

# Computing interpolants by hand

- Suppose we have data at  $n$  points and we have fit a polynomial
- How can we do this fit efficiently?
- Fit for one point is  $y=y_1$
- Fit for two points can be written as  $y=a(x-x_1)+b(x-x_2)$
- Fit for three points can be written as

$$y=a(x-x_1)(x-x_2)+b(x-x_1)(x-x_3)+c(x-x_2)(x-x_3)$$

- And so on ...
- Advantage: each coefficient can be calculated independent of others
  - Why?
  - What is the form of the coefficient computed?

# Lagrange and Newton forms for interpolations

- Lagrange and Newton modified these forms further to conveniently compute polynomials

# Lagrange Polynomials

- Summation of terms, such that:
  - Equal to  $f()$  at a data point.
  - Equal to zero at all other data points.
  - Each term is a  $n^{\text{th}}$ -degree polynomial

$$p_n(x) = \sum_{i=0}^n L_i(x) f(x_i)$$

$$L_i(x) = \prod_{k=0, k \neq i}^n \frac{(x - x_k)}{(x_i - x_k)}$$

$$L_i(x_j) = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

# Newton Interpolation

- Consider our data set of  $n+1$  points  $y_i=f(x_i)$  at  $x_0, x_1, \dots, x_i, \dots, x_n$ :  $x_n > x_0$
- Since  $p_n(x)$  is the **unique** polynomial  $p_n(x)$  of order  $n$ , write it:

$$p_n(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + \dots + b_n(x - x_0)(x - x_1) \cdots (x - x_{n-1})$$
$$b_0 = f(x_0)$$
$$b_1 = f[x_1, x_0] = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$
$$b_2 = f[x_2, x_1, x_0] = \frac{f[x_2, x_1] - f[x_1, x_0]}{x_2 - x_0}$$
$$\vdots$$
$$b_n = f[x_n, x_{n-1}, \dots, x_0] = \frac{f[x_n, \dots, x_1] - f[x_{n-1}, \dots, x_0]}{x_n - x_0}$$

- $f[x_i, x_j]$  is a **first divided difference**
- $f[x_2, x_1, x_0]$  is a **second divided difference**, etc.
- Efficient way of adding points to the interpolation!
- Used to fit data to a table

# Newton Interpolation

- Example
- Let  $x_0=1, f(x_0)=-5$  ;  $x_1=2, f(x_1)=-3$ ;  
 $x_2=3, f(x_2)=2$ ;  $x_3=4, f(x_3)=4$ .
- Build divided difference table
- $f[x_0]=-5$
- $f[x_1]=-3$   $f[x_0,x_1]=2$
- $f[x_2]=2$   $f[x_1,x_2]=5$   $f[x_0,x_1,x_2]=3/2$
- $f[x_3]=4$   $f[x_2,x_3]=2$   $f[x_1,x_2,x_3]=-3/2$   
 $f[x_0,x_1,x_2,x_3]=(3/2+3/2)/(1-4)=-1$
- To compute Newton form we need  $f[x_0], f[x_0,x_1], f[x_0,x_1,x_2], f[x_0,x_1,x_2,x_3]$

# Newton form

- Interpolation

$$P(x) = f[x_0] + f[x_0, x_1] (x - x_0) + f[x_0, x_1, x_2] (x - x_0)(x - x_1) + f[x_0, x_1, x_2, x_3] (x - x_0)(x - x_1)(x - x_2)$$

$$P(x) = -5 + 2(x - 1) + \frac{3}{2}(x - 1)(x - 2) - (x - 1)(x - 2)(x - 3)$$

# Error

- Define the error term as:

$$\varepsilon_n(x) = f(x) - p_n(x)$$

- If  $f(x)$  is an  $n^{\text{th}}$  order polynomial  $p_n(x)$  is of course exact.
- Otherwise, since there is a perfect match at  $x_0, x_1, \dots, x_n$
- This function has at least  $n+1$  roots at the interpolation points.

$$\therefore \varepsilon_n(x) = (x - x_0)(x - x_1) \cdots (x - x_n)h(x)$$

# Interpolation Errors

- Suppose we want to measure error at a point  $x$
- To make polynomial go through  $x$ , add to existing polynomial divided difference term.
- This is the error we make using existing polynomial

$$x \notin \{x_0, x_1, \dots, x_n\}$$

$$\varepsilon_n(x) = f(x) - p_n(x) = f[x_0, x_1, \dots, x_n, x] \prod_{i=0}^n (x - x_i)$$

- Comparing with Taylor series

$$f[x_0, x_1, \dots, x_n] = \frac{1}{n!} f^{(n)}(\xi)$$

# Interpolation Errors

$$\varepsilon_n(x) = f(x) - p_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{i=0}^n (x - x_i)$$

$$x \in [a, b], \xi \in (a, b)$$

- Looks a bit like Taylor series remainder
- Recall, first  $n+1$  terms of the Taylor Series is also an  $n^{\text{th}}$  degree polynomial.

