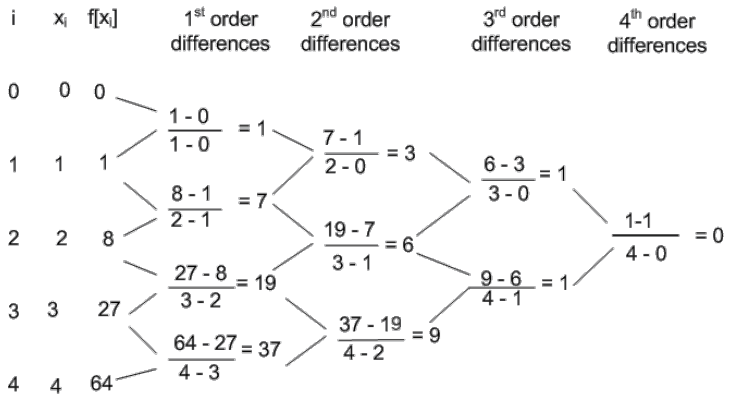
Interpolation:
The Divided Difference Method

MATH2070: Numerical Methods in Scientific Computing I

Location: http://people.sc.fsu.edu/~jburkardt/classes/math2070_2019/interpolation_divided_difference/interpolation_divide



A divided difference table repeatedly compares neighboring values.

Divided differences

- *Divided differences can interpolate n pairs of data (x_i, y_i);*
- *The divided difference table is constructed recursively;*
- *The divided difference polynomial can be evaluated and plotted;*
- *The divided difference table can easily add or drop a new data item;*
- *The divided difference polynomial can be converted to power form;*

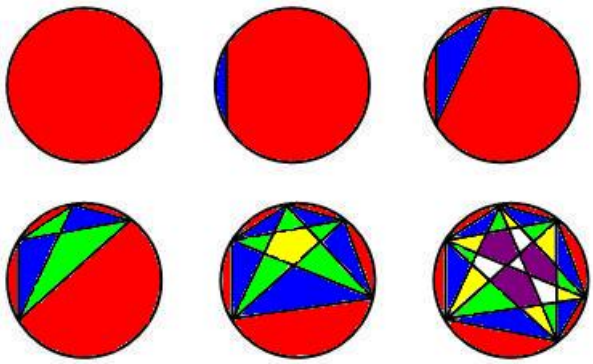
1 Two Pizza Problems

Joe, a lazy pizza shop owner was asked to slice a pizza into 16 pieces. The customer didn't specify that the pieces had to be the same size or shape. Joe discovered he could do this with just 5 cuts.

Joe also realized that using 0, 1 or 2 cuts results in 1, 2 or 4 pieces. Now he had the following partial table:

Cuts	0	1	2	3	4	5	6
Pieces	1	2	4	?	?	16	?

Joe is puzzled by the powers of 2 showing up in his table. He wonders if 8 shows up under 3 or 4 cuts, but he is too lazy to experiment. He would like a formula that reveals the maximum number of pieces that can be made from a given number of cuts.



Pizza pieces by connecting dots.

At Lina's pizza shop, she has a different procedure. She places n dots along the rim of the pizza, and then makes pieces by cutting a straight line between every pair of dots. One dot does nothing, leaving just 1 piece. Two dots allow her to create two pieces. Three dots, which make a triangular piece in the center of the pizza, actually create a total of 4 pieces. Continuing in this way, it would look like the number of pieces is simply 2 to the power of the number of dots minus 1. That is, until she looked at using 6 and 7 dots.

Dots	1	2	3	4	5	6	7	8
Pieces	1	2	4	8	16	31	57	?

Can we find a formula that explains these results, and predicts the number of pieces resulting from 8 dots?

Both puzzles can be seen as interpolation problems. We will see a way of dealing with it using divided differences and Newton's polynomial form!

2 Newton polynomial representation

The **Newton polynomial representation** is a special way of representing a polynomial. Instead of multiplying the i-th monomial, the coefficient c_i multiplies a product of i factors representing the distance from various "center" points.

