Lecture `algor`: Algorithms and complexity, simple examples

August 26, 2022

Summary: Horner's method for polynomial evaluation as a simple algorithm to examine complexity in terms of flop count. Concepts of linear and quadratic complexity.

References: Chapter 1 of *Numerical Analysis* by T. Sauer.

Polynomial evaluation by Horner's method

This section first describes two ways to represent a polynomial (the first way is really a special case of the second way), and then considers the problem of evaluation.

Polynomial representations

Probably you have encountered a generic degree-n polynomial expressed as

$$p(x) = a_1 + a_2 x + a_3 x^2 + \cdots + a_{n+1} x^n,$$

where the $a_k \in \mathbb{R}$ for $k = 1, \ldots, n+1$ are real constants. Here we index the a_k starting from $k = 1$ to follow MATLAB's convention. In summation notation (arguably more precise)

$$p(x) = \sum_{k=1}^{n+1} a_k x^{k-1}. \tag{1}$$

For example, take

$$p(x) = \tfrac{15}{4} - \tfrac{9}{4} x + \tfrac{1}{2} x^2, \tag{2}$$

for which $n = 2$, $a_1 = \frac{15}{4}$, $a_2 = -\frac{9}{4}$ and $a_3 = \frac{1}{2}$. The representation (1) relies on the *monomial basis*, and we'll encounter it again when studying *polynomial interpolation* later in the course.

Another, more general, polynomial representation involves shifts of the independent variable x:

$$p(x) = c_1 + c_2(x - r_1) + c_3(x - r_1)(x - r_2) + \cdots + c_{n+1}(x - r_1)(x - r_2)\cdots(x - r_n),$$

again for real numbers $c_k \in \mathbb{R}$, $k = 1, \cdots, n+1$. In summation notation

$$p(x) = \sum_{k=1}^{n+1} c_k \prod_{j=1}^{k-1} (x - r_j), \tag{3}$$

where by convention the empty product is $\prod_{j=1}^{0}(x - r_j) = 1$. Here is an example:

$$p(x) = 2 - \tfrac{1}{4}(x - 1) + \tfrac{1}{2}(x - 1)(x - 3). \tag{4}$$

This has the shifted form with $n = 2$, $c_1 = 2$, $c_2 = -\frac{1}{4}$, $c_3 = \frac{1}{2}$, $r_1 = 1$, and $r_2 = 3$. It's important to realize that the same polynomial can have different representations. Indeed, simple arithmetic shows that (2) and (4) are the same polynomial. Note also that the monomial form is really just a special case of the shifted form, one for which all *base points* $r_k = 0$. These observations show that a shifted representation for a polynomial is not unique. We can always find another relative to a different set of base points.

Evaluation at a point x

Consider the problem of evaluating a polynomial p at the input value x, that is computation of the output value $y = p(x)$. We seek an efficient evaluation of the polynomial expressed in shifted form. To see that there are some issues at play, let's consider first the monomial representation

$$p(x) = 3.99x^4 + 7.34x^3 - 2.03x^2 + 9.1x - 5.277. \tag{5}$$

A straightforward approach to compute $p(x)$ in MATLAB goes as follows:

```
p = 3.99*x^4 + 7.34*x^3 - 2.03*x^2 + 9.1*x - 5.277,
```

which is equivalent to

```
p = 3.99*x*x*x*x + 7.34*x*x*x - 2.03*x*x + 9.1*x - 5.277,
```

This involves 10 multiplications and 4 additions (we view a subtractions as the addition of a negative number), that is 14 floating point operations. A *floating point operation* or *flop* is a single arithmetical operation such as a multiplication, addition, subtraction, or division (technically as performed on a computer; more on this later). The number of flops (operations) needed to perform a given algorithm is a measure of how much work a computer (or a person!) must expend in carrying it out.

Can we do better? One simple way involves recursion:

```
x2 = x*x;
x3 = x*x2;
x4 = x*x3;
p = 3.99*x4 + 7.34*x3 - 2.03*x2 + 9.1*x - 5.277,
```

Now we have 7 multiplications and 4 additions, that is 11 flops. Nonetheless, we can do even better by exploiting the *nested expansion*

```
p = -5.277 + x*(9.1 + x*(-2.03 + x*(7.34 + x*3.99)))
```

Starting first with the computation `x*3.99` and working outward, we now perform only 4 multiplications and 4 additions, that is 8 flops. One point of these observations is that how we organize an algorithm affects the operation count.

