Evaluating the polynomial at a point

- Recall that we have a data structure for each piecewise polynomial (linear, quadratic, cubic and cubic Hermite).
- We have a routine that sets evenly spaced interpolation nodes given the (i) number of intervals, (ii) degree of polynomial on each interval, (iii) the left endpoint of interval we are interpolating function and (iv) either the right endpoint or length of interval. We will use function overloading to make a generic call `set_xi`.
- We have a routines to compute the coefficients which will be called with a generic call `compute_coefficients`.
- Recall that once we have our coefficients we want to see how good an approximation our piecewise interpolant is to a given function. So we evaluate the function and the interpolant at many points (probably evenly spaced) and create an error vector.
- When we want to evaluate our piecewise interpolant at some point $x$ then
we have to decide which coefficients to use in our formula. That is, we have to decide the subinterval.

- For linears this means we want to find $i$ such 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.

- For linears we might have a function `evaluate_linear_polynomial (poly, x, loc) result(pn_at_x)` where `poly` is a member of our piecewise linear class, $x$ is the point we are evaluating at, and `loc` is the interval number which $x$ is in. Then our statement would simply be

```plaintext
pn_at_x = poly % coefficients(1,loc) &
+ poly % coefficients(2,loc) * ( x - poly % x(loc) )
```

- Of course, we don’t know apriori what interval $x$ is in so `loc` might just be our guess for the interval and then we need to determine which interval it is in.
- For quadratics this means we want to find $i$ (odd integer) such that $x \in [x_i, x_{i+2}]$. Once we do this, then we simply use the formula

$$a_i + b_i(x - x_i) + c_i(x - x_i)(x - x_{i+1})$$

where our coefficients $a_i, b_i, c_i$ are known.

- So for quadratics we have the same calling statement (remember our goal is to use a generic call so we need the routines to look the same) and simply have the executable statements

```plaintext
xi = poly % x(loc); xip1 = poly % x(loc+1)
pn_at_x = poly % coefficients(1,loc) &
 + poly % coefficients(2,loc) * ( x -xi ) &
 + poly % coefficients(3,loc) * ( x - xi )*(x-xip1 )
```

- So, unlike the situation where we just have one $n$th degree polynomial, here we have a different polynomial over each subinterval. We have the additional problem of incorporating a search algorithm.

- The “brute force” approach to find the interval would be to loop from $j = 1$ to $j = n$ and check (assuming $x_j < x_{j+1}$ for all $j$) if $x$ is less than right
endpoint (done if true) otherwise increment \( j \). Of course we should probably add a check to make sure \( x \) is in the interval \([x_1, x_{n\text{points}}]\). For linears this is just
do j =1,n_intervals
  if ( x <= poly % x(j+1 ) ) then
    loc = j; exit ! x in [x_j,x_j+1]
  end if
end do

* We could also have checked to see if

\[ x_j \leq x \leq x_{j+1} \]

but this would have been two tests and we only really need one. Why?

* This is fine if we only have a few points but if we have a lot of points, or we
  are in higher dimensions (e.g., \( R^{10} \)), then it is costly.

* A second approach is to exploit the monotonicity of the \( x_i \) and to use a binary search, i.e., essentially the bisection method. We know that \( x_1 \leq x \leq x_n \) and then we choose the middle \( x_i \) (say \( x_{mid} \)) and see if \( x_1 \leq x \leq x_{mid} \) or \( x_{mid} \leq x \leq x_n \). We continue as in the bisection method approximately halving the number of intervals where \( x \) is contained at each step.

* Note that in this case we are halving an integer which is our index on \( x \). This is in contrast to the bisection method where we are halving a real
number, the length of the interval. So for example, if we have \( x \in [x_1, x_{10}] \) then we choose the intervals \([x_1, x_5], [x_5, x_{10}]\) (because \((10 - 1)/2 = 4\) in integer arithmetic) and the first interval contains 5 nodes whereas the second contains 6 so since we are using integers we don’t always find a middle \( x_i \).  

- For a random point \( x \) we really can’t do better than this binary search.  
- However, if we are evaluating our interpolant at a set of points \( z_j, j = 1, m \) where \( z_j < z_{j+1} \) then it is more efficient to exploit the left-to-right ordering of the points.  
- That is, if \( z_j \in [x_i, x_{i+1}] \) then to find the location of \( z_{j+1} \) we first look in \([x_i, x_{i+1}]\). If it is not there then we could do a binary search starting with \([x_{i+1}, x_n]\).  
- An alternate strategy would be to look at the next interval to the right \([x_{i+1}, x_{i+2}]\) and check to see if \( z_j \) is in it before starting our binary search on \([x_{i+2}, x_n]\). Either strategy could be coded.  
- Note that we will need this search algorithm regardless of which type of piecewise interpolation we choose to use.
Search Algorithm

- Our goal is to write a module for piecewise interpolation using piecewise linear, quadratic, cubic and Hermite cubic.
- Since to evaluate each of these piecewise polynomials at a point $x$ we need to determine the interval containing $x$.
- We are going to write one search routine which would work for all of these piecewise polynomials; we could have a separate version for each polynomial. Remember the difficulty is that the interval number and the node number of the left coordinate of the interval are not the same, except for linear.
- Since we want our routine to work for any of our piecewise polynomials we have to be careful about our input.
- In a first attempt, we might just pass in poly, a member of our class. So we might have something like subroutine find_interval (poly, x, loc ) where loc is an initial guess for the interval if we use the strategy described above.
• If we pass in *poly* this causes a small problem because we have to declare it. Why is this a problem?