A Newton polynomial can be written as:

$$p(x) = \sum_{i=0}^{d} c_i * \prod_{j=1}^{i} (x - z_j) \quad \text{Mathematical form, 0-based indexing of c}$$

Here, the three items of data are:

- `d`, the degree;
- `c(0:d)`, the coefficients;
- `z(1:d+1)`, the centers;

A general degree $d = 3$ polynomial in Newton form would look like:

$$\begin{aligned} p(x) = & \, c(0) \\ & + c(1) * (x - z(1)) \\ & + c(2) * (x - z(1)) * (x - z(2)) \\ & + c(3) * (x - z(1)) * (x - z(2)) * (x - z(3)) \end{aligned}$$

while a particular example would be

$$\begin{aligned} p(x) = & 0 \\ & + 1 * (x - 0) \\ & + 3 * (x - 0) * (x - 1) \\ & + 1 * (x - 0) * (x - 1) * (x - 2))) \end{aligned}$$

which is the Newton representation of $p(x) = x^3$ using `c = [0, 1, 3, 1]` and `z = [0, 1, 2, 3]`.

We could print a Newton polynomial based on the values of `c` and `z` as follows:

```
function newton_print ( c, z, title )

  n = length ( z );

  fprintf ( 1, '\n' );
  fprintf ( 1, '%s\n', title );
  fprintf ( 1, '\n' );

  fprintf ( 1, '  p(x) =                                %14f\n', c(1) );
  for i = 2 : n
    fprintf ( 1, '       + ( x - %14f) * ( %14f\n', z(i-1), c(i) );
  end

  fprintf ( 1, '  ' );
  for i = 1 : n - 1
    fprintf ( 1, ')' );
  end
  fprintf ( 1, '\n' );

  return
end
```

Listing 1: newton_print.m: Print a Newton polynomial.

3 Newton polynomial evaluation

Because MATLAB does not allow 0-indexes, we increment the coefficient indices by 1, so the formula becomes:

$$p(x) = \sum_{i=0}^{d} c_{i+1} * \prod_{j=1}^{i} (x - z_j) \quad \text{MATLAB form, 1-based indexing of c}$$

Just as we did for a standard polynomial, we can evaluate a Newton polynomial efficiently by starting with the last coefficient times the last factor, and then gradually working down to the first coefficient:

```
function value = newton_value ( c, z, x )
  d = length ( c ) - 1;
  value = c(d+1);
  for i = d : -1 : 1
    value = value * ( x - z(i) ) + c(i);
  end
  return
end
```

Listing 2: Evaluate a Newton polynomial (scalar x).

If we want to be allowed to use a vector `x` as input, then we need to make two adjustments to the code:

```
function value = newton_value ( c, z, x )
  d = length ( c ) - 1;
  value = c(d+1) * ones ( size(x) );        <-- #1
  for i = d : -1 : 1
    value = value .* ( x - z(i) ) + c(i);   <-- #2
  end
  return
end
```

Listing 3: newton_value.m: Evaluate a Newton polynomial (vector x).

4 Exercise: Newton polynomial evaluation

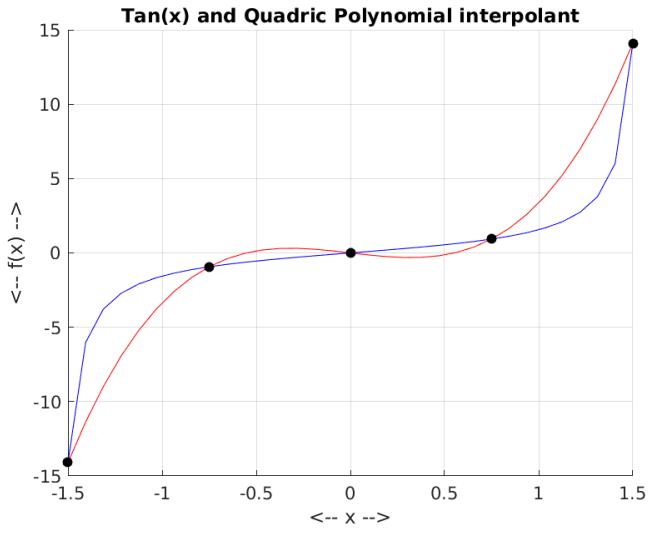
The Wikipedia page for *Newton polynomial* describes an interpolant to the tangent function, with data:

