Chapter 4: Interpolation & Extrapolation

§ 1. Introduction

(a) Basic Strategy. Given data table \((x_i, y_i)\), find \(y(\bar{x})\) for an \(\bar{x}\) which is not one of the data points.

* Naive approach: come up with a function \(y(x)\) to go through all (or some) points; then calculate \(y(\bar{x})\).

* This is inefficient and subject to roundoff error; inferior to the practice of constructing \(y(\bar{x})\) directly from \(y_i\).

A philosophical question: Why is extrapolation more "risky" than interpolation? Fundamentally no difference. For latter \(\bar{x}\) bounded in \([x_j, x_{j+1}]\).

(b) Order of approximation & error estimate. How many points to use? Will additional points always increase accuracy? Two extremes: use 2 points or all points.

Order of interpolation = \(#\) of points used - 1.

(c) Local vs. nonlocal fitting interpolation

Local interp.: a finite # of nearest neighbors. This implies discontinuity in derivative as \(\bar{x}\) crosses \(x_i\). Very easy to see via 2-point linear interpolation, but generally true. "Nearest neighbors" change 1 member as \(\bar{x}\) crosses \(x_i\).

* Interpolation by "spline function": to ensure continuity to a certain order of derivative, "softer" interpolation.
\section{Local interpolation}

(a) Search table for the nearest neighbours, e.g. by bisection. [Can you write a program for this?]

\[ x \]
\[ x_1 \quad x_2 \quad \cdots \quad x_j \quad \cdots \quad x_{j+1} \quad \cdots \quad x_n \]

(b) Linear interpolation:

\[ y = y_j + \frac{y_{j+1} - y_j}{x_{j+1} - x_j} (x - x_j) \]

(c) Polynomial interpolation through \( m \) points surrounding \( x \in (x_j, x_{j+1}) \):

- Find the indices of the points, \( k, k+1, \ldots, k+m \).

- \[ K = \min \left( \max \left( j - \frac{m-1}{2}, 1 \right), n+1-m \right) \]

- Takes care of \( m \) being odd or even, and both ends.

\textbf{* Lagrange formula:}

\[ P(x) = \frac{(x-x_1)(x-x_2)\cdots(x-x_m)}{(x_{k+1}-x_1)(x_{k+1}-x_2)\cdots(x_{k+1}-x_m)} y_{k+1} + \frac{(x-x_1)(x-x_2)\cdots(x-x_m)}{(x_{k+2}-x_1)(x_{k+2}-x_2)\cdots(x_{k+2}-x_m)} y_{k+2} + \cdots + \frac{(x-x_1)\cdots(x-x_m-1)}{(x_{k+m}-x_1)\cdots(x_{k+m}-x_m-1)} y_{k+m} \]

\[ = \sum_{j=1}^{m} \frac{\prod_{i=1, i \neq j}^{m} (x - x_i)}{\prod_{i=1, i \neq j}^{m} (x_j - x_i)} y_j \] \((m-1)\text{th order polynomial}\)

- Straightforward, but not numerically efficient. One has to multiply the same factors many times.

(d) Neville's algorithm: a better implementation of the Lagrange formula. [Ref. N. R. 83.1, p 102.]

- Saves calculation; gives error estimation.
(Explain pyramid along with formula)

- Pyramid:

<table>
<thead>
<tr>
<th>Order</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>\ldots</th>
<th>m-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 )</td>
<td>( P_i \rightarrow C_{1i} \rightarrow P_{2i} \rightarrow C_{2i} \rightarrow \ldots \rightarrow P_{mi} \rightarrow \ldots \rightarrow P_{m-1} \rightarrow x_m )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P is polynomial that goes through all indexed points.

- How to construct P's algebraically:

\[
P_{i+1}(x) = \frac{(x-x_i)P_i + (x_{i+1}-x_i)P_{i+1}}{x_{i+1}-x_i}
\]

\[
P_{i+2}(x) = \frac{(x-x_i)P_{i+1} + (x_{i+2}-x_i)P_{i+2}}{x_{i+2}-x_i}
\]

\[
P_{i+3}(x) = \frac{(x-x_i)P_{i+2} + (x_{i+3}-x_i)P_{i+3}}{x_{i+3}-x_i}
\]

\[
P_{i+4}(x) = \frac{(x-x_i)P_{i+3} + (x_{i+4}-x_i)P_{i+4}}{x_{i+4}-x_i}
\]

\[
\vdots
\]

\[
P_{i+(m-1)}(x) = \frac{(x-x_i)P_{i+(m-2)} + (x_{i+(m-1)}-x_i)P_{i+(m-1)}}{x_{i+(m-1)}-x_i}
\]

