Polynomial Evaluation

(Com S 477/577 Notes)

Yan-Bin Jia

Sep 17, 2020

Polynomials are perhaps the best understood and most applied functions. The foundation comes from algebra and calculus. Taylor's expansion says that a function can be locally expanded around a point into a polynomial whose coefficients depend on the derivative and higher order derivatives of the function at the point. A main reason people have such interest in polynomials — and in polygons, polyhedra, and polytopes — is because of the following approximation theorem that has both theoretical and practical relevance.

Theorem 1 (Weierstrass Approximation Theorem) *If f is any continuous function on the finite closed interval $[a, b]$, then for every $\epsilon > 0$ there exists a polynomial $p(x)$ (whose degree and coefficients depend on ϵ) such that*

$$\max_{x \in [a,b]} \Big| f(x) - p(x) \Big| < \epsilon.$$

Evidently, the above theorem does not tell us how to construct $p(x)$, or even what the degree of $p(x)$ is. This will be addressed in the interpolation and approximation of functions.

There are many applications of polynomials which can be directly evaluated by computers:

- Optimization of polynomial objective functions subject to linear and nonlinear constraints lies in the core of operations research, a field that has impact on resource allocation, transportation, scheduling, economics, etc.
- Polynomials are used by scientists and engineers to interpolate their experimental data and model the behaviors of physical processes.
- Systems of multi-variate polynomial equations have received much attention in robotics (motion planning in particular), machine vision, etc.
- In computer graphics and geometric modeling, parametric curves and surfaces are based on polynomials to model objects in two and three dimensions.
- In computer vision, polynomials are often fit to image data to describe shape contours.

Therefore, we will devote quite a few topics in this course to polynomials: evaluation, multiplication, and root finding.

The most common form of a polynomial $p(x)$ is the *power form*:

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n, \qquad a_n \neq 0. \tag{1}$$

Here n is the *degree* of the polynomial, and $a_0, a_1, \ldots, a_n$ are its *coefficients*. Though a polynomial is a sum of powers, its efficient evaluation does not simply reduce to power evaluations and summations. Nevertheless, we first look at how to efficiently evaluate a power.

1 Evaluation of Powers

In this section we shall study the problem of computing the special polynomial x^n efficiently, given x and n, where n is a positive integer. Suppose, for example, that we need to compute x^{16} ; we could simply start with x and multiply by x fifteen times. But it is possible to obtain the same answer with only four multiplications, each a square operation. That is, we successively obtain x^2, x^4, x^8, and x^{16}.

The same idea applies, in general, to any value of n in the following way. Write n as a binary number. Replace each "1" except the one at the leading digit by the pair of letters SX, replace each "0" by S. The result is a rule for computing x^n from x, if "S" is interpreted as the operation of *squaring* and "X" is interpreted as the operation of *multiplying by* x. For example, if $n = 23$, its binary representation is 10111; so we obtain the the rule SSXSXSX after discarding the leading digit and performing the replacements described above. The rule states that we should "square, square, multiply by x, square, multiply by x, square, and multiply by x"; in other words, we should successively compute $x^2, x^4, x^5, x^{10}, x^{11}, x^{22}, x^{23}$.

This binary method is easily justified by a consideration of the sequence of exponents in the calculation. Suppose we reinterpret "S" as the operation of multiplying by 2 and "X" as the operation of adding 1, and start with 1 instead of x. The rule will then lead to a computation of n because of the properties of the binary number system.

The S-and-X binary method requires that the binary representation of n be scanned from left to right. Computer programs generally prefer to go the other way, because the available operations of division by 2 and remainder mod 2 will deduce the binary representation from right to left. Therefore the following algorithm, based on a right-to-left scan of the number, is often more convenient:

```
POWER(x, n)
1   k ← n
2   y ← 1
3   z ← x
4   while k > 0
5         m ← k
6         k ← k/2
7         if m > 2k
8            then y ← z × y
9         z ← z × z
10  return y
```

The table below illustrates the execution of this procedure in the evaluation of x^{23}. The ith row lists the values of k, y, and z at the start of the ith iteration of the **while** loop of lines 4–9. And the last row lists the variable values at the termination of the loop. The value of x^{23} is stored in y.

k	y	z
23	1	x
11	x	x^2
5	x^3	x^4
2	x^7	x^8
1	x^7	x^{16}
0	x^{23}	x^{16}

The procedure POWER maintains the invariant that $x^n = yz^k$ at the start of each iteration of the **while** loop of lines 4–9. Since k is at least halved in every iteration, it will decrease to 0 to terminate the loop. The invariant implies that $x^n = y \cdot z^0 = y$ at the termination.

