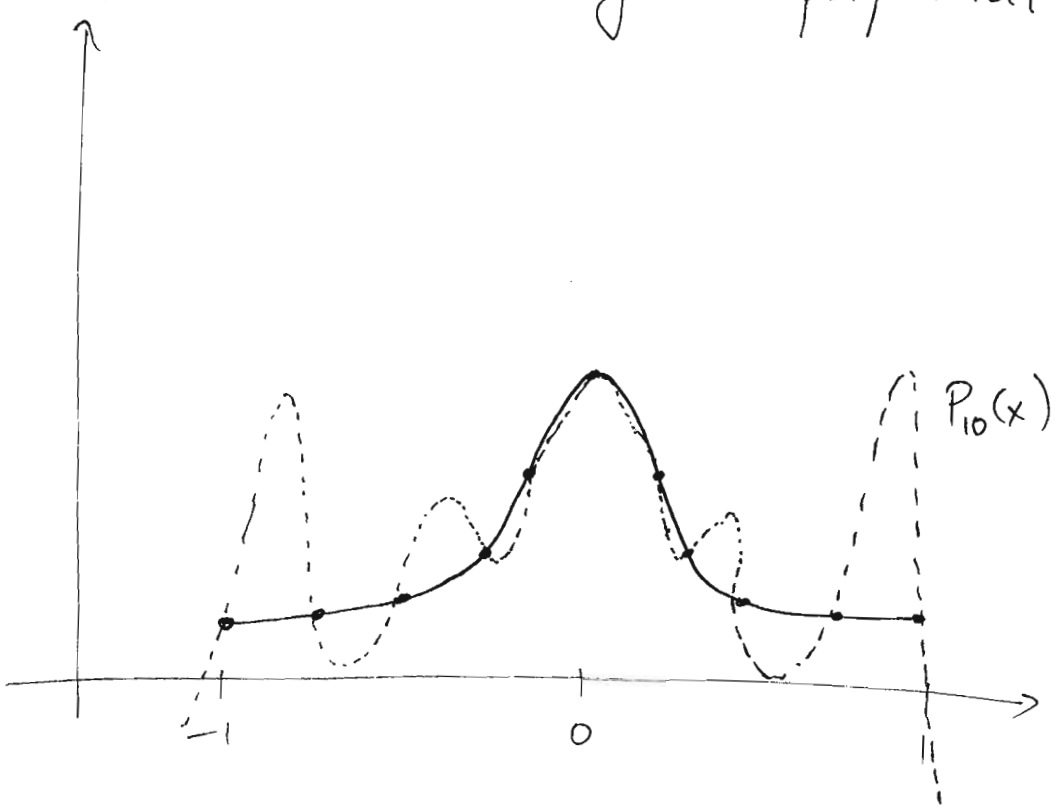
$$f(x) = \frac{1}{1 + 25x^2}$$



Approximation with a degree=5 polynomial



Approximation with a degree = 10 polynomial



Thus in this case the polynomials $P_n(x)$ do <u>not</u> uniformly converge to $f(x)$ as we add more points

A possible improvement stems from the following idea:

$$f(x) - P(x) = \frac{f^{(n)}(\theta)}{n!} \underbrace{(x-x_1)\cdots(x-x_n)}$$

↑ This we have no control over

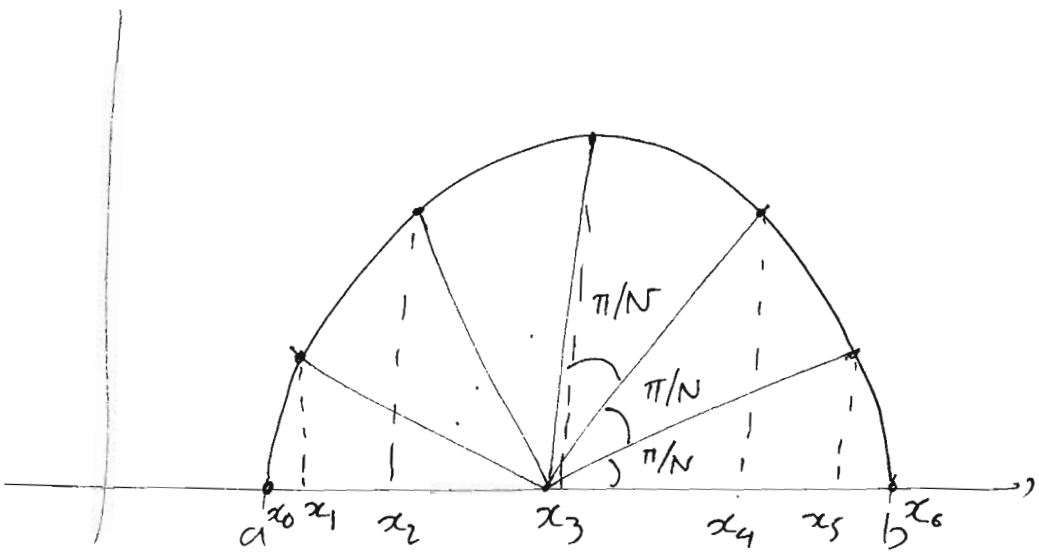
↑ Idea: Select the points $x_1, x_2, \ldots, x_n$ to minimize this product.

The value of the product $(x-x_1)\cdots(x-x_n)$ is minimized by selecting the x_i's as the <u>Chebyshev points</u>.