Horner's method (according to Wikipedia due to William George Horner in 1819) exploits a nested expansion for any shifted representation. For example, (4) has the nested expansion

$$p(x) = 2 + (x-1)(-\tfrac{1}{4} + (x-3)\tfrac{1}{2}) = c_1 + (x - r_1)(c_2 + (x-3)c_3).$$

So to evaluate $y = p(x)$ in MATLAB, we might type the following.

```
y=c3
y=y*(x-3) + c2
y=y*(x-1) + c1
```

This hints at the general algorithm shown below in `horner.m`. This function has a few bells and whistles. First, if called with only 3 arguments, then it chooses all based points as zero. It also uses `.*`, so that the function can be called on an arbitrarily structured arrays, e.g. on a vector or rectangular array of evaluation points. Here is a demonstration of its use in OCTAVE to evaluate (4).

```
octave:1> n = 2; c=[2 -1/4 1/2]; r = [1 3]; x = [3.1 3.2 3.3];
octave:2> y=horner(n,c,x,r)
y =
   1.5800   1.6700   1.7700
```

```
% Evaluates polynomial from nested form using Horner's Method
% Input:   n, polynomial degree (passed, but could be discerned from length of c)
%          c, array of n+1 coefficients, where c(1) fixes constant term
%          x, coordinate at which to evaluate
%          r, array of d base points, if needed
% Output:  y, value of polynomial at x
% Example: y = c1 + c2(x-r1) + c3(x-r1)(x-r2) + c4(x-r1)(x-r2)(x-r3)
% Warning: No error checking. Passing an n < length(c)-1 will yield wrong answer.
% function y=horner(n,c,x,r); if nargin < 4, r=zeros(n,1); end
  function y=horner(n,c,x,r); if nargin < 4, r=zeros(n,1); end
y=c(n+1);
for i = n:-1:1
    y = y.*(x-r(i))+c(i);
end
```

If an algorithm requires many flops, then it is expensive. The more expensive an algorithm, the longer it takes to execute on a computer. Therefore, we are often quite interested in estimating the operation count (or complexity) of a given algorithm. Estimation is often necessary, as in practice we can not and need not know the exact count. How many flops does Horner's method require? Studying the algorithm, we see that all flops occur within the `for` loop. For each instance of i, there are clearly 3 flops if x is a scalar, and $3m$ flops if x is a vector of length m. The number of instances of i is n, so the total flop count is $3n$ for a scalar x and $3mn$ for a vector x. *So long as we agree to fix and not change the size of* x, the cost of the algorithm is then $O(n)$. Although there is a precise definition of the order symbol O, here you may take $O(n)$ to mean that

$$\lim_{n\to\infty} \frac{\text{exact number of operations needed to carry out algorithm}}{n} = K,$$

where K is a constant. The idea here is that the work amount is about Kn for large enough n. This is *linear complexity.* A side note: you might wonder why $3n = 12$ for $n = 4$, whereas we saw above that the nested evaluation of (5) required only 8 flops. The reason is that all the r_k shifts are zero in (5). If we know in advance that all shifts are zero, then the algorithm for scalar x requires $2n$ flops.

Another example

Let's consider an other example, a simple one that has quadratic complexity: matrix-vector multiplication. Let A be an $m \times n$ matrix and $\mathbf{v}$ a length-n vector, and consider the computation of $\mathbf{w} = A\mathbf{v}$. Each component of the length-m vector $\mathbf{w}$ involves multiplication of a row of A into $\mathbf{v}$. Indeed,

$$w_j = \sum_{k=1}^{n} a_{jk}v_k = a_{j1}v_1 + a_{j2}v_2 + \cdots + a_{jn}v_n.$$

The cost to compute w_j is then n multiplications and $n-1$ additions, so $2n-1$ flops. Since the vector $\mathbf{w}$ has m-components, the total cost to compute $\mathbf{w}$ is $2mn - m$ flops.

Suppose now that we restrict to the case that A is a square matrix. Then the total cost is $2n^2 - n$, that is an $O(n^2)$ cost. This is *quadratic complexity.* Indeed, suppose, as is the case now, that

$$\lim_{n\to\infty} \frac{\text{exact number of operations needed to carry out algorithm}}{n^2} = K,$$

where K is a constant. Then for large enough n the cost is about Kn^2. If an $n = 100$ product requires about a $10,000K$ work amount, then an $n = 500$ product will require a $250,000K$ work amount. Therefore, computing an $n = 500$ matrix-vector product should take about 25 times longer than computing an $n = 100$ matrix-vector product.