Polynomial interpolation

Most functions occurring in scientific computing cannot be evaluated exactly. This means that such functions must be approximated by functions that can be evaluated exactly in finitely many steps, namely the polynomials (and, more generally, the piecewise polynomials and, even more generally, the piecewise rationals, and even these use polynomials). Further, the simplest polynomial approximation scheme is polynomial interpolation, which is the topic of Chapter 2 of the book. Here is a somewhat different (and faster-moving) description of the basic ideas.

The **monomial of degree** k is, by definition, the function $x \mapsto x^k$. (Mathematics has yet to develop a symbol for this function, so, I made one up, namely $()^k$.)

A polynomial is a weighted sum of (finitely many) monomials:

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n, \tag{1}$$

and the a_k are its **coefficients**. (1) is only one of several ways to label these coefficients. In `Matlab` (and in Algebra), the following alternative is standard:

$$p(x) = a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_n, \tag{2}$$

(Note how in this way, coefficient-index and x-exponent always sum to n.) Specifically, if `a = [`a_1`, `a_2`, ..., `a_n`]`, then `polyval(a,x)` returns the value at `x` of the polynomial (2). The textbook does something in between:

$$p(x) = a_1 + a_2 x + \cdots + a_n x^{n-1}. \tag{3}$$

However it is done, the **leading coefficient** of a polynomial is the nonzero coefficient of the monomial of highest degree appearing in p, and that degree is, by definition, the **degree** of the polynomial. There is also the notion of **highest** coefficient, but it depends how we view the polynomial: if we think of p as a polynomial of degree $< n$, then its highest coefficient is the coefficient of x^{n-1}.

What about the polynomial

$$p : x \mapsto 0,$$

i.e., the polynomial that is identically zero? It is said to be of degree -1; it has no leading coefficient. However, for any $n \geq 0$, its highest coefficient as a polynomial of degree $< n$ is 0.

Let `x` be an arbitrary vector, and let `a` be the n-vector that contains the coefficients of the polynomial p given in (2). Then the script

```
v = zeros(size(x));
for k=1:length(a)
   v = x.*v + a(k);
end
```

constructs the vector `v` of the same shape as `x` for which `v(i)` equals the value of p at `x(i)` for `i=1:length(x)`. (In fact, this script even works in case `x` is, more generally, an arbitrary matrix.) The script incorporates **nested multiplication** or **Horner's method** and is based on the observation that we can save multiplications by factoring (3) appropriately. E.g., for $n = 4$,

$$p(x) = ((a_1 * x + a_2) * x + a_3) * x + a_4.$$

(With the book's more standard formulation (3), the `for` statement would be slightly more complicated: `for k=length(a):-1:1` .)

Now assume that both `x` and `a` are column vectors. Then the statement

```
y = a(1)*x.^(n-1) + a(2)*x.^(n-2) + ... + a(n-1)*x +  a(n)
```

generates the column vector `y` which contains the values of p at the points in `x`, i.e., `y(i)`$= p($`x(i)`$)$, all `i`. This formulation is ready-made for matrix multiplication. If we define

```
V = [x.^(n-1) x.^(n-2) ... x  ones(size(x)) ];
```

to be the matrix with columns `x.^(n-1)`, `x.^(n-2)`, ..., `x.^0`, then we can write the calculation of `y = p(x)` as

```
y = V*a;
```

Of course, I don't propose to evaluate p this way; for that, the recommended method is nested multiplication aka Horner's method. But this matrix formulation is ideal for polynomial *interpolation*, as follows.

Suppose we are to find a polynomial of degree $< n$ that matches given values y_i at given data sites x_i, $i = 1, \ldots, n$. This means that we are seeking a column n-vector `a` so that `V*a = y`. If there is such a vector, then `Matlab` computes it this way:

```
a = V\y;
```

In effect, `matlab` solves the linear system `V*? = y` numerically, something to be discussed at greater length later.