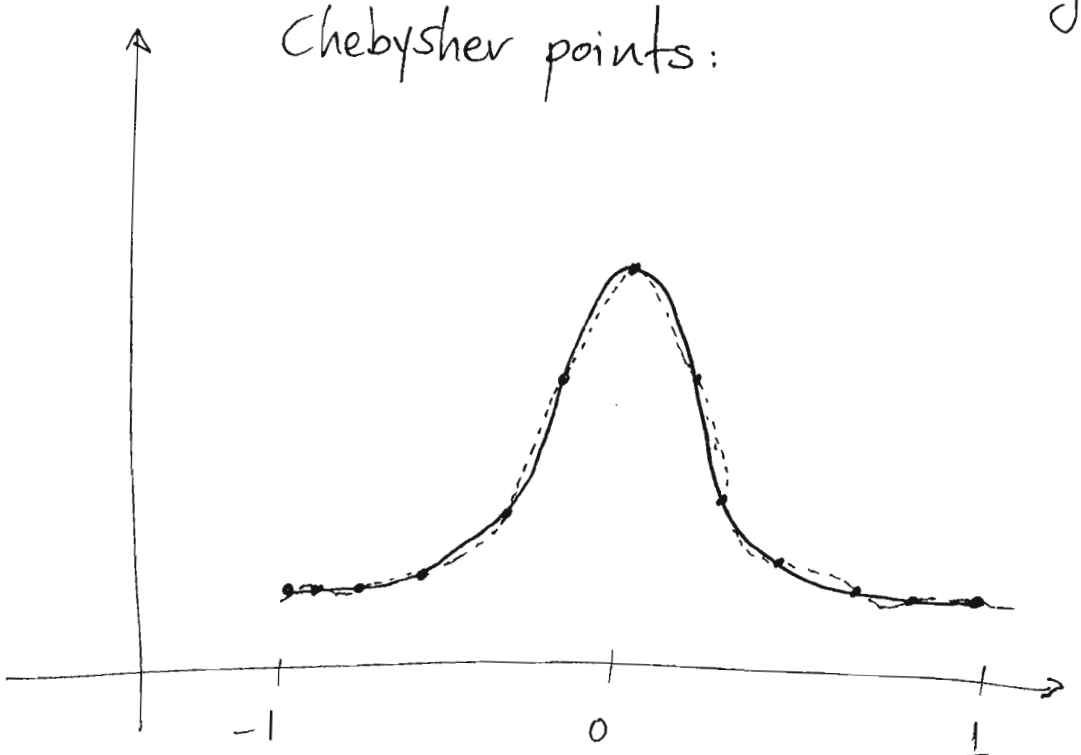
If the interpolation interval is $[a,b]$, the Chebyshev points are given by:

$$x_i = a + (b-a)\sin\left(\frac{i\pi}{N}\right) \qquad i = 0, 1, 2, \ldots, N$$

Graphically, these points are the projections on the x-axis of $(N+1)$ points located along the half circle with diameter the interval $[a,b]$, at equal arc-lengths:



Now, we can re-try Runge's function using Chebyshev points:



In fact, it is possible to show that, using Chebyshev points, we can guarrantee that

$$|f(x) - P(x)| \xrightarrow[n \to \infty]{} 0$$

provided that over $[a,b]$ both $f(x)$ and its derivative $f'(x)$ remain bounded (The benefit is that this condition does not place restrictions on higher-order derivatives of $f(x)$)

Although using Chebyshev points mitigates some of the drawbacks of high-order polynomial interpolants, this is still a non-ideal solution, since:

→ We do not always have the flexibility to pick the x_i's

→ Polynomial interpolants of high degree typically require more than $O(n)$ computational cost to construct

→ Local changes in the data points affect the entire extent of the interpolant.

A better remedy: Use piecewise polynomials

Assume that the x-values $\{x_i\}_{i=1}^{n}$ are sorted in ascending order:

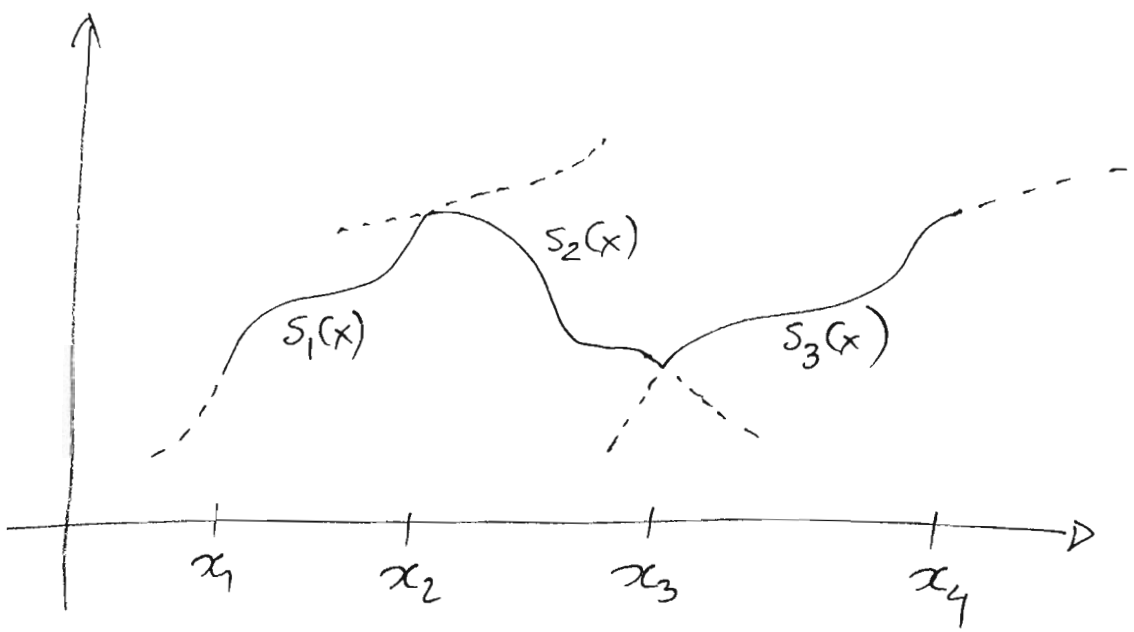
$$a = x_1 < x_2 < \cdots < x_{n-1} < x_n = b.$$