• How can we modify the code if we want to do this?

• We will take the approach of not passing in a member of our class but simply pass in the components that we need. We have

  ```fortran
  subroutine find_interval (flag_method, n_points, n_intervals, xi, x, loc)
  Here *xi* are the nodes, *loc* is an initial guess for our interval; the others variables should be self-explanatory.
  ```

• On output the subroutine returns the interval *x* is in and overwrites *loc* with this since it is the initial guess for the next call to this routine.

• In the main testing program we can initialize *loc* to one since we start looking in the first interval to the left.

• Let’s look at what we need before we decide on the final structure of the code.
Outline of search strategy

- Check to see if $x$ is in desired region; that is, between the left node of interval $loc$ and $xi(n\_points)$. For linears this is just

```fortran
if ( x > xi(n\_points) .or. x < xi(loc) ) then
    print *, "error: x not in given region", x, xi(loc), xi(n\_points)
    stop
end if
```

Note that for quadratics or cubics we will have to map the interval designated by $loc$ to a node; for example, if you are in the fifth interval then the node number at the left endpoint is nine since $2(5-1) +1=9$ for quadratics.

- To do this, we will provide a function `map_interval_to_node(flag\_method, loc)` which simply determines the left node number of the interval $loc$.

- For linears its the same as the interval number.
- For quadratics it’s $2(\text{interval number} - 1)+1$
• Then to modify the above code for any of our piecewise polynomials we have

\[
\text{node}_\text{loc} = \text{map}_\text{interval}_\text{to}_\text{node} (\text{flag}_\text{method}, \text{loc})
\]

\[
\text{if} \ (x > x_i(\text{n_points}) \ \text{or} \ x < x_i(\text{node}_\text{loc}) \ ) \ \text{then}
\]

• In the code I added a check to make sure \( x \) is not actually equal to \( x_i(\text{n_points}) \).

• We said we would first check to see if \( x \) is in the interval \( \text{loc} \). For linears this conditional is just

\[
\text{if} \ (x \geq x_i(\text{loc}) \ \text{and} \ x \leq x_i(\text{loc} + 1) \ ) \ \text{then}
\]

• Now for the general routine we can change \( \text{loc} \) to \( \text{node}_\text{loc} \) but this is not all we need to change. What else needs modifying for the general case?

• We have
inc = get_increment( flag_method )
if ( x >= xi( node_loc ) .and. x <= xi( loc + inc ) ) then

• Next our strategy is to check to see if \( x \) is in the interval to the right of our initial guess (if it wasn’t in that one). For linears this is just

else if ( x >= xi( loc +1 ) .and. x <= xi( loc +2 ) ) then

\[
\text{loc = loc +1 ! in next interval}
\]
return

• What do we have to change to do the general case?
else if ( \( x \geq x_i(node_loc+inc) \) .and. \( x \leq x_i(node_loc+2*inc) \) ) then

- If \( x \) is not in the initial or the next interval to the right, then we do a binary search.

  call binary_search (flag_method, loc_init, n_intervals, x, xi, loc_final )

Here \( loc_init \) is set to \( loc + 2 \). Why?

- \( loc_final \) returns the interval location where \( x \) is; we then set \( loc = loc_final \) in our routine find_interval and return to calling program.
Binary search

• First we check to see if there is only one interval left; this is just

```cpp
if (loc_init == n_intervals ) then
    loc_final = loc_init
    return ;  end if
```

• For the loop over the section of code where we half the intervals, I used a do loop which is not counter controlled; to leave the loop we simply use `exit`. We simply say `do` and `end do` instead of adding an increment.

• This can easily lead to an infinite loop if something is coded incorrectly. To avoid this I usually put a manual counter. For example, I initialize it to zero before the do loop and manually increment it in the loop. Then I put in a check such as

```cpp
if ( counter > n_intervals ) stop  ! error- infinite loop
```

• We first find the midpoint interval (modulo integer arithmetic) and check to see if by chance $x$ equals the left endpoint of this interval. If so, then we have our interval.

```cpp
if ( loc midpoint == x ) then
    stop ;  ! found interval
end if
```
• If not, then we check to see which part of the interval $x$ is in. As in the bisection method we move either the left or right endpoint node to midpoint node.

• We terminate the loop when we only have one interval left.

• We can take a look at the actual code now.
In this case we have a cubic on each subinterval. To uniquely determine a cubic we need 4 conditions.

Consequently, we need to add 2 points in the interior of each interval to interpolate so that with the 2 endpoints we will have a total of 4 points.

We add these points uniformly so that we add the point which is one-third of the interval length from the left endpoint and one-third of the interval length from the right endpoint.