Are we entitled to expect a solution to this linear system `V*? = y` regardless of the choice of the n-vectors `x` and `y`? Since `V` is a square matrix, we only have to check whether the *homogeneous* system `V*? = zeros(size(y))` has only the trivial solution. So, suppose `V*a` is a zero vector. This means that our polynomial p with coefficient vector `a` takes the value 0 at each of the points $x_1, \ldots, x_n$. Hence, *assuming that there are no repeats in the sequence* $x_1, \ldots, x_n$, our polynomial p of degree $< n$ has at least n distinct zeros. There is only one such polynomial, namely the polynomial with all its coefficients equal to zero. Since `V` is a square matrix, it follows that the linear system `V*? = y` has exactly one solution for every choice of `y`.

We have proved: *If $(x_1, \ldots, x_n)$ is a sequence with no repeats, then, for arbitrary $(y_1, \ldots, y_n)$, there is exactly one polynomial p of degree $< n$ that matches the given information in the sense that $p(x_i) = y_i$, $i = 1{:}n$.*

The matrix `V` (with the order of the columns inverted, i.e., as in the textbook) has been called the **Vandermonde** matrix (by Lebesgue). In `matlab`, one assembles it by a loop, making use of the fact that `x.^(k+1)` equals `x.*(x.^k)`:

```
n = length(x);
V = ones(n,n);
for k=n-1:-1:1
   V(:,k) = x.*V(:,k+1);
end
```

However, there is usually not much of a reason for assembling the Vandermonde since it is more efficient to construct the interpolating polynomial in Newton form rather than the power form.

Newton form of the interpolant

The Newton form of the polynomial interpolant is motivated as follows: Suppose we already know the polynomial interpolant p_{k-1} to f at the k points $x_1, \ldots, x_k$, i.e., $p_{k-1}(x_j) = f(x_j)$, $j = 1{:}k$, and we want to interpolate f also at x_{k+1}. Rather than starting from scratch, we propose to construct p_k in the form

$$p_k(x) = p_{k-1}(x) + c_{k+1}(x - x_1)(x - x_2) \cdots (x - x_k).$$

For, no matter how the coefficient c_{k+1} is chosen, this p_k will interpolate f at $x_1, \ldots, x_k$, and, to make it also interpolate at x_{k+1}, we only have to solve one equation in the one unknown, c_{k+1}. This gives

$$c_{k+1} = \frac{f(x_{k+1}) - p_{k-1}(x_{k+1})}{(x_{k+1} - x_1) \cdots (x_{k+1} - x_k)}. \tag{4}$$

If we start this procedure with $k = 1$ and, correspondingly, $p_0(x) = f(x_1)$, we can build up the interpolating polynomial p_{n-1} point by point and degree by degree in the form

$$p_{n-1}(x) = c_1 + c_2(x - x_1) + c_3(x - x_1)(x - x_2) + \cdots + c_n(x - x_1)(x - x_1) \cdots (x - x_{n-1}). \tag{5}$$

This way of writing a polynomial is called its **Newton form (with centers $x_1, \ldots, x_{n-1}$)**. It, too, invites use of *nested multiplication* since we can rewrite it as

$$p_{n-1}(x) = c_1 + (x - x_1)(c_2 + (x - x_2)(c_3 + \cdots + (x - x_{n-1})c_n \cdots)),$$

hence can evaluate it by the following generalization of the earlier script:

```
v = zeros(size(x));
for k=length(c):-1:1
   v = (x - x(k)).*v + c(k);
end
```


In fact, if $x_1 = x_2 = \ldots = x_{n-1} = 0$, then the Newton form (5) simply reduces to the familiar power form (3).

As it turns out, there is a more useful way of computing the coefficients $c_1, \ldots, c_n$ in the Newton form for the interpolating polynomial than by (4), and that is via *divided differences*. Here are the basic facts. In the discussion, f is some given function to be interpolated, $x_1, \ldots, x_n$ is a sequence of pairwise distinct points, and $p_{i:j}$ denotes the unique polynomial of degree $\le j - i$ that interpolates to f at the points $x_i, \ldots, x_j$.