Define $I_k = [x_k, x_{k+1}]$

$$h_k = |x_{k+1} - x_k|$$

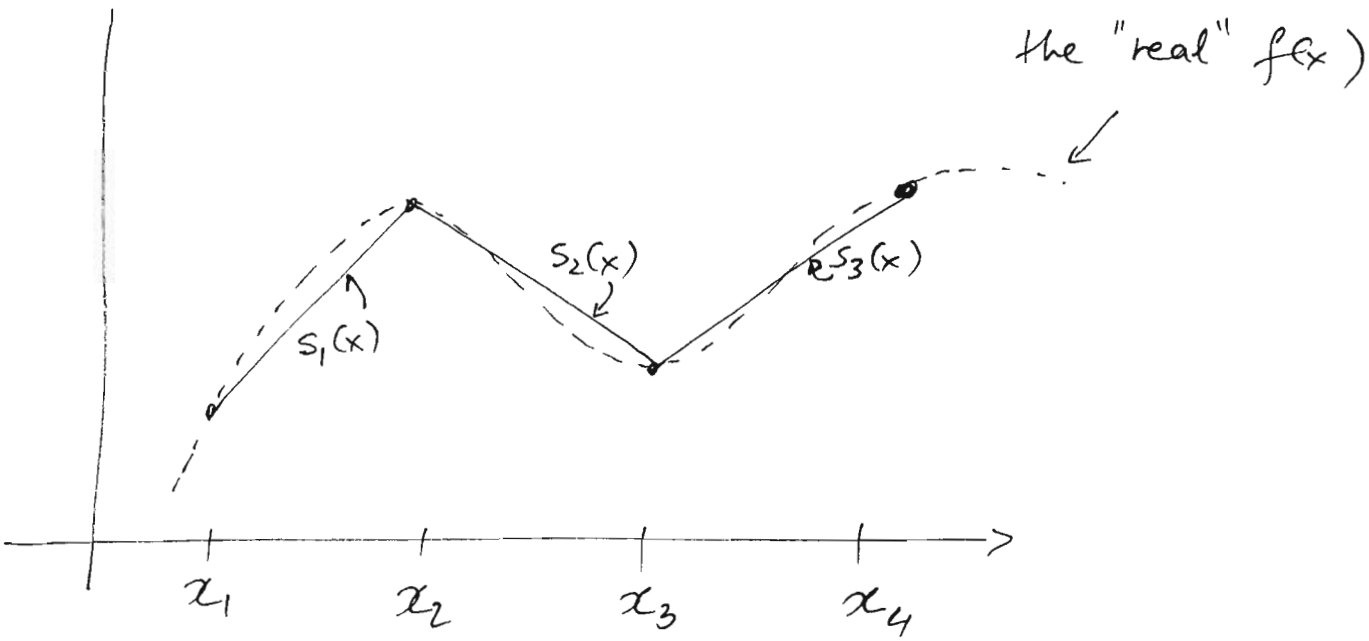
We also define the polynomials $s_1(x), s_2(x), \ldots, s_{n-1}(x)$ and use each of them to define the interpolant $s(x)$ at the respective interval I_k:

$$s(x) = \begin{cases} s_1(x), & x \in I_1 \\ s_2(x), & x \in I_2 \\ \vdots & \\ s_{n-1}(x), & x \in I_{n-1} \end{cases}$$



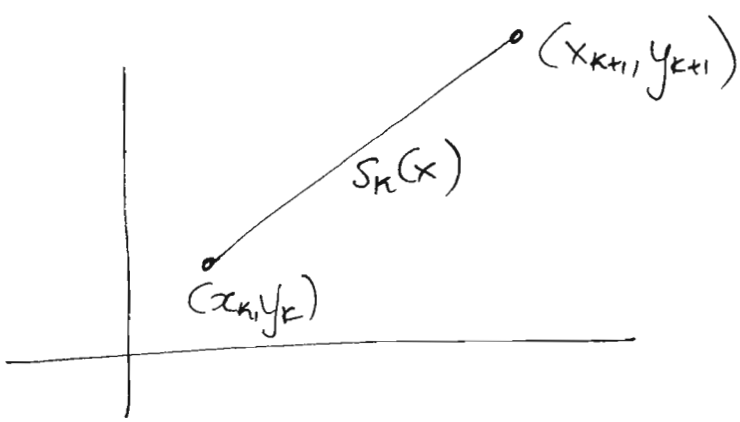
The benefit of using piecewise-polynomial interpolants, is that each $s_k(x)$ can be relatively low-order and thus non-oscillatory and easier to compute.

The simplest piecewise polynomial interpolant is a <u>piecewise linear</u> curve:



In this case every S_k can be written out explicitly as:

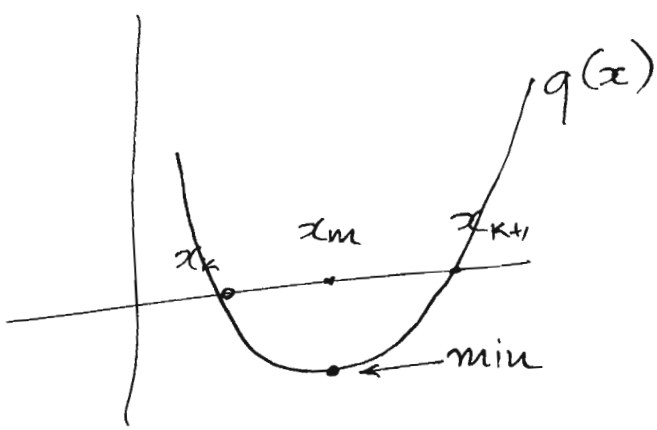
$$S_k(x) = y_k + \frac{y_{k+1} - y_k}{x_{k+1} - x_k}(x - x_k)$$



The next step is to examine the error $e(x) = f(x) - S_k(x)$ in the interval I_k. From the theorem we presented in the last lecture, we have that, for any $x \in I_k$ there is a $\theta_k = \theta(x_k)$ in I_k, such that:

$$e(x) = f(x) - S_k(x) = \frac{f''(\theta_k)}{2} \underbrace{(x - x_k)(x - x_{k+1})}_{q(x)} \qquad (1)$$

We are interested in the maximum value of $|q(x)|$ in order to determine a bound for the error. $q(x)$ is a quadratic function which crosses zero at x_k & x_{k+1}, thus the extreme value is obtained at the midpoint.



$$x_m = \frac{x_{k+1} + x_k}{2}$$

Thus $$|q(x)| \le |q(x_m)| = \left(\frac{x_{k+1} - x_k}{2}\right)^2 = \frac{h_k^2}{4}$$

for all $x \in I_k$.

Thus, using equation (1) we obtain:

$$|f(x) - S_k(x)| \leq \max_{x \in I_k} \left| \frac{f''(x)}{2} \right| \cdot \max_{x \in I_k} |(x - x_k)(x - x_{k+1})|$$

$$= \max_{x \in I_k} \left| \frac{f''(x)}{2} \right| \cdot \frac{h_k^2}{4}$$

$$\Rightarrow \quad |f(x) - S_k(x)| \leq \frac{1}{8} \max_{x \in I_k} |f''(x)| \cdot h_k^2$$

for all $x \in I_k$.

