Summary & Goals

• The problem we are addressing is to find a polynomial which interpolates (i.e., agrees with or passes through) a given set of points \((x_i, y_i)\).

• An equivalent problem is approximating a complicated function by determining an interpolating polynomial.

• We said this polynomial is guaranteed to be unique if the points are distinct.

• One way to find this polynomial is to write it as \(p_n(x) = a_0 + a_1x + \cdots + a_nx^n\) and then determine the \(n + 1\) linear equations so that \(p_n(x_i) = y_i, i = 1, 2, \ldots, n + 1\).

• This approach is computationally intensive because we have to solve \(n + 1\) linear equations.

• We first looked at the Lagrange form of the interpolating polynomial which writes the \(n\)th degree polynomial as a combination of \(n\)th degree polynomials \(L_i\) which have the property that \(L_i(x_j) = 0\) if \(i \neq j\) and \(= 1\) if \(i = j\). We
wrote routines to determine the coefficients and evaluate the polynomial at a point.

- We said that there was another form of the interpolating polynomial - the Newton form - which has the advantage that we can easily add data. It writes $p_n(x) = a_0 + a_1(x - x_1) + a_2(x - x_1)(x - x_2) + \cdots$.

- Then we asked ourselves if interpolating a large data set by a single polynomial is the best approach. We saw from Runge’s example that this type of interpolation can often lead to large “wiggles” and it NOT recommended.

- The alternative is to use piecewise polynomial interpolation. Here we break our domain into subintervals and over each interval we use, e.g., a linear polynomial to interpolate the function values. If we use a linear or higher degree polynomial on each subinterval, then the piecewise polynomial is continuous.

- We said that we could also interpolate derivatives values as well as function values.

- We looked at the freeware GNUPLOT for plotting data and functions.
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.
• 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
Below we see some examples of piecewise linear interpolation for Runge’s example.
• 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.
• 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

\[
a_i = y_i \quad y_{i+1} = y_i + b_i(x_{i+1} - x_i) \quad \Rightarrow \quad b_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}
\]

• 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

Evaluate the piecewise linear interpolant to $\sin x$ on $[0, \pi]$ using 2 equal subintervals 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

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

• On the second interval the coefficients are

$$a_2 = \sin \frac{\pi}{2} = 1 \quad b_2 = \frac{\sin \pi - \sin \frac{\pi}{2}}{\pi - \frac{\pi}{2}} = \frac{0 - 1}{\frac{\pi}{2}} = -\frac{2}{\pi}$$
• The point $\frac{\pi}{3}$ is in the first interval so

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

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

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

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

$$[x_1, x_3], \quad [x_3, x_5], \quad \cdots \quad [x_{2n-1}, x_{2n+1}]$$

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:

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

- 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.
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 to, e.g., 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 using uniformly spaced points over an interval. It could easily be changed to the case where we are given explicit data.
Structure of calling program

- **Given:** type of piecewise interpolating polynomial (e.g., linear, quadratic, cubic, cubic Hermite), the endpoints of the entire interval we want to interpolate (equivalently the left endpoint and the length), and the number of equal subintervals.
- call routine to set number of intervals, number of points, and allocate arrays
- call routine to set evenly spaced x-coordinates over a given interval (e.g., for linears we only need endpoints, for quadratics we need endpoints and midpoints, etc.)
- evaluate y-points (& derivatives if using Hermite interpolation) from function
- call routine to compute coefficients
- to evaluate piecewise interpolating polynomial at points we need to
  - 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
• We also need functions for $f(x)$ and $f'(x)$ (if Hermite cubic used)
Defining our Derived Data Type

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.

```fortran
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
```
Recall that on each subinterval we have 2 coefficients \((a_i, b_i)\) for linears so we have an array where the row number corresponds to the subinterval and the first column to \(a_i\) and the second to \(b_i\).

For quadratics we have three coefficients per interval. For cubics we have four.

**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; of course we could have also chosen them to be stored in a 2-d array whose length is the number of points by two columns.

