1 Lecture 8: Interpolating polynomials.

1.1 Horner's method

Before turning to the main idea of this part of the course, we consider how to evaluate a polynomial.

Recall that a polynomial is an expression of the form

$$\sum_{k=0}^{n} a_k x^k.$$

A naive program to evaluate this expression is

```
function p = polyval(a,x)
[rows cols] = size(a)
 n = cols
 p = 0
 for k = 1:n
 p = p + a(k)*(x^(k-1))
end
```

Note that since matlab indexes arrays from 1 to n, we end up with a polynomial of degree $n-1$,

$$a_n x^{n-1} + \ldots + a_1.$$

This function uses $n-1$ multiplications and $n-1$ applications of the exponential function `x^k`. As mentioned earlier in this class, this does not seem right because we use the complicated exponential function in order to evaluate a simpler polynomial. We can eliminate the exponential function by writing

```
function p = polyval(a,x)
[rows cols] = size(a)
n = cols
t = x
p = a(1)
for k = 2:n
 p = a(k)*t + p
 t = x*t
end
```

Note that this simplifies the program to only use $2n-2$ multiplications.

However, there is a slightly simpler way that is known as Horner's method. This is also the procedure behind synthetic division. To understand this last method,

observe that if $p_k(x) = a_n x^k + a_{n-1}x^{k-1} + \ldots + a_{n-k}$, then we may set $p_{k+1}(x) = x * p_k(x) + a_{n-k-1}$ and simplifying we obtain that

$$p_{k+1} = a_n x^{k+1} + a_{n-1}x^k + \ldots + a_{n-k}x + a_{n-k-1}.$$

Thus the matlab function

```
function p = polyval(a,x)
%Use Horner to evaluate the polynomial
%a(1)x^(n-1) + a(2) x^(n-2) + ...+ a(n)

[rows cols] = size(a)
n = cols;
p = a(1);
for k = 2:n
 p = p*x + a(k);
end
```

will return the value of

$$a(1)x^{n-1} + a(2)x^{n-2} + \ldots + a(n).$$

This will require only $n-1$ multiplications and may help to explain why matlab's built-in function `polyval` uses $a(1)$ as the coefficient of the highest order term.

Exercise. Can you adapt this method to evaluate expressions of the form

$$\sum_{k=0}^{n} a_k(t_1 \cdot t_2 \ldots t_k)?$$

Note that if $a_k = 1$ for $k = 0 \ldots n$, $t_0 = 1$ and $t_k = x/k$ for $k = 1 \ldots n$, this will give a Taylor polynomial for the exponential function.

1.2 The interpolation problem.

Throughout this section, we are given a finite set of points in the plane, $\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$ and we ask if we can find a polynomial of degree n, p_n so that $p_n(x_k) = y_k$, $k = 0 \ldots n$. If such a polynomial exists, then we say that p_n is an interpolating polynomial for these points. The x coordinates are called the *nodes*. Note that we must that the nodes are $n+1$ different values in order for this to be possible: we cannot expect to find a polynomial for $p(x)$ takes two different values at one value of x. If the values $y_k = f(x_k)$ for some function f, we will speak about interpolating the function f at the nodes $x_0 \ldots x_k$

Why do we care? We list several applications of polynomial interpolation.

- This provides one way to approximate a general function by a simpler function. When we study numerical integration, we will see that we can produce useful integration rules by interpolating a function by a polynomial, integrating the polynomial and decreeing that the result is an approximate value for the function.

- One can view interpolation as a means of data compression. Rather than store a large table of values, say the value of $\sin(x)$ for all possible machine numbers x (there are about 2^{64} machine numbers!), we may store a few values and then use an interpolating polynomial to compute the rest. Or, even better, we would store a program to compute values of the interpolating polynomial. This is not so far from what was done until the 1970's when every advanced mathematics textbook contained tables of values of trigonometric functions and students were trained to use linear interpolation to compute $\sin x$ for angles that were not on the table. It is also not so far from what goes on inside a chip today.

- Interpolating polynomials can also help in manipulating functions in other ways. To approximate a derivative when we only have a few points, we can interpolate and then differentiate the polynomial. To find the inverse, we can compute a few pairs $(x_k, f(x_k))$ and then interpolate the pairs $(f(x_k), x_k)$.

- Still we must be careful. High degree interpolating polynomials do not always approximate the function very well. Thus, there are other techniques for approximating functions that we will need to study in the future.

Whenever we consider a new problem, we should ask: Does the problem have at least one solution? Does it have more than one solution? To answer the first of these questions, we recall a fundamental fact from algebra: A nonzero polynomial of degree n has at most n roots. This is because for each root, r, we may find a factor $(x - r)$ and since the degree is n and a polynomial of can only have n such factors.

Theorem 1 *(uniqueness) If we have two polynomials p and q which satisfy*

$$p(x_k) = q(x_k), \qquad k = 0, \ldots, n$$

then $p(x) = q(x)$ for all x.

