Adding Data Points to an Existing Set

- Suppose that we determined an interpolating polynomial for a set of data and decide that we want to add a couple of more points in a specific region to improve our approximation.

- Recall that the Lagrange form of the interpolating polynomial is

\[ p_n = \sum_{i=1}^{n+1} y_i L_i(x), \]

\[ L_i(x) = \frac{(x - x_1)(x - x_2) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_{n+1})}{(x_i - x_1)(x_i - x_2) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_{n+1})} \]

- If we add additional points to our data set then we have to start all over with the Lagrange form of our polynomial.

- Recall that the interpolating polynomial is unique so if we find another way to determine this polynomial which allows us to add points without starting over, then it could be useful.
Newton Form of the Interpolating Polynomial

• This approach to determining the interpolating polynomial using divided differences and allows adding data without starting over.

• When we first looked at writing an interpolating polynomial we wrote it as \( c_1 + c_2 x + c_3 x^2 + \cdots + c_{n+1} x^n \) in terms of the monomials \( 1, x, x^2, \ldots, x^n \). This required the solution of \( n + 1 \) linear equations.

• Recall that when we wrote the Lagrange form of say the third degree interpolating polynomial we wrote it in terms of sums of cubic polynomials. The Newton form takes a different approach.

• The Newton form of the line passing through \((x_1, y_1), (x_2, y_2)\) is

\[
p_1(x) = a_1 + a_2(x - x_1)
\]

• The Newton form of the parabola passing through \((x_1, y_1), (x_2, y_2), (x_3, y_3)\) is

\[
p_2(x) = a_1 + a_2(x - x_1) + a_3(x - x_1)(x - x_2)
\]
The general form of the Newton polynomial passing through \((x_i, y_i), \ i = 1, \ldots, n + 1\) is

\[
p_n(x) = a_1 + a_2(x-x_1)+a_3(x-x_1)(x-x_2)+\cdots+a_{n+1}(x-x_1)(x-x_2)\cdots(x-x_n)
\]

As in the case of the Lagrange form of the interpolating polynomial we first find the coefficients \(a_i, \ i = 1, \ldots, n = 1\) and then we use this formula to evaluation \(p_n\) at a point.
Hermite Interpolation

- Last time we looked at interpolating data points or equivalently function values.
- Sometimes we have the function value and its derivatives which we would like to interpolate.
- For example, suppose we have the data for the position and velocity of a vehicle at various times.
- When we interpolate derivative values this is called Hermite interpolation as opposed to just function values which is Lagrange interpolation.
- We can obtain the Lagrange form for a Hermite polynomial.
- If we use the divided difference table the values $x_i$ are repeated $m$ times if we want to interpolate $m - 1$ derivatives; i.e., for the function value and each of its $m - 1$ derivatives. The derivative values are used for the first divided difference between two repeated points.
• We can write codes to evaluate an interpolating polynomial using the Newton form of the interpolating polynomial in a straightforward manner.

• We can also incorporate routines to handle interpolating derivative values.

• However, we don’t want to spend too much time on this now because the next example shows us that as the number of interpolating points increase (and thus the degree of the polynomial increases) we don’t always get the results we want.
Runge’s Example

Consider the function

\[ f(x) = \frac{1}{1 + 25x^2} \quad -1 \leq x \leq 1 \]

which we want to interpolate (using only function values) with an increasing number of points. We interpolate with evenly spaced points. The black curve represents the function.
• As you can see, the interpolant gets more “wiggles” in it as it is required to interpolate more points.

• It is for this reason that one should not use a single polynomial to interpolate a lot of data.

• The general rule is that

  HIGH DEGREE POLYNOMIAL INTERPOLATION SHOULD BE AVOIDED

• What can we do instead?

• We use something called piecewise interpolation.