- `d = 4`, the degree;
- `c(1:d+1)`=[-14.1014, 17.5597, -10.8784, 4.83484, 0.0];
- `z(1:d+1)`=[-1.5, -0.75, 0.0, 0.75, 1.5];

We can tabulate this polynomial over $[-1.5, 1.5]$, and plot the function and polyomial interpolant:

```
c = [ -14.1014, 17.5597, -10.8784, 4.83484, 0.0 ];
z = [ -1.5,     -0.75,     0.0,    0.75,    1.5 ];
tz = tan ( z );
nx = 33;
x = linspace ( -1.5, +1.5, nx );
px = newton_value ( c, z, x );
tx = tan ( x );
for i = 1 : nx
  fprintf ( 1, ' %f  %f  %f\n', x(i), px(i), tx(i) );
end
hold ( 'on' );
plot ( x, px, 'r-', x, tx, 'b-' )
plot ( z, tz, 'k.', 'markersize', 20 );
hold ( 'off' );
```

Listing 4: newton_value_test.m: Evaluate the Wikipedia Newton polynomial.



A 4th degree Newton polynomial interpolates $\tan(x)$ *at 5 points.*

5 Newton polynomial to power polynomial

Given a d-degree Newton polynomial with data `c, z`, we can compute the coefficients of the corresponding power form polynomial.

```
function c2 = newton_to_power ( c1, z1 )

  n = length ( c1 );
  c2(1:n) = c1(1:n);
  for j = 1 : n - 1
    for i = n - 1 : -1 : j
      c2(i) = c2(i) - z1(i-j+1) * c2(i+1);
    end
  end

  return
end
```

Listing 5: newton_to_power.m: Convert Newton to power form.

Using this function, we can determine the power polynomial form of the Wikipedia example above:

```
c1 = [ -14.1014, 17.5597, -10.8784, 4.83484, 0.0 ];
z1 = [ -1.5,     -0.75,     0.0,    0.75,    1.5 ];
c2 = newton_to_power ( c1, z1 );
Output:

  c2 = -0.0000   -1.4775   -0.0000    4.8348         0
```

So our interpolant can also be written as

$$p(x) = -1.4775x + 4.8348x^3$$

6 The divided difference table

Suppose we have n pairs of data (x_i, y_i), and we want to compute an interpolating polynomial $p(x)$. We have already seen one technique for doing so, using Lagrange polynomials as the basis. However, the Lagrange polynomials can be expensive to evaluate, and if we want to add an $n+1$-th point to our data set, the new Lagrange representation must be completely recomputed. The divided difference approach will offer an interpolation method that is simpler to evaluate, and is flexible enough to add new data with little extra cost.

To construct a divided difference table, we can imagine starting with two columns of data, the x and y values. To the right of the y column, we construct column 1, the first-order differences, by computing the ratio of the differences of neighboring y and x values. Thus, column 1 begins with $\frac{y_2-y_1}{x_2-x_1}$ and terminates with $\frac{y_n-y_{n-1}}{x_n-x_{n-1}}$, a total of $n-1$ entries. Column 2 is formed by the ratio of differences of the column 1 values, divided by the differences of x values that are two steps apart, and so on.

Symbolically, we are filling in this table:

$$\begin{array}{llllll}
x_0 & y_0 & & & & \\
 & & \Delta y_0 & & & \\
x_1 & y_1 & & \Delta^2 y_0 & & \\
 & & \Delta y_1 & & \Delta^3 y_0 & \\
x_2 & y_2 & & \Delta^2 y_1 & & \Delta^4 y_0 \\
 & & \Delta y_2 & & \Delta^3 y_1 & \\
x_3 & y_3 & & \Delta^2 y_2 & & \\
 & & \Delta y_3 & & & \\
x_4 & y_4 & & & &
\end{array}$$

It will help to work with an example using real data. Thus, suppose our data was `x=[0,1,2,4,5]` and `y=[1,13,13,13,21]`.