Additionally, if we assume all data points are equally spaced, i.e $h_1 = h_2 = \cdots = h_{n-1} = h \left(= \frac{b-a}{n-1} \right)$

we can additionally write:

$$|f(x) - s(x)| \leq \frac{1}{8} \max_{x \in [a,b]} |f''(x)| \cdot h^2$$

We often express the quantity on the right-hand side using the "infinity norm" of a given function, defined as

$$\|f\|_\infty = \max_{x \in [a,b]} |f(x)|$$

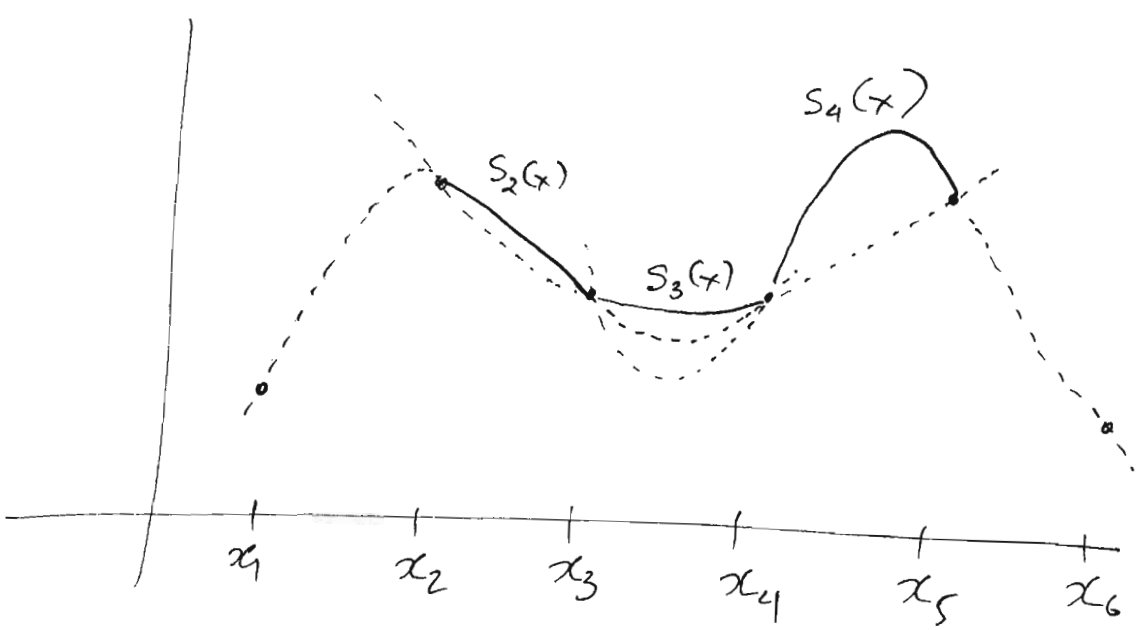
Thus, using this notation:

$$\boxed{|f(x) - s(x)| \leq \frac{1}{8} \|f''\|_\infty \cdot h^2}$$

Note that

- As $h \to 0$, the maximum discrepancy between f & s vanishes (proportionally to h^2)
- As we introduce more points, the quality of the approximation increases consistently, since the criterion above only considers the second derivative $f''(x)$ and not any higher order.

A possible improvement Piecewise cubic interpolation



In this approach each $s_k(x)$ is a <u>cubic</u> polynomial, designed such that it interpolates the 4 data points:

$$(x_{k-1}, y_{k-1}),\ (x_k, y_k),\ (x_{k+1}, y_{k+1}),\ (x_{k+7}, y_{k+2})$$

As we will see, the benefit is that the error can be made even smaller than with piecewise linear curves; the drawback is that (as seen in the last figure) $s(x)$ can develop "kinks" (or corners) where 2 pieces s_k & s_{k+1} are joined.

Error of piecewise cubics:

$$f(x) - s_k(x) = \frac{f''''(\theta_k)}{4!} \underbrace{(x-x_{k-1})(x-x_k)(x-x_{k+1})(x-x_{k+2})}_{q(x)}$$

An analysis similar to the linear case can show that $|q(x)| \le \frac{9}{16} \max^4\{h_{k-1}, h_k, h_{k+1}\}$

If we again assume $h_1 = h_2 = \cdots = h_k = h$, the error bound becomes:

$$|f(x)-s(x)| \le \frac{1}{24}\|f''''\|_\infty \frac{9}{16}\cdot h^4$$

$$\Rightarrow \boxed{f(x)-s(x) \le \frac{9}{384}\|f''''\|_\infty h^4}$$

The next possibility we shall consider, is a piecewise cubic curve

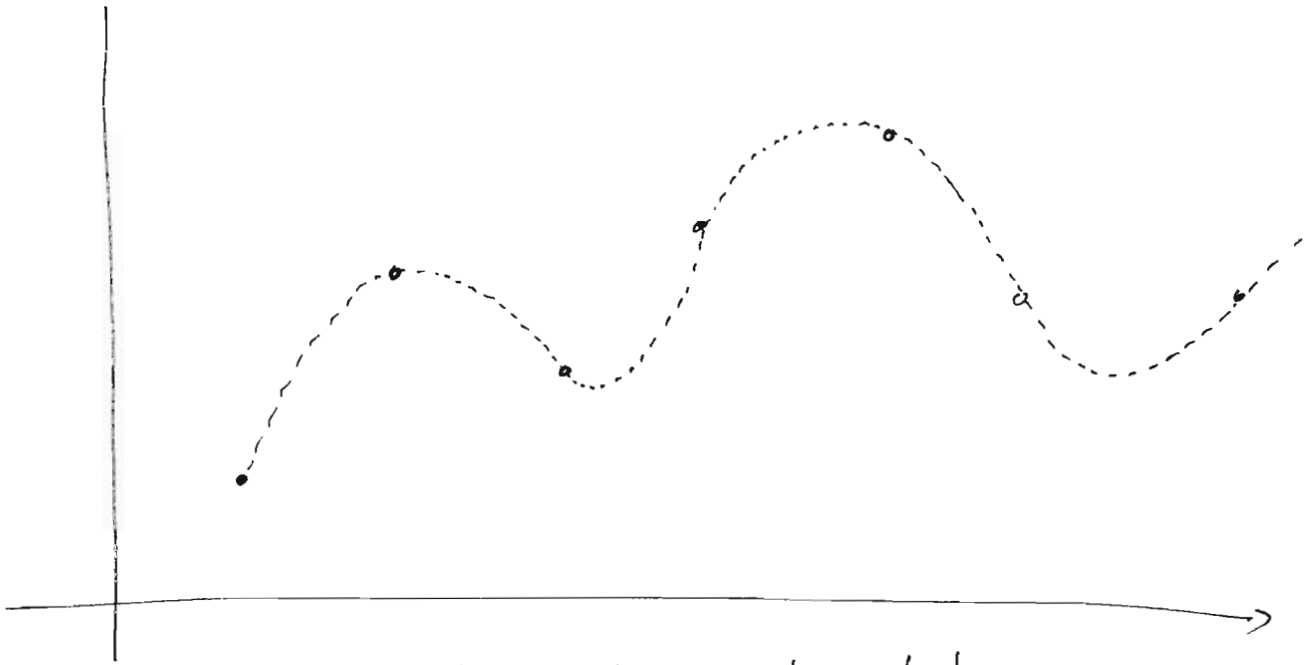
$$s(x) = \begin{cases} s_1(x) & x \in I_1 \\ \vdots & \\ s_{n-1}(x) & x \in I_{n-1} \end{cases}$$