For example, on the interval $[x_i, x_{i+3}]$ we have the cubic polynomial

$$C_i = a_i + b_i(x - x_i) + c_i(x - x_i)(x - x_{i+1}) + d_i(x - x_i)(x - x_{i+1})(x - x_{i+2})$$

This is the formula we code on each interval once we have the coefficients.

To determine the coefficients we proceed as in the linear and quadratic case. That is, we determine the formulas from the interpolation conditions.
\[ C_i(x_i) = y_i \quad C_i(x_{i+1}) = y_{i+1} \]

\[ C_i(x_{i+2}) = y_{i+2} \quad C_i(x_{i+3}) = y_{i+3} \]

Note that here \( \Delta x_i = x_{i+3} - x_i \) and by construction \( \frac{1}{3}\Delta x = x_{i+1} - x_i = x_{i+2} - x_{i+1} = x_{i+3} - x_{i+2} \)
Piecewise Cubic Hermite Interpolation

• We now want to incorporate cubic Hermite interpolation.
• Here we want to interpolate both the function values at each endpoint of our interval as well as the derivative values.
• We will add the derivative values as a component of our derived type.
• The node and interval numbering will be the same as piecewise linear interpolation.
• From our previous work we know that on the interval \([x_i, x_{i+1}]\) the polynomial has the form

\[ a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^2(x - x_{i+1}) \]

• On each interval we have 4 coefficients so we have to allocate memory.
• Once we know the coefficients, then we can evaluate the polynomial with
We first need to determine analytically the coefficients on each interval which we need to code.

Let \( H_i(x) \) denote the cubic Hermite polynomial on the interval \([x_i, x_{i+1}]\). Then

\[
H_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^2(x - x_{i+1})
\]

and

\[
H'_i(x) = b_i + 2c_i(x - x_i) + d_i\left(2(x - x_i)(x - x_{i+1}) + (x - x_i)^2\right)
\]

Satisfying the interpolation conditions gives:

\[
H_i(x_i) = y_i \Rightarrow a_i = y_i
\]

\[
H'_i(x_i) = y'_i \Rightarrow y'_i = b_i
\]

\[
H_i(x_{i+1}) = y_{i+1} \Rightarrow y_{i+1} = y_i + y'_i \Delta x + c_i \Delta x_i^2 + 0 \Rightarrow c_i = \frac{1}{\Delta x_i^2}\left(y_{i+1} - y_i - y'_i \Delta x_i\right)
\]
\[ H'_i(x_{i+1}) = y'_{i+1} \Rightarrow y'_{i+1} = y'_i + 2c_i \Delta x_i + d_i \left( 0 + \Delta x_i^2 \right) \]

\[ \Rightarrow d_i = \frac{1}{\Delta x_i^2} \left( y'_{i+1} - y'_i - 2c_i \Delta x_i \right) \]

where again \( \Delta x_i = x_{i+1} - x_i \).

- So we have our formulas to code for the coefficients.
- In addition to the given function we are interpolating we have to include a routine for its derivatives since we are interpolating these too.
- **Warning** In our testing code suppose we add a loop that has statements like

```python
if (flag_method == 'cubichermite') then
    poly % yprime (i ) = fprime(xi )
```

where \texttt{fprime} is our function to calculate the derivative and \texttt{xi} is the \( x \) coordinate of the node we are evaluating at. This can cause a problem. Why? What can we do to fix it?
Project V

• For the project in this part of the course, you will complete the quadratic module, write a cubic module and the cubic Hermite module.

• You will then generate piecewise linear, piecewise quadratic, piecewise cubic and piecewise cubic Hermite interpolants to a given function. You will plot and compute errors.
Part VI

• Before we begin the next part of the course we are going to take Thursday and do a couple of things.

• First we are going to review some other types of loops (we have only really concentrated on the counter-controlled loop), and named do constructs and if constructs.

• Next we will see how to time different sections of our code; this is useful for benchmarking codes and comparing different methods.

• Then I will show you a good source for finding “library” routines and demonstrate how easy it is to use this routines. We will do more of this later.

• Part VI, which we will start on Tuesday is concerned with solving ordinary differential equations. For example, for initial value problems we know a starting value for our unknown, say we know \( y(t) \) at \( t_0 \), and we have a formula for the rate of change of \( y(t) \) (i.e., we know \( y'(t) \)) and our goal is to find \( y \) at any value of \( t > t_0 \). We will also consider two point boundary value
problems where we know \( y(x) \) at \( x = a \) and \( x = b \) and we know how \( y''(x) \) behaves in the interval \((a, b)\). Our goal is to find \( y(x) \) for all \( a < x < b \).
Classwork

• Download the file `class_pw_interpolation.f90` which contains the module `class_pw_linear_inter` and some routines in `class_pw_quadratic_inter` plus the search routines.

• Write a file to test out the program using a continuous piecewise linear interpolant to $\sin x^2$ on the interval $[0, 2\pi]$. Run the code for 8 and 16 intervals for linears. Compare the interpolant and the given function at 101 evenly spaced points in the given interval. Calculate the error norm. Does it decrease as the number of intervals increases?