$$
\begin{array}{llll}
0 & 1 & & \\
 & & \frac{12}{1} = 12 & \\
1 & 13 & & \frac{-12}{2} = -6 \\
 & & \frac{0}{1} = 0 & \\
2 & 13 & & \frac{0}{3} = 0 \\
 & & \frac{0}{2} = 0 & \\
4 & 13 & & \frac{8}{3} = 2.66 \\
 & & \frac{8}{1} = 8 & \\
5 & 21 & &
\end{array}
\qquad
\begin{array}{ll}
 & \\
 & \\
\frac{-6}{4} = -1.5 & \\
 & \frac{2.16}{5} = 0.432 \\
\frac{2.66}{4} = 0.66 & \\
 & \\
 & \\
 &
\end{array}
$$

which we can compare by typing

```
t = data_to_difference_table ( ([0,1,2,4,5], [1,13,13,13,21] )
```

7 Exercise: Write a function to compute difference tables

Although it's easy to fill in a divided difference table by hand, the correct indexing can be a little tricky. Keep in mind that the n items of y data go in column 1, and then we have to compute columns 2 through n. Column 1 is a copy of the n entries of y. Column 2 involves $n-1$ differences of entries in column 1. Column j involves $n+1-j$ differences of entries in column $j-1$.

```
function difference_table
input
  x
  y
output
  t

  n = number of x values
  t = n x n array of zeros
  set first column of t to y

  for j from 2 to n                <-- Fill in columns 2 to n
    for i from 1 to n + 1 - j      <-- Fill in rows 1 to n+1-j
      t(i,j) <-- ( t(i+1,j-1) - t(i,j-1) ) / ( x(i+j-1) - x(i) )
    end loop
  end loop

end function
```

Listing 6: Pseudocode for a divided difference table.

Test your code by seeing if you can recreate the table on the first page, with `x=[0,1,2,3,4]` and `y=0,1,8,27,64]`.

8 Data to Newton coefficients

It turns out that we can use the first row of the divided difference table as the coefficient vector `c` for a Newton polynomial interpolant to the data (x_i, y_i), using the x_i data as our `z` centers.

We can write a new version of the difference table function, which computes all the differences as before but only saves the first row:

```
function [ c, z ] = data_to_newton ( x, y )

n = length ( x );
c(1:n) = y(1:n);
for i = 2 : n
  for j = n : -1 : i
    c(j) = ( c(j) - c(j-1) ) / ( x(j) - x(j+1-i) );
  end
end
z = x;
```

Listing 7: data_to_newton.m: Compute Newton coefficients from data.

To verify that we get what we need, try the command:

```
[ c, z ] = data_to_newton ( [0,1,2,3], [0,1,8,27] )
```

9 Newton coefficients to Data

If we have a set of Newton coefficients and centers, we can easily recover the data. All we have to do is evaluate the Newton polynomial at each of the Newton centers:

```
function [ x, y ] = newton_to_data ( c, z )

  x = z;
  y = newton_value ( c, z, x );

  return
end
```

Listing 8: newton_to_data.m: Recover data from Newton polynomial.

For example, the command

```
[ x, y ] = newton_to_data ( [0,1,3,1,0], [0,1,2,3,4]  )
```

will recover the data from the table displayed at the beginning of this document.

10 Creating an interpolant to data

To understand how divided differences might be used, let's take some raw data, use our `data_to_newton()` function to construct the Newton coefficients of the interpolant, and then our `newton_value()` function to evaluate it at 101 plot points.

```
  xdata = linspace ( 0.0, 2.0, 11 );
  ydata = humps ( xdata );
  [ c, z ] = data_to_newton ( xdata, ydata );

  xplot = linspace ( 0.0, 2.0, 101 );
  yplot = newton_value ( c, z, xplot );
  plot ( xplot, yplot );
```

Listing 9: humps_newton_linspace.m: Plot a divided difference interpolant to humps().

With 11 points, our interpolant is correctly matching the data. However, it seems to swing away from the true function near the left hand side of the plot. What happens if we use 21 points instead?