where each $s_k(x) = a_3^{(k)}x^3 + a_2^{(k)}x^2 + a_1^{(k)}x + a_0^{(k)}$

and the coefficients $a_i^{(j)}$ are chosen as to <u>force</u> that the curve has continuous values, first and second derivatives:

$$s_k(x_{k+1}) = s_{k+1}(x_{k+1})$$
$$s_k'(x_{k+1}) = s_{k+1}'(x_{k+1})$$
$$s_k''(x_{k+1}) = s_{k+1}''(x_{k+1})$$

The curve constructed this way is called a <u>cubic spline</u> interpolant.



A cubic spline interpolation

Note the increased smoothness (continuity of values & derivatives) at the endpoints of each interval I_k.

The cubic spline

As always our goal in this interpolation task is to define a curve $s(x)$ which interpolates the n data points

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \quad (\text{where } x_1 < x_2 < \cdots < x_n)$$

In the fashion of piecewise polynomials we will define $s(x)$ as a <u>different</u> cubic polynomial $s_k(x)$ at each sub-interval $I_k = [x_k, x_{k+1}]$, i.e:

$$s(x) = \begin{cases} s_1(x) & x \in I_1 \\ s_2(x) & x \in I_2 \\ \vdots & \vdots \\ s_k(x) & x \in I_k \\ \vdots & \vdots \\ s_{n-1}(x) & x \in I_{n-1} \end{cases}$$

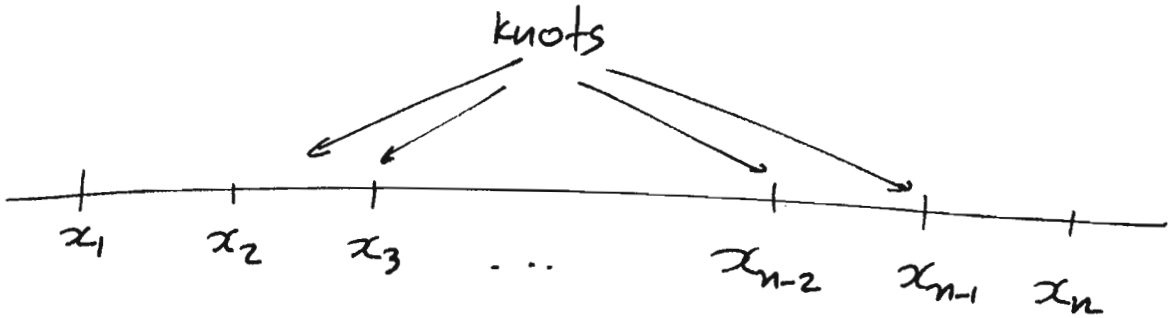
Each of the s_k's is a cubic polynomial

$$s_k(x) = \underline{a_0^{(k)}} + \underline{a_1^{(k)}} x + \underline{a_2^{(k)}} x^2 + \underline{a_3^{(k)}} x^3$$

Unknown coefficients

Since we have $(n-1)$ piecewise polynomials, in total we shall have to determine $4(n-1) = 4n-4$ unknown coefficients.

The points $(x_2, x_3, \ldots, x_{n-1})$ where the formula for $s(x)$ changes from one cubic polynomial (S_k) to another (S_{k+1}) are called <u>knots</u>



<u>Note</u>: In some textbooks, the extreme points x_1 & x_n are also included in the definition of what a knot is. We will stick with the definition we stated previously...

The piecewise polynomial interpolation method described as "cubic spline" also requires the neighboring polynomials S_k & S_{k+1} to be joined at x_{k+1} with a certain degree of smoothness. In detail:

The curve should be continuous: $S_k(x_{k+1}) = S_{k+1}(x_{k+1})$

The derivative (slope) should be continuous: $S_k'(x_{k+1}) = S_{k+1}'(x_{k+1})$

The 2nd derivative, as well: $S_k''(x_{k+1}) = S_{k+1}''(x_{k+1})$

(Note: If we force the next (3rd) derivative to match, this will force S_k & S_{k+1} to be exactly identical).

When determining the unknown coefficients $\{a_i^{(j)}\}$, each of these 3 smoothness constraints (for knots $k=2,3,\ldots,n-1$) needs to be satisfied, for a total of $3(n-2) = 3n-6$ constraint equations. We should not forget that we additionally want to <u>interpolate</u> all <u>n</u> data points i.e

$$s(x_i) = y_i \quad \text{for } i=1,2,\ldots,n \quad (n \text{ equation}$$

In total we have $(3n-6)+n = 4n-6$ total equations to satisfy, and $4n-4$ unknowns... consequently we will need <u>2</u> more equations to ensure that the unknown coefficients will be uniquely determined. Several plausible options exist on how to do that:

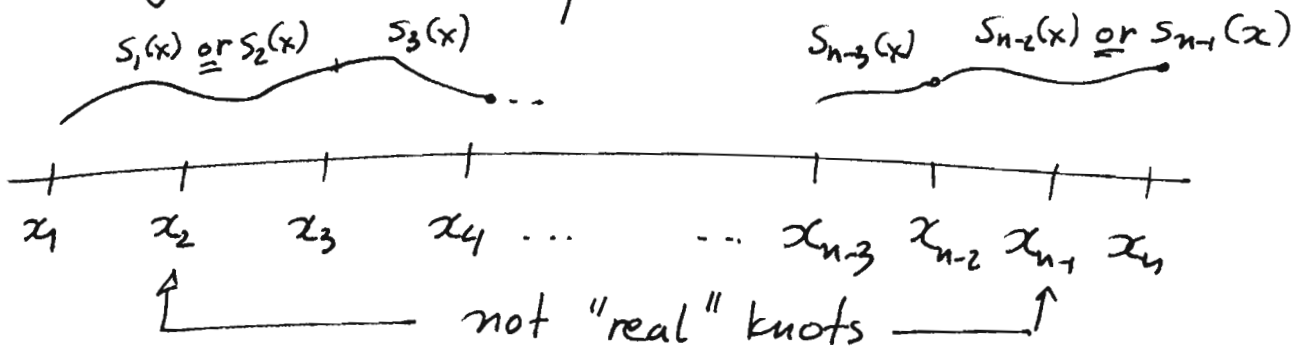
1. The "not-a-knot" approach: We stipulate that at the locations of the first (x_2) and last knot (x_{n-1}) the <u>third</u> derivative of $s(x)$ should also be continuous.

e.g.

$$S_1'''(x_2) = S_2'''(x_2)$$

and, $$S_{n-2}'''(x_{n-1}) = S_{n-1}'''(x_{n-1})$$

As we discussed before, these 2 additional constraints will effectively cause $S_1(x)$ to be identical with $S_2(x)$, and $S_{n-2}(x)$ to coincide with $S_{n-1}(x)$. In this sense, x_2 and x_{n-1} are no longer "knots" in the sense that the formula for $S(x)$ "changes" at these points: (hence the name).