1. Since $p_{i:j}$ is the unique interpolant of degree $\le j - i$ to f at $x_i, \ldots, x_j$, its highest coefficient, i.e., the coefficient of x^{j-i}, only depends on f. We recognize this by writing this coefficient as $f[x_i, \ldots, x_j]$. In other words,

$$p_{i:j}(x) =: f[x_i, \ldots, x_j] x^{j-i} + \mathbf{l.o.t.}.$$

   Note that $f[x_i, \ldots, x_j]$ *is a symmetric function of the points* $x_i, \ldots, x_j$ since the interpolating polynomial doesn't depend on the order in which we write down the interpolation points.
2. Now note that

$$p(x) := \frac{(x - x_1) p_{2:n}(x) + (x_n - x) p_{1:n-1}(x)}{x_n - x_1}$$

   is a polynomial of degree $< n$ that interpolates to f at $x_1, \ldots, x_n$. Indeed, for $x = x_1$, the first term in the numerator is 0 and what is left reduces to $p_{1:n-1}(x_1)$, hence equals $f(x_1)$. The story for $x = x_n$ is similar. Finally, if $i = 2{:}(n-1)$, then $p_{2:n}(x_i) = p_{1:n-1}(x_i) = f(x_i)$, i.e., the numerator simplifies to $(x_n - x_1) f(x_i)$, hence $p(x_i) = f(x_i)$.
3. It follows that the p in 2. must be $p_{1:n}$. In particular, their highest coefficients must be the same. With the notation introduced in 1., this means that

$$f[x_1, \ldots, x_n] = \frac{f[x_2, \ldots, x_n] - f[x_1, \ldots, x_{n-1}]}{x_n - x_1}. \tag{6}$$

   This basic formula is, perhaps, easier to remember in the following form:

$$f[M, a, b] = \frac{f[M, a] - f[M, b]}{a - b}$$

where M stands for an arbitrary sequence of points, and a and b are two specific points.

This basic formula explains the customary name for $f[x_1, \ldots, x_n]$ as a **divided difference of order** $n - 1$, since it shows it to be a difference quotient or 'divided difference' of two divided differences of one order less. It also leads to the systematic calculation of the coefficients in the interpolating polynomial

$$p_{n-1}(x) = p_{1:n}(x) = \sum_{k=1}^{n} f[x_1, \ldots, x_k] \prod_{j<k} (x - x_j) \tag{7}$$

via the **divided difference table**, as explained in the textbook.

It is this formula for the interpolating polynomial you should commit to memory since it also gives you the standard error formula for polynomial interpolation, as follows. Suppose x_{n+1} is yet another point. Then, by the formula,

$$p_{1:n+1}(x) = p_{1:n}(x) + f[x_1, \ldots, x_{n+1}] \prod_{j=1}^{n} (x - x_j).$$

In particular, for $x = x_{n+1}$, we obtain

$$f(x_{n+1}) = p_{1:n+1}(x_{n+1}) = p_{1:n}(x_{n+1}) + f[x_1, \ldots, x_{n+1}] \prod_{j=1}^{n} (x_{n+1} - x_j),$$

and this holds for any x_{n+1} not equal to $x_1, \ldots, x_n$. We conclude that, for any x not equal to $x_1, \ldots, x_n$,

$$f(x) = p_{1:n}(x) + f[x_1, \ldots, x_n, x] \prod_{j=1}^{n} (x - x_j). \tag{8}$$

This formula you should commit to memory as well. In fact, if you only remember this formula, you can derive from it the Newton form for $p_{1:n}$. What if x here equals one of the x_j? Well, then error is zero and the product $\prod_{j=1}^{n}(x - x_j)$ also vanishes, hence, regardless of what $f[x_1, \ldots, x_n, x]$ might be in that case, the formula is still correct! It turns out that the divided difference $f[x_1, \ldots, x_k]$ makes perfectly good sense even when some or all of the x_i coincide, as the guiding example of $f[x_1, x_2]$ illustrates: For $x_1 \neq x_2$, $f[x_1, x_2]$ is the slope of the secant to f at the points x_1 and x_2; as $x_2 \to x_1$, this converges to the slope of the tangent to f at x_1, hence one sets $f[x_1, x_1] := f'(x_1)$.