• When a graphing program plots a function like we did in the previous examples it connects closely spaced points with a straight line. If the points are close enough, then the result looks like a curve to our eye. This is called piecewise linear interpolation.
Examples of Piecewise Functions

Piecewise Constant Function
We will be interpolating data using a piecewise linear function on each subinterval so the linear function defined on the interval \([x_{i-1}, x_i]\) and the function defined on \([x_i, x_{i+1}]\) will match up at \(x_i\); i.e., we are using continuous, piecewise linear functions.
Below we see some examples of a piecewise linear interpolant to $\sin x^2$ on (-2,2). In each plot we are using uniform subintervals and doubling the number of subintervals from one plot to the next.
Below we see some examples of piecewise linear interpolation for Runge’s example. Note that this form of interpolation does not have the “wiggles” we encountered by using a higher degree polynomial.
• Of course we can also do **piecewise quadratic or cubic interpolation**. Usually we don’t go higher than this.

• Alternately, we can perform **piecewise Hermite interpolation** such as piecewise cubic Hermite where we interpolate the function value and first derivatives at each end of the interval (i.e., we have four conditions on each subinterval).

• Remember that piecewise interpolation is almost always preferable to using a higher order polynomial.
Piecewise Linear Interpolation

• Assume that we are given \((x_1, y_1), (x_2, y_2), \ldots, (x_{n+1}, y_{n+1})\). We want to construct the piecewise linear interpolant.
• We divide our domain into points $x_i \ i = 1, \ldots, n + 1$
• The linear interpolation on the interval $[x_i, x_{i+1}]$ is just
  \[ L_i(x) = a_i + b_i(x - x_i) \]
  where
  \[ a_i = y_i \quad y_{i+1} = y_i + b_i(x_{i+1} - x_i) \Rightarrow b_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \]
• Thus our piecewise linear interpolant is given by
  \[
  L(x) = \begin{cases} 
  L_1(x) & \text{if } x_1 \leq x \leq x_2 \\
  L_2(x) & \text{if } x_2 \leq x \leq x_3 \\
  \vdots & \vdots \\
  L_i(x) & \text{if } x_i \leq x \leq x_{i+1} \\
  \vdots & \vdots \\
  L_n(x) & \text{if } x_n \leq x \leq x_{n+1}
  \end{cases}
  \]
• Note that \( \mathcal{L}(x) \) is continuous but not differentiable everywhere.

Example

Determine the piecewise linear interpolant to \( \sin x \) on \([0, \pi]\) found by using 2 equal subintervals; then evaluate it at the points \( \frac{\pi}{3}, \frac{3\pi}{5} \). Compute error.

• Our intervals are \([0, \frac{\pi}{2}]\) and \([\frac{\pi}{2}, \pi]\).

• On the first interval the coefficients are

\[
a_1 = \sin 0 = 0 \quad b_1 = \frac{\sin \frac{\pi}{2} - \sin 0}{\frac{\pi}{2} - 0} = \frac{2}{\pi}
\]

so we have \( \mathcal{L}_1 = 0 + \frac{2}{\pi}(x - 0) = \frac{2x}{\pi} \).

• On the second interval the coefficients are

\[
a_2 = \sin \frac{\pi}{2} = 1 \quad b_2 = \frac{\sin \pi - \sin \frac{\pi}{2}}{\pi - \frac{\pi}{2}} = \frac{0 - 1}{\frac{\pi}{2}} = -\frac{2}{\pi}
\]
which gives $L_2 = 1 - \frac{2}{\pi}(x - \frac{\pi}{2})$.

• The point $\frac{\pi}{3}$ is in the first interval so

$$L_1(x) = a_1 + b_1(x - x_1) \Rightarrow L_1(\frac{\pi}{3}) = 0 + \frac{2}{\pi}(\frac{\pi}{3} - 0) = \frac{2}{3}$$

The actual value of $\sin \frac{\pi}{3}$ is 0.866025 so our error is approximately 0.2. The error is fairly large because the length of our subinterval is large $\frac{\pi}{2} \approx 1.57$.

