Interpolation:
The Lagrange basis

MATH2070: Numerical Methods in Scientific Computing I

Location: http://people.sc.fsu.edu/~jburkardt/classes/math2070_2019/interpolation_lagrange/interpolation_lagrange.pdf

---

Each Lagrange basis function touches the data at a single point.

In mathematics, we think of functions as formulas $y = f(x)$ that allow us to produce a y for any x. In reality, we often have a small set of data pairs (x_i, y_i), and we wish to create a simple formula $p(x)$, called an **interpolant**, that matches our data, and allows us to "predict" y values for new x values.

If there really was a formula $f(x)$ that generated our data, we want to compare $f(x)$ and $p(x)$ and see how well our interpolant approximated the actual formula.

The polynomial interpolation problem

Given n pairs of data values (x_i, y_i),

- *determine a polynomial $p(x)$ so that, for $1 \leq i \leq n$:*

$$p(x_i) = y_i$$

- *if the data was generated by a function $f(x)$, estimate the error $|f(x) - p(x)|$;*
- *consider the behavior of the error as n is increased;*

If only one data pair is known, the interpolant is the corresponding constant function: $p(x) = y_1$. Things get more interesting when $n = 2$:

1 The case of two data values:

If two sets of data are given, we are asking for a linear interpolant. We can write:

$$p(x) = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

As shown in class, if the data comes from a function $f(x)$, then we have

$$\text{error} = f(x) - p(x) = \frac{f''(\xi)}{2!}(x - x_1)(x - x_2)$$

In discussing Taylor series, we have seen errors bounded by $O(h^2)$; the quadratic factor $(x - x_1)(x - x_2)$ reminds us of such behavior, but here it is a little more complicated. When it is clear that we are speaking about a particular set of n data points, we may use the shorthand $\omega(x)$ to refer to the function formed by this product:

$$\omega(x) = (x - x_1)(x - x_1) \dots (x - x_n)$$

Although our linear function matches the data, it will be helpful if we rewrite that formula so that we can guess how to generalize it when there is more data to interpolate. We begin by formulating two basis functions:

$$\ell_1(x) = \frac{x - x_2}{x_1 - x_2}$$
$$\ell_2(x) = \frac{x - x_1}{x_2 - x_1}$$

which allows us to rewrite our linear function as:

$$p(x) = y_1\,\ell_1(x) + y_2\,\ell_2(x)$$

The second version of the formula makes it much easier to see what is going on. If we want to move to 3 data points, we just have to find a set of three basis functions with the right properties.

2 Example: Linear interpolation of e^x

```
1 x_data = [ 0, 1 ];
2 y_data = exp ( x_data );
3
4 x = linspace ( -0.5, 1.5, 101 );
5 y = exp ( x );
6 y2 = y_data(1) * ( x_data(2) - x ) / ( x_data(2) - x_data(1) ) ...
7    + y_data(2) * ( x - x_data(1) ) / ( x_data(2) - x_data(1) );
8 plot ( x, y, 'b-', x, y2, 'r-' )
```

Listing 1: exp_linear.m

We see that given any pair of (x, y) data values, we can write a formula for a linear polynomial that matches that data. Can it be this easy when there is more data?

3 Example: Quadratic interpolation of e^x

If we have three pairs of data, we need three basis functions, with the properties

$$\begin{array}{lll} \ell_1(x_1) = 1 & \ell_1(x_2) = 0 & \ell_1(x_3) = 0 \\ \ell_2(x_1) = 0 & \ell_2(x_2) = 1 & \ell_2(x_3) = 0 \\ \ell_3(x_1) = 0 & \ell_3(x_2) = 0 & \ell_3(x_3) = 1 \end{array}$$

The formula for $\ell_1(x)$ is simply

$$\ell_1(x) = \frac{x - x_2}{x_1 - x_2}\,\frac{x - x_3}{x_1 - x_3}$$

and you should see that this suggests formulas for all three basis functions. But instead of writing them out, let's try to automate that process:

```
function value = ell ( j, x_data, x )
% ell evaluates the j-th Lagrange basis function.
  value = ones ( size ( x ) );
  n = length ( x_data );

  for i = 1 : n
    if ( i ~= j )
      value = value .* ( x - x_data(i) ) / ( x_data(j) - x_data(i) );
    end
  end

  return
end
```

Listing 2: ell.m

```
x_data = [ 0, 0.5, 1 ];
y_data = exp ( x_data );

x = linspace ( -0.5, 1.5, 25 );
y = exp ( x );
y2 = y_data(1) * ell(1,x_data,x)  ...
   + y_data(2) * ell(2,x_data,x)  ...
   + y_data(3) * ell(3,x_data,x);

plot ( x, y, 'b-', ...
       x, y2, 'r-', ...
       x_data, y_data, 'k.', 'markersize', 25 );
```