# Efficient polynomial evaluation

- Given a polynomial in power form how many operations does it take to evaluate it?
- $a_p x^p + \dots + a_1 x + a_0$

## Horner's Rule

- Horner's rule (Horner, 1819) *recursively* evaluates the polynomial  $a_p x^p + \dots + a_1 x + a_0$  as:  
$$((\dots(a_p x + a_{p-1})x + \dots)x + a_0.$$
- costs  $p$  multiplications and  $p$  additions, no extra storage.  
Reduces complexity from  $O(p^2)$  to  $O(p)$

# Interpolation: the story so far

- Given a function at  $N$  points, find its value at other point(s)
- So far: polynomial interpolation
  - Polynomials are guaranteed to approximate any given function in an interval as accurately as we want
- Different polynomial bases
  - Monomial or Power basis
  - Newton and Lagrange basis
- For a given set of points and function values
  - interpolating polynomial is unique
- Interpolation problem requires solution of a linear system
  - System is dense for Monomial/Power basis
  - Newton and Lagrange forms allow the direct solution of the polynomial interpolation form
  - Newton form particularly convenient to add new values
- Error for interpolation with  $n$  points is related to the value of the  $(n+1)^{\text{th}}$  derivative of the underlying function

# Polyinterp

- Lagrange interpolation code
  - x,y are points and function values
  - u are points where vector function  $v = \text{polyinterp}(x,y,u)$

```

n = length(x);
v = zeros(size(u));
for k = 1:n
  %Lagrange function k at u
  w = ones(size(u));
  for j = [1:k-1 k+1:n]
    w = (u-x(j))./(x(k)-x(j)).*w;
  end
  v = v + w*y(k);
end

```

- Cost: 2 nested loops, so the cost is  $n^2$ .

- $k = 5, n = 9$
- $j = [1:k-1 k+1:n]$
- $j = \begin{matrix} 1 & 2 & 3 & 4 & 6 & 7 \\ 8 & 9 & & & & \end{matrix}$

$$p_n(x) = \sum_{i=1}^n L_i(x) f(x_i)$$

$$L_i(x) = \prod_{k=1, k \neq i}^n \frac{(x - x_k)}{(x_i - x_k)}$$

$$L_i(x_j) = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

# Examples of polynomial interpolation

- Go to MATLAB demo
  - Vandermonde
  - Polynomial interpolation for small set
  - For larger set
- See that even for six points we have a problem
  - In between the data points, (especially in first and last subintervals), function shows excessive variation.
  - overshoots changes in the data values.
  - As a result, full-degree polynomial interpolation is hardly ever used for data and curve fitting.
- However we saw polynomial interpolation works well when degree is low

# Piecewise linear interpolation

- Simple idea
  - Connect straight lines between data points
  - Any intermediate value read off from straight line
- The *local variable*,  $s$ , is
- $s = x - x_k$
- The *first divided difference* is
- $\delta_k = (y_{k+1} - y_k)/(x_{k+1} - x_k)$
- With these quantities in hand, the interpolant is
- $L(x) = y_k + (x - x_k) (y_{k+1} - y_k)/(x_{k+1} - x_k)$
- $= y_k + s\delta_k$
- Linear function that passes through  $(x_k, y_k)$  and  $(x_{k+1}, y_{k+1})$

# Piecewise linear interpolation

- Same format as all other interpolants
- Function `diff` finds difference of elements in a vector
- Find appropriate sub-interval
- Evaluate
- Jargon:  $x$  is called a “knot” for the linear spline interpolant

```
function v = piecelin(x,y,u)
%PIECELIN Piecewise linear interpolation.
% v = piecelin(x,y,u) finds piecewise linear L(x)
% with L(x(j)) = y(j) and returns v(k) = L(u(k)).
% First divided difference
delta = diff(y)./diff(x);
% Find subinterval indices k so that x(k) <= u <
x(k+1)
n = length(x);
k = ones(size(u));
for j = 2:n-1
k(x(j) <= u) = j;
end
% Evaluate interpolant
s = u - x(k);
v = y(k) + s.*delta(k);
```

## How good is piecewise linear interpolation?

Recall from Polynomial interpolation: If  $f \in \mathcal{C}^n[I]$ , then

$$f(x) - p_{n-1}(x) = \frac{(x - x_1) \dots (x - x_n) f^{(n)}(\xi)}{n!}$$

for some point  $\xi$  in the interval containing  $I$  and  $x$ .

We need to apply this to a polynomial of degree  $n - 1 = 1$ , so we obtain

$$f(x) - p_1(x) = \frac{(x - x_i)(x - x_{i+1}) f''(\xi)}{2}$$

- So we can reduce error by choosing small intervals where 2<sup>nd</sup> derivative is higher
  - If we can choose where to sample data
  - Do more where the “action” is more