Note: (i) Two parents already agree on all internal points. Each "linear combination" adds the end points by raising polynomial power by 1. (ii) Given an \( x \) value, these are NOT polyn. but \#'s!
- Error of interpolation & reformulation

Going back to pyramid: each interpolation can be seen as correction of either "parent."

\[ C_{l,i} = \frac{P_i \cdots (i+l)_{	ext{new}} - P_i \cdots (i+l-1)}{(i+l)} \text{ upper} \]

\[ D_{l,i} = \frac{P_i \cdots (i+l)}{(i+l)} \frac{P(i+h) \cdots (i+l)}{l} \text{ lower} \]

(l: level of interpolation; i: location in table)

The C's and D's can be computed easily via a recursive formula:

\[ C_{l+1,i} = \frac{(x_i-x)(C_{l,i} - D_{l,i})}{(x_i-x_{i+l+1})} \]

\[ D_{l+1,i} = \frac{(x_i-x_{i+l+1}) \cdot x_i - x}{(x_i-x_{i+l+1})} \]

Proof: \[ C_{l+1,i} = \frac{P_i \cdots (i+l+1) - P_i \cdots (i+l)}{i} \]

\[ = \frac{(x - x_{i+l+1})P \cdots (i+l) + (x_i - x)P_{i+1} \cdots (i+l+1)}{x_i - x_{i+l+1}} - P_i \cdots (i+l) \]

\[ = \frac{(x_i - x)P \cdots (i+l) + (x_i - x)P_{i+1} \cdots (i+l+1)}{x_i - x_{i+l+1}} \]

\[ = \frac{x_i - x}{x_i - x_{i+l+1}} \left[ P_i \cdots (i+l+1) - P_i \cdots (i+l) \right] \]

\[ + \frac{x_i - x}{x_i - x_{i+l+1}} \frac{C_{l,i} - D_{l,i}}{2} \]

\[ = \frac{x_i - x}{x_i - x_{i+l+1}} \left[ C_{l,i+1} - D_{l,i} \right] \]
Code implementation:

- Cumulate the errors to arrive at final value.

* Why are C's & D's errors? Each is an interpolation for f(x) based on different set of points. C, D are differences between one order of interpolated f(x) next. Last ones used as criteria.

a) Where to start? Start from data point closest to target x. That way, we use the closest data points for interpolation. The errors are smaller, and we can hope to terminate within fewer layers.

b) How to initialize C's & D's? It turns out that for the 1st column, we use the successive formulas for C1, C2, etc as if C0 = D0 = P1, ..., Cn = Dn = Pn. This is as if the ancestor on the left of Pi, ..., Pn are C's.

Test: \( C_i = \frac{(x_1 - x)(C_{i-1} - D_{i-1})}{x_i - x_1} = \frac{x_i - x}{x_1 - x_2} \cdot \frac{(x_1 - x)(C_{i-1} - D_{i-1})}{x_i - x_2} \cdot \frac{(x_1 - x)(C_{i-1} - D_{i-1})}{x_i - x_2} - P_i = P_{i+1} - P_i. \)

c) Choose path that is straight from x to the top, a path centered around the starting point.

(d) Amount of calculation:

- Each P has 2 multiplication
  and 1 division \( \Rightarrow \) 3 calculations
- Each level has \((m-1) P's:\n
\( (m-1) + (m-2) + \cdots + 2 + 1 = \frac{m(m-1)}{2} \)

- Total: \( \frac{3}{2} \cdot m(m-1) \)
- Lagrange: \( 2(m-1) + 1 + 1 = 2m \)

for each point, total \( 2m^2 \) calculations.

Neville: saves more than 25%, & gives error estimation.
SUBROUTINE polint(xa, ya, n, x, y, dy)
INTEGER n, NMAX
REAL dy, x, y, xa(n), ya(n)
PARAMETER (NMAX=10)
INTEGER i, m, ns
REAL den, dif, dift, ho, hp, w, c(NMAX), d(NMAX)
ns = 1
dif = abs(x - xa(1))
do 11 i = 1, n
   dift = abs(x - xa(i))
   if (dift .lt. dif) then
      ns = i
      dif = dift
   endif
   c(i) = ya(i)
   d(i) = ya(i)
continue
y = ya(ns)
ns = ns - 1
do 13 m = 1, n - 1
   do 12 i = 1, n - m
      ho = xa(i) - x
      hp = xa(i + m) - x
      w = c(i + 1) - d(i)
      den = ho - hp
      if (den .eq. 0.) pause 'failure in polint'
      den = w / den
      d(i) = hp * den
      c(i) = ho * den
      continue
   if (2 * ns .lt. n - m) then
      dy = c(ns + 1)
   else
      dy = d(ns)
      ns = ns - 1
   endif
   y = y + dy
continue
return
END