• Now the point $\frac{3\pi}{5}$ is in the second interval so

$$L_2(x) = a_2 + b_2(x - x_2) \Rightarrow L_2(\frac{3\pi}{5}) = 1 - \frac{2}{\pi}(\frac{3\pi}{5} - \frac{\pi}{2}) = \frac{4}{5}$$

The actual value of $\sin \frac{3\pi}{5}$ is 0.950157 so our error is approximately 0.151.
Implementing Piecewise Linear Interpolation

- Given the data \((x_i, y_i), i = 1, 2, \ldots, n + 1\), we can determine the coefficients \(a_i, b_i, i = 1, \ldots, n\) on each of the \(n\) subintervals by the formulas

\[
a_i = y_i \quad b_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}, \quad i = 1, 2, \ldots, n
\]

This can be done once and stored just like we did for the \(n\)th degree interpolating polynomial.

- Unlike the case when we had a single \(n\)th degree polynomial, when we want to evaluate our piecewise linear interpolant at some point \(x\) then we have to decide which formula \(L_i(x)\) we need to use. That is, we have to decide the subinterval \([x_i, x_{i+1}]\) so that \(x \in [x_i, x_{i+1}]\). Once we do this, then we simply use the formula \(a_i + b_i(x - x_i)\) where our coefficients \(a_i, b_i\) are known.

- Before we look at the actual implementation, lets look at piecewise quadratic interpolation because it brings up some additional issues.
We know that to determine the **quadratic interpolation** on an interval we need 3 conditions so we have to add another point.

This is possible to do if we are approximating a given function by this piecewise quadratic function since we could just choose the midpoint.

However, if we have discrete data, we might not be able to add another point. In that case we must have an **even number** of intervals and then we choose the intervals \([x_i, x_{i+2}]\) to perform the quadratic interpolation.

For simplicity we will take the approach that we are going to approximate a given function so that we can generate another point in the interval and we choose the midpoint.
• We assume that we have \( n \) intervals and \( 2n + 1 \) points labeled \( x_i, \ i = 1, \ldots, 2n + 1 \). For example, the intervals are
\[
[x_1, x_3], \ [x_3, x_5], \ \cdots \ [x_{2n-1}, x_{2n+1}]
\]

• Then using the Newton form of the interpolating polynomial we have for the interval \([x_1, x_3]\) with \( x_2 \) the midpoint
\[
Q_1(x) = a_1 + b_1(x - x_1) + c_1(x - x_1)(x - x_2)
\]
so when we evaluate \( Q_1(x_1), Q_1(x_2), \) and \( Q_1(x_3) \) we get
\[
Q_1(x_1) = y_1 \Rightarrow a_1 = y_1
\]
\[
Q_1(x_2) = y_2 \Rightarrow y_2 = y_1 + b_1(x_2 - x_1) \Rightarrow b_1 = \frac{y_2 - y_1}{x_2 - x_1} = 2 \frac{y_2 - y_1}{\Delta x_1}
\]
\[
Q_1(x_3) = y_3 \Rightarrow y_3 = y_1 + b_1 \Delta x_1 + c_1 \Delta x_1 \left( \frac{\Delta x_1}{2} \right) \Rightarrow c_1 = 2 \frac{y_3 - y_1 - b_1 \Delta x_1}{\Delta x_1^2}
\]
where $\Delta x_1 = x_3 - x_1$.

- Thus our piecewise quadratic interpolant on the entire region $[x_1, x_{2n+1}]$ is given by a quadratic on each of the $n$ intervals:

$$Q(x) = \begin{cases} 
Q_1(x) & \text{if } x_1 \leq x \leq x_3 \\
Q_2(x) & \text{if } x_3 \leq x \leq x_5 \\
\vdots & \vdots \\
Q_i(x) & \text{if } x_i \leq x \leq x_{i+2} \\
\vdots & \vdots \\
Q_n(x) & \text{if } x_{2n-1} \leq x \leq x_{2n+1}
\end{cases}$$

- Note that as in the case of the piecewise linear interpolant, $Q(x)$ is continuous everywhere but not differentiable everywhere.