Moreover, recall from Calculus the *Mean Value Theorem* which says that $f[x_1, x_2] = f'(\eta)$ for some η between x_1 and x_2. Here is a generalization of this needed before we can make use of (8).

9 Proposition. *If f has n continuous derivatives, then*

$$f[x_1, \ldots, x_{n+1}] = f^{(n)}(\eta)/n!$$

for some point η in the smallest interval containing all the points $x_1, \ldots, x_{n+1}$.

In other words, an nth divided difference is like a (normalized) nth derivative.

For the proof, assume without loss of generality that $x_1 < x_2 < \cdots < x_{n+1}$ and consider the error

$$e := f - p_{1:n+1}.$$

It vanishes at the points $x_1 < \cdots < x_{n+1}$. Hence, by Rolle's Theorem (a special case of the Mean Value Theorem) and for $i = 1{:}n$, its first derivative has a zero between x_i and x_{i+1}, call it $x_i^{(1)}$, thus getting the n zeros $x_1^{(1)} < \cdots < x_n^{(1)}$ of the first derivative e' of e. Repeating this argument with e' instead of e, then with e'' instead of e', etc., we finally arrive at the statement that $e^{(n)}$, the nth derivative of e, vanishes at some point, η say, in the interval $(x_1 \mathinner{..} x_{n+1})$. In other words,

$$0 = e^{(n)}(\eta) = f^{(n)}(\eta) - p_{1:n+1}^{(n)}(\eta).$$

But $p_{1:n+1}$ is a polynomial of degree $\le n$, hence its nth derivative is simply $n!$ times its highest coefficient, and that coefficient is, by definition, the number $f[x_1,\ldots,x_{n+1}]$. In other words,

$$0 = f^{(n)}(\eta) - n!f[x_1,\ldots,x_{n+1}],$$

which is what we set out to prove. □

Here is a quick use of this error formula. Suppose we want to make a table $(x_i, f(x_i))$, $i = 0{:}N$, with $x_i := a + ih$, all i, and $x_N = b$, hence $h = (b-a)/N$, for some given interval $[a\,..\,b]$ and some given function f. Further, we would like to have as few entries in this table as we can get away with. Specifically, we want to choose h as large as possible (or, equivalently, N as small as possible), subject to the condition that linear interpolation in this table gives an absolute error no bigger than .0005.

Let $x_i \le x \le x_{i+1}$, for some i. Then we are approximating $f(x)$ by

$$p_{i:i+1}(x) = f(x_i) + f[x_i, x_{i+1}](x - x_i).$$

Hence, the absolute error in this approximation is

$$|f(x) - p_{i:i+1}(x)| = |f[x_i, x_{i+1}, x]||x - x_i||x - x_{i+1}|.$$

Since $x_i \le x \le x_{i+1}$, we know (assuming that f has two derivatives) that

$$f[x_i, x_{i+1}, x] = f''(\eta)/2$$

for some $\eta \in [x_i\,..\,x_{i+1}]$. We can also work out that the absolutely biggest value of the parabola $(x - x_i)(x - x_{i+1})$ for $x_i \le x \le x_{i+1}$ occurs at the midpoint where the parabola has the value $(h/2)(-h/2)$. Therefore, altogether,

$$|f(x) - p_{i:i+1}(x)| \le (\max_{x_i \le \eta \le x_{i+1}} |f''(\eta)|/2)\ h^2/4.$$

Suppose now that the function is $f(x) = \sin(x)$. Then its second derivative is $-\sin(x)$, hence is bounded by 1 in absolute value. Hence our error bound simplifies to $h^2/8$, i.e., it is sufficient to choose h so that

$$h^2/8 \le .0005.$$

The largest h satisfying this condition is

$$h^* := \sqrt{8 * .0005} = \sqrt{.004} \sim .0632.$$

Suppose further that $[0\,..\,\pi/2]$ is the interval of interest. Then we want the smallest *integer* N so that $(\pi/2 - 0)/N \le h^*$, i.e., so that

$$24.83 \sim (\pi/2 - 0)/h^* \le N.$$

Hence, $N = 25$ is our best choice.