2. Complete spline: If we have access to the derivative f' of the function being sampled by the y_i's (i.e $y_i = f(x_i)$), we can formulate the 2 additional constraints as:

$$s'(x_1) = f'(x_1)$$
$$s'(x_n) = f'(x_n)$$

$$\left(\text{or} \quad \begin{array}{l} s_1'(x_1) = f'(x_1) \\ s_{n-1}'(x_n) = f'(x_n) \end{array} \right)$$

Note that, qualitatively, using the complete spline approach is a better utilization of the flexibility of the spline curve in matching yet one more property of f; in contrast, the not-a-knot makes the spline "less flexible" by two degrees of freedom, in order to obtain a unique solution. However, we cannot always assume knowledge of f'

An additional 2 methodologies are:

3. The <u>natural cubic spline</u>: we use the following 2 constraints

$$s''(x_1) = 0$$
$$s''(x_n) = 0$$

Thus, $s(x)$ reaches the endpoints looking like a straight line (instead of a curved one).

4. Periodic spline:

$$s'(x_1) = s'(x_n)$$
$$s''(x_1) = s''(x_n)$$

This is useful when the underlying function f is also known to be periodical over $[a, b]$.

We will not discuss the analytic derivation of the cubic spline coefficients; instead we describe how to access this functionality within matlab, through the built-in functions spline and ppval

- The function spline is called as

$$s = \text{spline}(x, y)$$

x: The vector containing the x_i values

$$x = (x_1, x_2, \ldots, x_n)$$

y: A corresponding vector of y_i values

s: A specially encoded result, containing the necessary information for the generated spline. This is only used indirectly, by other MATLAB functions

The spline function can be used to implement either the not-a-knot, or the complete spline method.

→ If length(x) = length(y), then y is assumed to contain the values $y = (y_1, y_2, \ldots, y_n)$ and the spline is generated using the not-a-knot approach.

→ To implement the complete spline approach, we provide 2 additional values in the vector y, starting with $y_1' = f'(x_1)$ and ending with $y_n' = f'(x_n)$, i.e.

$$y = (y_1', y_1, y_2, \ldots, y_{n-2}, y_{n-1}, y_n, y_n')$$

In this case we obviously have

$$\text{length}(y) = \text{length}(x) + 2.$$

This triggers MATLAB to implement the complete spline approach.

The ppval function takes in the information encoded in $\leq$ (the output of spline) and evaluates the spline curve at a number of different locations.

Syntax $\quad v = \text{ppval}(s, u)$

u : A vector of m new x-locations where we want the spline to be interpolated/evaluated

$$u = (u_1, u_2, \ldots, u_m).$$

v : The corresponding y-values of these u_i locations

$$v = (v_1, v_2, \ldots, v_m).$$

Example:

```
x = 0 : pi/5 : 2*pi ;
y = sin (x) ;
s = spline (x,y) ;
u = 0 : pi/100 : 2*pi ;
v = ppval (s,u) ;
plot (u,v) ;
plot (x,y,u,v) ;
w = sin (u)
plot (u,v,u,w) ;
```

Error analysis:

for simplicity, we will again assume that

$$h_1 = h_2 = \cdots = h_{n-1} = h \qquad (h_k = x_{k+1} - x_k)$$

For the not-a-knot method, we have

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

The "approximate" inequality is used because the interpolation error can be slightly larger near the endpoints of the interval $[a,b]$.

This is a very comparable result with the (non-smooth) piecewise cubic polynomial method:

$$|f(x)-s(x)| \leq \frac{9}{384} \|f''''\|_\infty h^4$$

Note though that the computation of the piecewise cubic method was <u>very local</u> and simple (every interval could be independently evaluated) while the computation of the coefficients of the cubic spline is more elaborate.

<u>Hermite spline</u>

Let us assume a number of x-locations

$$x_1 < x_2 < \cdots < x_{n-1} < x_n$$

and let us make the hypothesis that we know both f <u>and</u> f' at every location x_i. We denote these values by

$$\left.\begin{aligned} y_i &= f(x_i) \\ y_i' &= f'(x_i) \end{aligned}\right\} \quad i = 1, 2, \ldots, n$$

Cubic Hermite splines

In the previous lecture, we introduced a different approach to piecewise-cubic polynomial interpolation. In particular, given n x-values (in ascending order)

$$x_1 < x_2 < \cdots < x_{n-1} < x_n$$

and n associated y-values (sampled from a function $f(x)$)

$$y_1, y_2, \ldots, y_{n-1}, y_n \quad , \text{ where } y_k = f(x_k)$$

and assume we also know the derivative $f'(x)$ at the same locations, denoted by:

$$y_1', y_2', \ldots, y_{n-1}', y_n' \quad , \text{ where } y_k' = f'(x_k)$$

As with other methods based on piecewise polynomials, we construct the interpolant as

$$s(x) = \begin{cases} s_1(x) & x \in I_1 \\ s_2(x) & x \in I_2 \\ \vdots & \vdots \\ s_{n-1}(x) & x \in I_{n-1} \end{cases} \qquad \text{where } I_k = [x_k, x_{k+1}].$$

In this case, each individual $s_k(x)$ is constructed to match both the function values y_k, y_{k+1} as well as the derivatives y_k', y_{k+1}' at the endpoints of I_k.

In detail:

$$\left.\begin{aligned} s_k(x_k) &= y_k \\ s_k(x_{k+1}) &= y_{k+1} \\ s_k'(x_k) &= y_k' \\ s_k'(x_{k+1}) &= y_{k+1}' \end{aligned}\right\} \qquad (*)$$

Since $s_k(x) = a_3x^3 + a_2x^2 + a_1x + a_0$ has 4 unknown coefficients, the 4 equations $(*)$ could uniquely define the appropriate values of $a_0 \dots a_3$ (as they do!).