- So one thing that is different from the linear case is the relationship between the number of intervals and the number of points.
Implementation Strategy

- Our ultimate goal is to write modules for piecewise linear, piecewise quadratic, piecewise cubic, and piecewise cubic Hermite interpolation.

- We want to have a test program where we define a member of the appropriate class, say `poly` and then, e.g., to compute the coefficients, we just say `call coefficients(poly)` and it will call the appropriate routine depending on what class `poly` belongs to. We would like to call all of our routines in our module by a generic name. That way we can easily perform any of the types of piecewise polynomial interpolation that we code.

- We will set up our test program to handle the case where we are approximating a given function. It could easily be changed to the case where we are given explicit data.
Structure of calling program

- **Given:** type of interpolating polynomial, interval information over which we are interpolating, number of subintervals, number of evenly spaced points to evaluate polynomial at. We will also need a flag which indicates which method to use.

- Call routine to set number of intervals, compute the number of points, and allocate arrays

- Call routine to set evenly spaced x-coordinates over a given interval

- Evaluate y-points (& derivatives if using Hermite interpolation) from function

- Call routine to compute coefficients

- Evaluate piecewise interpolating polynomial at a set of evenly spaced points
  - Call routine to find interval point is in
  - Call routine to evaluate polynomial
  - Write points to file for plotting and/or accumulate an error vector

- Determine an error between the interpolating polynomial and the function
• functions for $f(x)$ and $f'(x)$ (if Hermite cubic used)
In general, our classes can have the following components; of course for Hermite interpolation we will have to include the derivative values as well as the $y$ values. This definition is analogous to our class for the Lagrange form of the interpolating polynomial.

```
type pw_linear_inter

    integer :: n_points, n_intervals

    real(prec), pointer, dimension(:) :: x, y

    real(prec), pointer, dimension(:, :) :: coefficients

end type pw_linear_inter
```
Setting the number of points and Allocating arrays

- We agreed that we will input the number of intervals.
- We will have routines called `make_linear`, `make_quadratic`, etc. which will set the number of points given the number of intervals and allocate the appropriate arrays.
- Keep in mind that the coefficients are defined over an interval.
- Number of points:
  - Linear: number of points = number of intervals + 1
  - Quadratic: number of points = 2*number of intervals + 1
  - Cubic: number of points = ?
- Dimension of arrays
  - `poly % x` and `poly % y` are one-d arrays dimensioned by the number of points
The coefficients are stored in a two-d array

- *Linear*: `allocate ( poly % coefficients(2, n_intervals) )`
- *Quadratic*: `allocate( poly% coefficients(3, n_intervals))`
- *Cubic*: `allocate ( poly % coefficients(? , n_intervals ) )`

```fortran
subroutine make_linear ( poly )

integer, intent(in) :: n

type ( pw_linear_inter ), intent(inout) :: poly

integer :: n

n = poly % n_intervals

poly % n_points = n + 1

allocate ( poly % x(n+1), poly % y(n+1) )

allocate ( poly % coefficients(2,n ) )
```
end subroutine make_linear

We will add the interface statement

interface make_pw_poly
    module procedure make_linear
end interface make_pw_poly

end interface make_pw_poly

Next time we will look at piecewise quadratic interpolants and cubic Hermite interpolants. We will also write our routines for generating the interpolation points and determining the coefficients.
When we obtain interpolants of a complicated function or a set of data we want to be able to plot the interpolant along with the function or data.

Gnuplot is a free ware graphing utility package that works on many platforms and is easy to use.

If you have another software package that you typically use to make plots then you can use that for homework; otherwise Gnuplot is a simple alternative.