Proof. The polynomial $p(x) - q(x)$ is of degree at most n and has $n + 1$ roots. The only such polynomial is 0. ∎

Exercise. The conclusion of the uniqueness theorem is that p and q are equal for all x. If we write $p(x) = a_0 + a_1 x + \ldots + a_n x^n$ and $q(x) = b_0 + b_1 x + \ldots b_n x^n$, can we conclude that $a_j = b_j$ for $j = 0 \ldots n$.

A more interesting question is: Does there exist an interpolating polynomial? One way to think about this is to write

$$p_n(x) = a_n x^n + \ldots + a_0.$$

Then the conditions

$$\begin{aligned} a_n x_0^n + \ldots a_0 &= y_0 \\ a_n x_1^n + \ldots a_0 &= y_1 \\ &\ldots \\ a_n x_n^n + \ldots a_0 &= y_n \end{aligned}$$

give us a system of $n+1$ equations for the $n+1$ unknown coefficients. We can solve this system if we know the determinant

$$\begin{vmatrix} 1 & x_0 & \ldots & x_0^n \\ 1 & x_1 & \ldots & x_1^n \\ & & & \\ 1 & x_n & \ldots & x_n^n \end{vmatrix}$$

is nonzero. This is a well-known fact, the determinant is called the van der Monde determinant and is nonzero when all the nodes are distinct. However, this is not the most useful approach to finding the polynomial.

1.3 Lagrange polynomials.

One approach to finding interpolating polynomials is to first interpolate in the case when only one of the y_k's is one and the rest are zero. We define the kth Lagrange polynomial for the nodes $\{x_0, \ldots, x_n\}$ as the polynomial which has $\ell_k(x_k) = 1$ and $\ell_k(x_j) = 0$ for $j \neq k$. This can be written simply as $\ell_k(x_j) = \delta_{jk}$ where δ_{jk} is the Kronecker delta defined by

$$\delta_{jk} = \begin{cases} 1, & j = k \\ 0, & j \neq k. \end{cases}$$

Our uniqueness theorem tells us that there can be at most one Lagrange polynomial. We show that there is at least one by writing one down:

$$\ell_k(x) = \Pi_{j \neq k} \frac{(x - x_j)}{(x_k - x_j)}.$$

Our assumption that the nodes are distinct tells us that we never divide by zero and thus this formula has sense.

Exercise. How many multiplications and divisions are needed in order to evaluate the Lagrange polynomial at one value of x?

Once we have the Lagrange polynomials, it is easy to see that interpolating polynomials exist.

Theorem 2 *Given $n+1$ distinct nodes $\{x_0, \ldots x_n\}$ and values $y_0, \ldots y_n$, we can find an interpolating polynomial of degree n, p_n which satisfies*

$$p_n(x_k) = y_k, \quad k = 0 \ldots n.$$

Proof. We may write the polynomial in the Lagrange form:

$$p_n(x) = \sum_{k=0}^{n} \ell_k(x) y_k.$$

∎

Example. Find the Lagrange polynomial for the nodes $-h, 0, h$. Use them to find a polynomial for which $p(0) = 2$ and $p(h) = p(-h) = 1$.

Exercise. Use Lagrange polynomials and the method outlined above to find the linear function which has $p_1(a) = y_a$ and $p_1(b) = y_b$.

Exercise. How many multiplications does it take to evaluate the Lagrange form of the interpolating polynomial? How many coefficients do we need to store?

1.4 Newton form of the interpolating polynomial.

In this section, we look at another form of the interpolating polynomial. This form has the advantage that it is easy to evaluate as compared to the Lagrange form. And, as we shall see, it is easy to compute the coefficients.

The *Newton form of the interpolating polynomial* is

$$p_n(x) = \sum_{j=0}^{n} a_j \Pi_{i=0}^{j-1}(x - x_j). \tag{1}$$

In the equation (1), the product $\Pi_{j=0}^{-1}(x - x_j)$ is defined to be 1.

We make several observations:

- We may write this easily in nested form and thus it is easy to write a program to evaluate this expression.
- If we only take the first $k+1$ terms in p_n,

  $$p_k(x) = \sum_{j=0}^{k} a_j \Pi_{i=0}^{j-1}(x - x_j),$$

  then $p_k(x_j) = p_n(x_j)$ for $j = 0 \ldots k$. Thus we can find the coefficients $a_0 \ldots a_k$ by solving a simpler interpolation problem.

Example. Write $p_2(x)$ in nested form for easy evaluation.

Solution. The solution is

$$p_2(x) = ((x - x_2) * a_2 + a_1) * (x - x_1) + a_0.$$

∎

We describe how to find the coefficients a_j in the Newton form. As before we have $n + 1$ distinct nodes $\{x_0, \ldots x_n\}$ and we want to find p_n, a polynomial of degree n as in (1) so that $p_n(x_k) = y_k$. If we substitute the nodes x_k into p_n, we obtain the system of equations

$$\begin{aligned} a_0 &= y_0 \\ a_0 + a_1 * (x_1 - x_0) &= y_1 \\ a_0 + a_1 * (x_2 - x_0) + a_2 * (x_2 - x_0) * (x_2 - x_1) &= y_2 \\ &\ldots \\ \sum_{j=0}^{n} a_j \Pi_{k=0}^{j-1} (x_n - x_j) &= y_n \end{aligned}$$

Note that this system is in triangular form and fairly easy to solve. If we know $a_0 \ldots a_k$, we can use the first $k + 2$ equations to obtain the coefficient a_{k+1}.