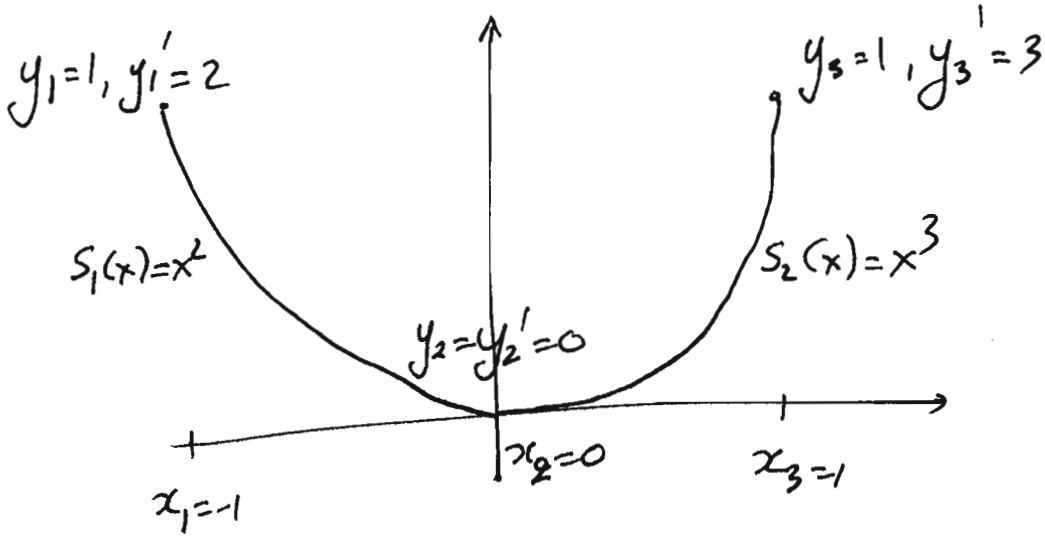
Note that, since we have

$$s_k(x_{k+1}) = y_{k+1} = s_{k+1}(x_{k+1})$$

and $$s_k'(x_{k+1}) = y_{k+1}' = s_{k+1}'(x_{k+1})$$

The resulting interpolant $s(x)$ is <u>continuous</u> with continuous derivatives (e.g. a C^1 function).

However, we do not strictly enforce that the 2nd derivative should be continuous, and in fact it generally will not be:



In this case $S_1''(0) = 2$

while $S_2''(0) = 0$

The most straightforward method for determining the coefficients of $S_k(x) = a_3x^3 + a_2x^2 + a_1x + a_0$ mimics the Vandermonde approach for polynomial interpolation:

$$
\left.
\begin{aligned}
S_k(x_k) = y_k &\Rightarrow a_3 x_k^3 + a_2 x_k^2 + a_1 x_k + a_0 = y_k \\
S_k(x_{k+1}) = y_{k+1} &\Rightarrow a_3 x_{k+1}^3 + a_2 x_{k+1}^2 + a_1 x_{k+1} + a_0 = y_{k+1} \\
S_k'(x_k) = y_k' &\Rightarrow a_3 \cdot 3x_k^2 + a_2 \cdot 2x_k + a_1 = y_k' \\
S_k'(x_{k+1}) = y_{k+1}' &\Rightarrow a_3 \cdot 3x_{k+1}^2 + a_2 \cdot 2x_{k+1} + a_1 = y_{k+1}'
\end{aligned}
\right\} \Rightarrow
$$

$$
\Rightarrow \begin{bmatrix} x_k^3 & x_k^2 & x_k & 1 \\ x_{k+1}^3 & x_{k+1}^2 & x_{k+1} & 1 \\ 3x_k^2 & 2x_k & 1 & 0 \\ 3x_{k+1}^2 & 2x_{k+1} & 1 & 0 \end{bmatrix} \begin{bmatrix} a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \begin{bmatrix} y_k \\ y_{k+1} \\ y_k' \\ y_{k+1}' \end{bmatrix}
$$

The 2nd method attempts to mimic the Lagrange interpolation approach, where we wrote

$$
p(x) = y_0 l_0(x) + y_1 l_1(x) + \cdots + y_n l_n(x)
$$

where $l_i(x_j) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$

What if we could do something similar, here?

Can we write

$$S_k(x) = y_k q_{00}(x) + y_{k+1} q_{01}(x) + y'_k q_{10}(x) + y'_{k+1} q_{11}(x)$$

Yes, if we have:

$$\begin{array}{|l|l|l|l|}
q_{00}(x_k) = 1 & q_{10}(x_k) = 0 & q_{01}(x_k) = 0 & q_{11}(x_k) = 0 \\
q_{00}(x_{k+1}) = 0 & q_{10}(x_{k+1}) = 0 & q_{01}(x_{k+1}) = 1 & q_{11}(x_{k+1}) = 0 \\
q'_{00}(x_k) = 0 & q'_{10}(x_k) = 1 & q'_{01}(x_k) = 0 & q'_{11}(x_k) = 0 \\
q'_{00}(x_{k+1}) = 0 & q'_{10}(x_{k+1}) = 0 & q'_{01}(x_{k+1}) = 0 & q'_{11}(x_{k+1}) = 1
\end{array}$$

(All q_{ij}'s are cubic polynomials!)

In the special case where $x_k = 0$, $x_{k+1} = 1$, these functions are symbolized with $h_{ij}(x)$ and called the canonical Hermite basis functions. Thus, in that case:

$$s(x) = y_0 h_{00}(x) + y_1 h_{01}(x) + y'_0 h_{10}(x) + y'_1 h_{11}(x)$$

In this case we can either solve a 4×4 system for the coefficients of each $h_{ij}(x)$, or construct it using simple algebraic arguments, e.g.

$h_{11}(0) = h_{11}'(0) = 0 \Rightarrow x^2$ is a factor of $h_{11}(x)$

$h_{11}(1) = 0 \Rightarrow x-1$ is a factor of $h_{11}(x)$

i.e. $h_{11}(x) = C\,x^2(x-1) = C\,(x^3-x^2)$

$h_{11}'(x) = C\,(3x^2-2x)$

$1 = h_{11}'(1) = C\cdot(3-2) = C \Rightarrow \boxed{C=1}$

Thus $h_{11}(x) = x^3-x^2$

The 4 basis polynomials are similarly derived to be:

$$h_{00}(x) = 2x^3-3x^2+1$$
$$h_{10}(x) = x^3-2x^2+x$$
$$h_{01}(x) = -2x^3+3x^2$$
$$h_{11}(x) = x^3-x^2$$

You need not memorize these...