Things get much, much worse. Is this an error? Actually, it looks like the interpolant is matching the data, but in between, its behavior is much worse near both endpoints. Endpoint problems should remind us that

when performing interpolation, evenly spaced data is often not ideal. If possible, the function should be sampled more heavily near the endpoints.

11 Chebyshev spacing

In our discussion of polynomial interpolation using the Legendre basis, we salready encountered the situation where equally spaced data resulted in interpolants that went crazy near the endpoints. At that time, we noted that a better spacing can be determined related to the zeros of the Chebyshev polynomials. If the interval is $[-1,+1]$, then these points are simply the cosines of angles equally spaced between 0^{circ} and 180^{circ}. A simple linear transformation will map these points to any interval $[a,b]$.

We have a function to do this, which has an interface similar to MATLAB's `linspace()`:

```
function x = chebyspace ( a, b, n )

  x = zeros ( n, 1 );
  if ( n == 1 )
    x(1) = ( a + b ) / 2.0;
  else
    for i = 1 : n
      theta = ( n - i ) * pi / ( n - 1 );
      c = cos ( theta );
      x(i) = ( ( 1.0 - c ) * a  ...
             + ( 1.0 + c ) * b ) ...
             /   2.0;
    end
  end

  return
end
```

Listing 10: chebyspace.m: Chebyshev spacing of points.

We can repeat our interpolation experiment for the *humps()* function, using 11, 21, 41 points, and see if we have reduced the wild oscillations that we saw for evenly spaced data. Use the file *humps_newton_chebyspace.m* to do this.

12 Adding new data

An advantage of the Newton form for an interpolating polynomial is that we can add a new data item cheaply. Assume that we have already computed `c1, z1`, the coefficients and centers for an interpolant of degree `d1`. Now we have an additional data pair (x,y) that we want to interpolate. We can set up a new difference table, in which x is the first center, and y is the first coefficient, and then use the old coefficients to complete the new table.

While this is hard to describe in words, it becomes much clearer if it is demonstrated on the board! Suppose our initial table is

```
1  2.71828
                4.6708
2  7.38906              4.0129
              12.6965             2.2984
3  20.0855            10.9081
              34.5126
4  54.5982
```

To evaluate the table, we only need to save `c=[2.71828, 4.6708, 4.0129, 2.2984 ]`. Now if want to add the data x=5, y=148.4132, the easiest way is to insert it at the beginning of our table, in which case we only need to compare our new data to the old coefficients in order to create an updated set:

```
5    148.4132
                36.4237
1     2.71828             10.5843
                4.6708              3.2857
2     7.38906             4.0129              0.9873
                                    2.2984
3    20.0855

4    54.5982
```

so that our new data is `c=[148.4132, 36.4237, 10.5843, 3.2857, 0.9873]` and `z=[5,1,2,3,4]`.

A code to carry out this update could look like this:

```
function [ c2, z2 ] = newton_append ( c1, z1, x, y )

  d1 = length ( z1 ) - 1;

  d2 = d1 + 1;
  c2 = zeros(d2+1,1);
  c2(2:d2+1) = c1(1:d1+1);
  c2(1) = y;

  z2 = zeros(d2+1,1);
  z2(2:d2+1) = z1(1:d1+1);
  z2(1) = x;

  for i = 2 : d2 + 1
    c2(i) = ( c2(i) - c2(i-1) ) / ( z2(i) - z2(1) );
  end

  return
end
```

Listing 11: newton_append.m: Add another data value.

Of course, we could simply have recomputed the entire table from the full data set, but this would take $O(n^2)$ work, while our append procedure requires $O(n)$ work instead.

We can test this procedure by starting with data `x=[1,2,3,4]`, `y=exp(x)`, and then adding the data item $(5, exp(5))$.

13 Computing Challenge (Not an assignment!)

- For Joe's pizza problem, use the divided difference method to define an interpolant to the data in Joe's table. Evaluate this interpolant at 3, 4, and 6 to fill in the missing entries.
- For Lina's pizza problem, use the divided difference method to define an interpolant to the data in Lina's table. Evaluate this interpolant to predict the number of pieces defined by using 8 dots.

This challenge will not be collected! If you are interested, we can compare my answer with yours.

This is the last lab meeting for Math 2070!