Example. Find the Newton form of the interpolation polynomial for the table

x	y
1	1
2	0
4	4

Solution. We are looking for a second degree polynomial of the form

$$p_2(x) = a_0 + a_1(x - 1) + a_2(x - 1)(x - 2).$$

Since $p_2(1) = a_0$, he condition $p_2(1) = 1$ tells us $a_0 = 1$. Then $p_2(2) = a_0 + a_1(2-1) = 1 + a_1$ so the equation $p_2(2) = 0$ tells us $a_1 = -1$. To find a_2 we have

$$p_2(4) = a_0 + a_1 * 3 + a_2 * 3 * 2 = -2 + a_2 * 6 \qquad and p_2(4) = 4.$$

which implies that $a_2 = 1$. Thus the Newton form is

$$p_2(x) = 1 - (x - 1) + (x - 1)(x - 2).$$

The above polynomial simplifies to $p_2(x) = (x-2)^2$ and it is easy to check that this is correct. Note that the simplification is helpful for human computation. For machine computation, we prefer to use the nested form. ∎

1.5 Divided differences

In this section, we observe a simple way to compute the coefficients a_j in the Newton form of the interpolating polynomial. For this section, we suppose that we have distinct nodes $\{x_0 \dots x_n\}$ and that we have a function f. Our interpolation problem is to find a polynomial $p_n(x)$ so that $p(x_k) = f(x_k)$.

As we observed in the previous section, the coefficient a_k in the Newton polynomial depends only on the nodes $(x_0, \dots, x_k)$ and the $f(x_0) \dots f(x_k)$. We introduce the notation

$$a_k = f[x_0, \dots x_k].$$

For reasons which will become clear, we call $f[x_0, \dots x_k]$ a *kth divided difference of* f. The following two facts serve as a basis for inductively computing the divided differences.

We begin with the easy bit.

Proposition 1 $f[x_k] = f(x_k)$

Exercise. Find $f[x_0, x_1]$.

More interesting is the following:

Theorem 3 *The divided differences satisfy*

$$\frac{f[x_1, \dots x_k] - f[x_0, \dots, x_{k-1}]}{x_k - x_0} = f[x_0, \dots, x_k].$$

Proof. Let p_k interpolate f at $x_0 \dots x_k$. Let p_{k-1} interpolate f at $x_0 \dots x_{k-1}$ and let q interpolate f at $x_1 \dots x_k$. By the definition of the divided difference, we have

- $f[x_0, \dots, x_k]$ is the coefficient of x^k in p_k.
- $f[x_0 \dots x_{k-1}]$ is the coefficient of x^{k-1} in p_{k-1}.
- $f[x_1 \dots x_k]$ is the coefficient of x^{k-1} in $q(x)$.

One may verify that

$$r(x) = q(x) + \frac{x - x_k}{x_k - x_0}(q(x) - p_{k-1}(x))$$

interpolates f at $x_0 \dots x_k$. At x_0, we have $r(x_0) = q(x_0) - (q(x_0) - p_{k-1}(x_0)) = p_{k-1}(x_0)$. At $x_1 \dots x_k$, $r(x_j) = q(x_j)$ for $j = 1 \dots k$. Thus by uniqueness,

$$p_k(x) = q(x) + \frac{x - x_k}{x_k - x_0}(q(x) - p_{k-1}(x)).$$

Examining the coefficient of x^k on each side gives the conclusion of the proposition. ∎

These propositions are the basis of a simple scheme to compute the coefficients in the Newton polynomial. We work through one example by hand.

Example. Find the interpolating polynomial for the values

x	1	2	4	6
y	-1	0	8	64

Solution. We use the divided difference table as follows:

k	x	y	$f[x_k, x_{k+1}]$	$f[x_k, x_{k+1}, x_{k+2}]$	$f[x_k, \ldots, x_{k+3}]$
0	1	-1	1	1	1
1	2	0	4	6	
2	4	8	28		
3	6	64			

The coefficients a_k for the Newton form of the polynomial are in the top row.

$$p_3(x) = -1 + 1 * (x-1) + 1 * (x-1) * (x-2) + 1 * (x-1) * (x-2) * (x-4).$$

Which simplifies to $p_n(x) = (x-2)^3$. ∎

Exercise. If $p_5(x)$ is a polynomial of degree 6 and $x_0, \ldots, x_10$ are distinct nodes, explain why all the 6th order divided $f[x_j, \ldots, x_{j+6}]$ differences are zero. Explain why all the fifth order divided differences $f[x_j, \ldots, x_{j+5}]$ are the same.

November 25, 2004