Listing 3: exp_quadratic.m

4 Example: Any degree interpolation of e^x

It seems like our interpolant does a better job if we increase the number of data points. Assume we are staying in the interval $[a, b]$, which for this example is $[0, 1]$. Do we have to create a new function to do the interpolation every time we increase n? Look at the changes we make:

```
n = 4;
x_data = linspace ( 0.0, 1.0, n );
y_data = exp ( x_data );
y2 = zeros ( size ( x ) );

x = linspace ( -0.5, 1.5, 25 );
y = exp ( x );
for i = 1 : n
  y2 = y2 + y_data(i) * ell(i,x_data,x);
end

plot ( x, y, 'b-', ...
       x, y2, 'r-', ...
       x_data, y_data, 'k.', 'markersize', 25 );
```

Listing 4: exp_any.m

Recall that for n point interpolation, our error estimate is:

$$\text{error} = f(x) - p(x) = \frac{f^{(n)}(\xi)}{n!}(x - x_1)(x - x_2)...(x - x_n) = \frac{f^{(n)}(\xi)}{n!}w(x)$$

Looking at this formula for the error, can you suggest why the exponential interpolant seems so well behaved in $[0, 1]$ even as we increase the value of n?

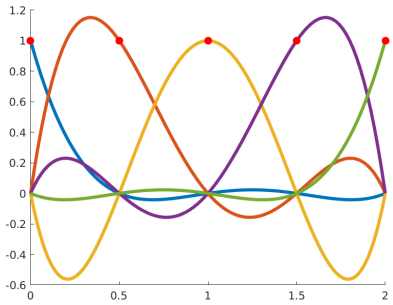
5 Exercise: Plot the Lagrange basis functions

Given an interval $[a,b]$, and assuming our data points are evenly spaced, we can compute and plot the individual basis functions.

```
function basis_lagrange ( a, b, n )
  x_data = linspace ( a, b, n );
  y_data = ones ( n, 1 );

  x = linspace ( a, b, 101 );

  clf ( );
  hold ( 'on' )
  for i = 1 : n
    y = ell ( i, x_data, x );
    plot ( x, y, 'linewidth', 3 );
  end
  plot ( x_data, y_data, 'r.', 'markersize', 25 );
  hold ( 'off' );

  return
end
```

Listing 5: basis_lagrange.m

Notice that each basis function is 1 at the correct place. Also notice that the basis functions can become negative, and they can exceed the value 1.



A set of 5 Lagrange basis functions.

Question: If I have $n = 5$ basis functions, what will I see if I plot $y(x) = \ell_1(x)+\ell_2(x)+\ell_3(x)+\ell_4(x)+\ell_5(x)$?

6 Example: Interpolate humps(x) with increasing n

When we are interpolating a function $f(x)$, it seems reasonable to expect that increasing the number of interpolation points will improve the quality of our interpolant. The only thing we are sure of is that $f(x)$ and $p(x)$ will agree at n points. We really don't know what happens in between. The function e^x is actually well behaved. Let's try interpolating the function `humps(x)` for a change.

To make it easy to vary n, we will write a function for which n is input:

```
function humps_lagrange ( n )

  x_data = linspace ( 0.0, 2.0, n );
  y_data = humps ( x_data );

```

```
  x = linspace ( 0.0, 2.0, 51 );
  y = humps ( x );
  y2 = zeros ( size ( x ) );
  for i = 1 : n
    y2 = y2 + y_data(i) * ell(i,x_data,x);
  end

  plot ( x, y, 'b-', ...
         x, y2, 'r-', ...
         x_data, y_data, 'k.', 'markersize', 25 );
```

Listing 6: humps_lagrange.m

Try this function for $n = 5$, 9, 17 and 33:

1. does each interpolant seem to match the data?
2. do the interpolants get closer to $f(x)$ as n increases?
3. if our error estimate is still correct, what is a reasonable explanation for this behavior?

7 Omega

In the error estimate for interpolation using nodes $x_1, x_2, \ldots x_n$, we saw a factor which is usually termed "omega", or $\omega(x)$:

$$\omega(x) = (x - x_1) * (x - x_2) * \ldots * (x - x_n)$$

which can be programmed as:

```
function value = omega ( x_data, x )

  n = length ( x_data );
  value = ones ( size ( x ) );

  for i = 1 : n
    value = value .* ( x - x_data(i) );
  end

  return
end
```

Listing 7: omega.m

If we have a bound for $f^{(n)}(\xi)$, then $\omega(x)$ can suggest how the error might vary as a function of x. Let's look at the interval $[0, 4]$ and plot $\omega(x)$ for a sequence of values $n = 1, 3, 5, 9, 17, 33$:

```
a = 0.0;
b = 3.5;
x = linspace ( a, b, 101 );

clf ( );
hold ( 'on' );
grid ( 'on' );
for n = [ 1, 3, 5, 9, 17, 33 ]
  if ( n == 1 )
    xdata = ( a + b ) / 2.0;
  else
    xdata = linspace ( a, b, n );
  end
  w = omega ( xdata, x );
  plot ( x, w );
  label = sprintf ( 'omega(x) for n = %d', n );
  title ( label );
```

```
18    pause ( 5 );
19  end
20  hold ( 'off' );
```

Listing 8: omega_plot.m

The sequence of plots should convince you that, using our equally spaced data points, the $\omega(x)$ factor can blow up near the endpoints. This doesn't mean that we *must* have large error at the endpoints, but it does suggest that it might be much easier to have large errors near the endpoints of the interpolation interval.

8 No Assignment for this lab!