- The coefficients are stored in a two-d array
  - *Linear*: `allocate ( poly % coefficients( n_intervals, 2 ) )`
  - *Quadratic*: `allocate( poly% coefficients( n_intervals, 3))`
  - *Cubic*: `allocate ( poly % coefficients( n_intervals,4 ) )`
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( n,2 ) )

end subroutine make_linear
We will add the interface statement

```fortran
interface make_pw_poly

    module procedure make_linear

end interface make_pw_poly
```

An analogous routine can be made for piecewise quadratic and piecewise cubic polynomials in their respective modules and an equivalent interface statement using the same generic call `make_pw_poly`. For example,

```fortran
subroutine make_quadratic ( poly )

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

    integer :: n

    n = poly % n_intervals

    poly % n_points = 2*n + 1
```
allocate ( poly % x(2*n+1), poly % y(2*n+1))
allocate ( poly % coefficients( n,3 ) )
end subroutine make_quadratic

interface make_pw_poly
   module procedure make_quadratic
end interface make_pw_poly
Setting the evenly spaced \( x \)-coordinates of points to interpolate

- We said that we would write our code assuming that we want to interpolate a given function (and perhaps its derivatives) on a given interval.
- If we had discrete data instead, we could simply change this to a routine to read in the data.
- So the next step would be to set the \( x_i \) values given the number of intervals, the left endpoint of the interval and its length. We will use evenly spaced points for simplicity. We will use a separate routine to do this because we know that e.g., in quadratics we must also set up the midpoint of the interval.
- Our routine for linears could have the name
  
  \[
  \text{subroutine set}_x_i \_\text{linear}(\text{poly}, x_\text{left}, \text{length})
  \]

- The executable statements for linears are basically what we have done before.
n = poly % n_intervals
deltax = length / real (n )
x = x_left -deltax

do i = 1, poly % n_points
    x = x + deltax
    poly % x (i) = x
end do

• To do quadratics, all we need to change above is instead of incrementing with \( \Delta x \) we increment with \( \frac{\Delta x}{2} \). Recall that the poly % n_points already is set to the total number of points.
Implementing the Coefficients

• Given the \((x_i, y_i), i = 1, 2, \ldots, \) number of points, we can determine the coefficients of the polynomial on each subinterval from the given analytic formulas. This must only be done once.

**Linear polynomials:**

• 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

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

• Recall that we stored our coefficients in an array dimensioned the number of intervals by 2.
subroutine compute_linear_coefficients ( poly )
type ( pw_linear_inter ), intent(inout) :: poly

integer :: n, i
real(prec) :: xi, xip1, yi, yip1

n = poly % n_intervals

do i = 1, n
    xi = poly % x(i) ; xip1 = poly % x(i+1)
    yi = poly % y(i); yip1 = poly % y(i+1)
    poly % coefficients ( i, 1 ) = yi
    poly % coefficients ( i, 2 ) = ( yip1 - yi ) / ( xip1 - xi)
end do

dend subroutine compute_linear_coefficients
Quadratics:

Recall that the formulas for the coefficients for the polynomial

\[ a_i + b_i(x - x_i) + c_i(x - x_i)(x - x_{i+1}) \]

on the \( i \)th interval with points \([x_k, x_{k+2}]\) with midpoint \( x_{k+1} \) are

\[ a_i = y_k \]
\[ b_i = 2\frac{y_{k+1} - y_k}{\Delta x_i} \]
\[ c_i = 2\frac{y_{k+2} - y_k - b_i \Delta x_i}{\Delta x_i^2} \]

where \( \Delta x_i = x_{i+2} - x_i \). To derive these formulas we write on the \( i \)th interval

\[ p_2(x) = a_i + b_i(x - x_k) + c_i(x - x_k)(x - x_{k+1}) \]

and set \( p_2(x_j) = y_j \) for \( j = k, k + 1, k + 2 \).

Of course we need the relationship between the interval \( i \) and its left endpoint \( k \); for linears \( k = i \) but for quadratics \( k = 2i - 1 \)
The main loop in our subroutine `compute_quadratic_coefficients` should look like

```
do i = 1, n  ! loop over number of intervals
    k = 2 * i - 1  ! the index of left endpoint
    xk = poly % x(k); xkp1 = poly % x(k+1); xkp2 = poly % x(k+2);
    yk = poly % y(k); ykp1 = poly % y(k+1); ykp2 = poly % y(k+2);
    deltax = xkp2 - xk
    poly % coefficients ( i, 1 ) = yk
    bi = two * ( ykp1 - yk ) / deltax
    poly % coefficients ( i, 2 ) = bi
    poly % coefficients ( i, 3 ) = &
        two * ( ykp2 - yk - bi*deltax ) / (deltax**2)
```
Next time (after the test) we will see a way to determine the subinterval a point $x$ is in.

Midterm next class!