The number of multiplication required by the procedure is

$$\lfloor \log n \rfloor + \nu(n),$$

where $\nu(n)$ is the number of ones in the binary representation of n. This corresponds to the number of times line 8 is executed. Because of the bookkeeping time required by this algorithm, the binary method is usually not of importance for small values of n, say $n \leq 10$, unless the time for a multiplication is comparatively large. If the value of n is known in advance (and stays the same for multiple evaluations), the left-to-right binary method is preferable.

2 Evaluation of Polynomials

There are a variety of operations we might wish to define for polynomials. But first let us look at the most basic one of all — evaluation. Suppose we would like to obtain the value of a polynomial given in the power form (1) at a point $x = t$. We rewrite it in the following nested form:

$$p(x) = a_0 + x\Big(a_1 + x\Big(a_2 + \cdots + x(a_{n-1} + xa_n)\cdots\Big)\Big).$$

Below is an iterative procedure referred to as *Horner scheme* or *nested multiplication*:

$$\begin{aligned} b_n &\longleftarrow a_n \\ b_{n-1} &\longleftarrow a_{n-1} + tb_n \\ &\vdots \\ b_i &\longleftarrow a_i + tb_{i+1} \\ &\vdots \\ b_1 &\longleftarrow a_1 + tb_2 \\ p(t) = b_0 &\longleftarrow a_0 + tb_1 \end{aligned}$$

The above evaluation involves n multiplications and n additions. Suppose an arithmetic operation is always done in constant time, that is, $\Theta(1)$. Then the evaluation takes time $\Theta(n)$ since the total number of arithmetic operations involved is on the order of n.

How do we evaluate the derivative $p'(x)$ at t? We could first obtain the derivative as

$$p'(x) = a_1 + 2a_2x + \ldots + na_nx^{n-1}.$$

and then evaluate this new polynomial of degree $n-1$ at t using Horner scheme. But there is a small efficiency trick we can play here. Indeed, the intermediate quantities $b_0, \ldots, b_n$ computed above can serve another purpose. Note from the above iterative procedure that $a_n = b_n$ and

$$a_i = b_i - b_{i+1}t, \qquad \text{for } i = 0, \ldots n-1.$$

Substituting these equations into $p(x)$ yields

$$\begin{aligned} p(x) &= a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \\ &= b_n x^n + (b_{n-1} - b_n t)x^{n-1} + \cdots + (b_1 - b_2 t)x + b_0 - b_1 t \\ &= b_0 + (x-t)b_n x^{n-1} + (x-t)b_{n-1}x^{n-2} + \cdots + (x-t)b_1 \\ &= b_0 + (x-t)q(x), \end{aligned} \tag{2}$$

where $q(x) = b_n x^{n-1} + \cdots + b_2 x + b_1$. In evaluating $p(t)$ as a number, we need to determine the coefficients $b_1, \ldots, b_n$ of a polynomial of degree $n-1$.

By differentiating (2) we get

$$p'(x) = q(x) + (x-t)q'(x).$$

In particular

$$p'(t) = q(t).$$

Because $p(x)$ is a polynomial, we have a very simple method for computing its derivative. Indeed, when evaluating $p(t)$ by Horner scheme, we can simultaneously evaluate $p'(t)$.

Let the coefficients $c_1, c_2, \ldots, c_n$ be used for evaluating $p'(t) = q(t)$. The pseudo-code for obtaining both $p(t)$ and $p'(t)$ is as follows.

$b_n \leftarrow a_n$
$c_n \leftarrow b_n$
for $k = n-1$ **downto** 1
    $b_k \leftarrow a_k + tb_{k+1}$
    $c_k \leftarrow b_k + tc_{k+1}$
$b_0 \leftarrow a_0 + tb_1$

The values $p(t)$ and $p'(t)$ will be stored in b_0 and c_1, respectively, after the execution.

References

[1] Frank Deutsch. *Best Approximation in Inner Product Spaces*. Springer-Verlag New Yaork, Inc., 2001.

[2] M. Erdmann. Lecture notes for *16-811 Mathematical Fundamentals for Robotics*. The Robotics Institute, Carnegie Mellon University, 1998.

[3] D. E. Knuth. *Seminumerical Algorithms*, *The Art of Computer Programming*, vol. 2, 3rd edition, Addison-Wesley, 1998.