In the more general case where $I_k = [x_k, x_{k+1}]$ (instead of $[0,1]$) we can obtain the basis polynomials using a change of variable $t = \frac{x - x_k}{x_{k+1} - x_k}$ as follows

$$S_k(x) = y_k \underbrace{h_{00}(t)}_{q_{00}(x)} + y_{k+1} \underbrace{h_{01}(t)}_{q_{01}(x)} + y'_k \underbrace{(x_{k+1} - x_k) h_{10}(t)}_{q_{10}(x)}$$

$$+ y'_{k+1} \underbrace{(x_{k+1} - x_k) h_{11}(t)}_{q_{11}(x)}$$

The last (and, quite common) approach for generating the Hermite spline is using tools similar to Newton interpolation.

Remember, when interpolating through $(x_0, y_0), \ldots, (x_3, y_3)$ we obtain:

$$\begin{aligned} p(x) = {} & f[x_0] \cdot 1 \\ & + f[x_0, x_1] \cdot (x - x_0) \\ & + f[x_0, x_1, x_2] \cdot (x - x_0)(x - x_1) \\ & + f[x_0, x_1, x_2, x_3] (x - x_0)(x - x_1)(x - x_2) \end{aligned}$$

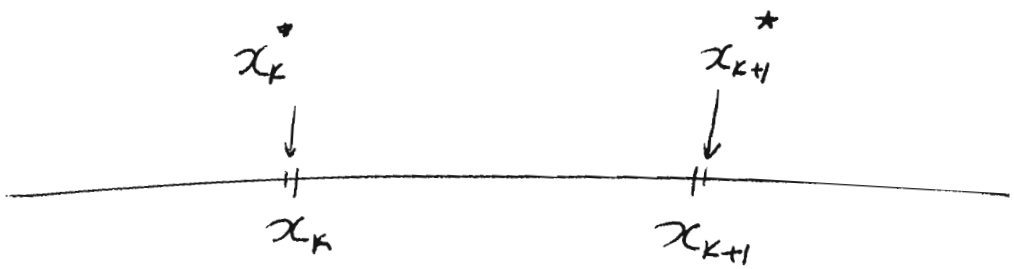
The idea is as follows:

Perform newton interpolation through the points

$$(x_k^*, y_k^*) \;,\; (x_k, y_k) \;,\; (x_{k+1}, y_{k+1}) \;,\; (x_{k+1}^*, y_{k+1}^*)$$

where $x_k^* = x_k - \varepsilon$

$x_{k+1}^* = x_{k+1} + \varepsilon$



We will compute this interpolant using the Newton method, and ultimately set $\varepsilon \to 0$ such that x_k^* converges onto x_k, and x_{k+1}^* respectively onto x_{k+1}.

Thus:

$$S_k(x) = f[x_k^*]$$
$$+ f[x_k^*, x_k]\,(x - x_k^*)$$
$$+ f[x_k^*, x_k, x_{k+1}]\,(x - x_k^*)(x - x_k)$$
$$+ f[x_k^*, x_k, x_{k+1}, x_{k+1}^*]\,(x - x_k^*)(x - x_k)(x - x_{k+1}).$$

Taking the limit $\varepsilon \to 0$

$$S_K = \left(\lim_{x_K^* \to x_K} f[x_K^*] \right) \cdot$$

$$+ \left(\lim_{x_K^* \to x_K} f[x_K^*, x_K] \right) (x - x_K)$$

$$+ \left(\lim_{x_K^* \to x_K} f[x_K^*, x_K, x_{K+1}] \right) (x - x_K)^2$$

$$+ \left(\lim_{\substack{x_K^* \to x_K \\ x_{K+1}^* \to x_{K+1}}} f[x_K^*, x_K, x_{K+1}, x_{K+1}^*] \right) (x - x_K)^2 (x - x_{K+1})$$

We use the shorthand notation

$$f[x_K, x_K] := \lim_{x_K^* \to x_K} f[x_K^*, x_K]$$

and construct the finite difference table as usual.

? x_k^*	$f[x_k^*]$?			
x_k	$f[x_k]$	$f[x_k^*, x_k]$?		
x_{k+1}	$f[x_{k+1}]$	$f[x_k, x_{k+1}]$	$f[x_k^*, x_k, x_{k+1}]$	
? x_{k+1}^*	$f[x_{k+1}^*]$?	$f[x_{k+1}, x_{k+1}^*]$?	$f[x_k, x_{k+1}, x_{k+1}^*]$	$f[x_k^*, x_k, x_{k+1}, x_{k+1}^*]$

when $\varepsilon \to 0$, the quantities in this table that involve x_k^* or x_{k+1}^* may need to be expressed through limits. e.g.

$$x_k^* \longrightarrow x_k$$

$$x_{k+1}^* \longrightarrow x_{k+1}$$

$$f[x_k^*] = y_k^* \longrightarrow y_k$$

$$f[x_{k+1}^*] = y_{k+1}^* \longrightarrow y_{k+1}$$

$$f[x_k^*, x_k] = \frac{f[x_k] - f[x_k^*]}{x_k - x_k^*} \xrightarrow[x_k^* \to x_k]{} f'(x_k) = y_k' \ !$$

$$f[x_{k+1}, x_{k+1}^*] = \frac{f[x_{k+1}^*] - f[x_{k+1}]}{x_{k+1}^* - x_{k+1}} \xrightarrow[x_{k+1}^* \to x_{k+1}]{} f'(x_{k+1}) = y_{k+1}' \ .$$

Thus, the table gets filled as follows

x_K	y_K			
x_K	y_K	y_K'		
x_{K+1}	y_{K+1}	$f[x_K, x_{K+1}]$	$f[x_K^*, x_K, x_{K+1}]$	
x_{K+1}	y_{K+1}	y_{K+1}'	$f[x_K, x_{K+1}, x_{K+1}^*]$	$f[x_K^*, x_K, x_{K+1}, x_{K+1}^*]$

The remaining divided differences are computed normally using the recursive definition.

Often times we skip the "stars" on x_K's and use the simpler notation $f[x_K, x_K]$, $f[x_K, x_K, x_{K+1}, x_{K+1}]$ etc.