APPENDIX 6

Evaluating polynomials: getting along without variables

polynomials:1 Being able to evaluate arbitrary polynomials is very useful. We want a procedure with two arguments, the first a number x and the second an array $[a_0\ a_1\ \ldots\ a_{n-1}]$ to be interpreted as the coefficients of a polynomial. The procedure should return $P(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$. In choosing this order for the arguments, I am following the usual rule of PostScript with an argument x first and then the object to be applied to it (the polynomial). The point is that this choice makes composition easy.

In using polynomial evaluation in some tools, such as `mkpath`, the derivatives of P are also needed. The method used to evaluate $P(x)$ can evaluate $P'(x)$ with little extra effort.

In many applications, a polynomial has to be evaluated many times, and it is therefore important to design the evaluation procedure to be efficient. This will offer an excuse to include a few remarks about managing the stack without variable names.

1. The most straightforward way to do it

Here is a simple procedure that will evaluate an arbitrary polynomial $a_3x^3 + a_2x^2 + a_1x + a_0$ of degree three.

```
% arguments:  number x and array a = [ a0 a1 a2 a3 ]
cubic-poly { 2 dict begin
  /a exch def
  /x exch def
  a 0 get
  a 1 get x mul add
  a 2 get x 2 exp mul add
  a 3 get x 3 exp mul add
end } def
```

Exercise 1. *Extend this procedure, using a* `for` *loop, so that it will evaluate a polynomial of arbitrary degree. Be careful that your procedure works even for a polynomial of degree* 0 *(a constant). Also, it should return* 0 *if the array is empty. (Note that the degree is one less than the length of the coefficient array.)*

Exercise 2. *Extend in turn the procedure from the previous exercise so that it will return the array of two numbers* $[P(x)\ P'(x)]$.

2. Horner's method

thod of evaluation:1 The PostScript command `exp` is somewhat slow, and the straightforward procedure used above is therefore probably inefficient. Better is an elegant method of evaluating polynomials due to the nineteenth century English
Horner, W. G.:1 mathematician W. G. Horner. It does not use `exp`, but gets by with just successive multiplications and additions.

We start off by rewriting a cubic polynomial:

$$P(x) = a_3x^3 + a_2x^2 + a_1x + a_0 = (((a_3)x + a_2)x + a_1)x + a_0\ .$$

In other words, to evaluate the polynomial it suffices to calculate in succession

$$\begin{aligned}&a_3\\&a_3x + a_2\\&(a_3x + a_2)x + a_1\\&(((a_3)x + a_2)x + a_1)x + a_0\end{aligned}$$

or, in other words, giving these expressions labels, we calculate

$$\begin{aligned}b_2 &= a_3\\b_1 &= b_2x + a_2\\b_0 &= b_1x + a_1\\b_{-1} &= b_0x + a_0\ .\end{aligned}$$

At the end $b_{-1} = P(x)$. In PostScript this becomes

```
/b a 3 get def
/b b x mul a 2 get add def
/b b x mul a 1 get add def
/b b x mul a 0 get add def
```

and at the end $b = P(x)$. We can even get by without definitions.

```
a 3 get
x mul a 2 get add
x mul a 1 get add
x mul a 0 get add
```

will leave $P(x)$ on the stack. There is the germ of a simple loop here.

Exercise 3. *Write a procedure which evaluates an arbitrary fourth degree polynomial $P(x)$ using a loop, and without defining b.*

It would be easy enough to construct a PostScript procedure that implements Horner's algorithm using variables x and a, but it a bit more interesting to construct one that does all its work on the stack. The point of this is that accessing the value of a variable is an operation costly in time. Besides, it's often an enjoyable exercise.

tack:manipulation:2

I haven't said much about what's involved in sophisticated stack management. The most important thing to keep in mind is this:

- *If you are going to get along without variable names, then the stack has to hold at every moment the entire state of the computation.*

In Horner's method, the state of the computation involves the value of x, the current value of the polynomial, and a specification of the coefficients yet to be used. In the following code, this is encapsulated in the list of unused coefficients a_i, the current polynomial value P, and the variable x, sitting on the stack bottom to top in that order, inside a repeat loop.

```
  % x [ a0 a1 ...  ]
  /horner {
  aload length             % x a0 a1 ...  an n+1
  dup 2 add -1 roll        % a0 a1 ...  an n+1 x
  exch 1 sub {             % a0 a1 ...  P=an x
    dup 4 1 roll           % a0 ...  x ak P x
    mul add exch           % a0 ...  a[k-1] P x
  } repeat
  % at end P x on stack
  pop                     % P
} def
```

3. Evaluating the derivatives efficiently

Very often in plotting a graph it is useful to obtain the value of $P'(x)$ at the same time as $P(x)$. Horner's method allows this to be done with little extra work.

The formulas for the b_i above can be rewritten as

$$\begin{aligned} a_3 &= b_2 \\ a_2 &= b_1 - b_2 x \\ a_1 &= b_0 - b_1 x \\ a_0 &= b_{-1} - b_0 x \end{aligned}$$

We can therefore write

$$\begin{aligned} P(X) &= a_3 X^3 + a_2 X^2 + a_1 X + a_0 \\ &= b_2 X^3 + (b_1 - b_2 x) X^2 + (b_0 - b_1 x) X + (b_{-1} - b_0 x) \\ &= b_2 (X^3 - X^2 x) + b_1 (X^2 - X x) + b_0 (X - x) \\ &= (b_2 X^2 + b_1 X + b_0)(X - x) + b_{-1} \\ &= (b_2 X^2 + b_1 X + b_0)(X - x) + P(x)\ . \end{aligned}$$

Therefore the coefficients b_i have significance in themselves; they are the coefficients of a simple polynomial:

$$b_2 X^2 + b_1 X + b_0 = \frac{P(X) - P(x)}{X - x} = \text{(say) } P_1(X)\ .$$

One remarkable consequence of this is that we can evaluate $P'(x)$ easily since a simple limit argument gives

$$P'(x) = b_2 x^2 + b_1 x + b_0$$

which means that we can apply Horner's method to the polynomial

$$P_1(X) = b_2 X^2 + b_1 X + b_0$$

in turn to find it. We can in fact evaluate $P(x)$ and $P'(x)$ more or less simultaneously, if we think a bit about it. Let the numbers c_i be calculated from $P_1(X)$ in the same way that the b's came from $P(X) = P_0$.

$$\begin{aligned} b_2 &= a_3 \\ b_1 &= b_2 x + a_2 \\ b_0 &= b_1 x + a_1 \\ b_{-1} &= b_0 x + a_0 \\ c_1 &= b_2 \\ c_0 &= c_1 x + b_1 \\ c_{-1} &= c_0 x + b_0 \end{aligned}$$

concluding with $P(x) = b_{-1}$, $P'(x) = c_{-1}$. This requires that we store the values of b_i to be used in calculating the c's. In fact we can avoid this by interlacing the calculations:

$$\begin{aligned} b_2 &= a_3 \\ c_1 &= b_2 \\ b_1 &= b_2 x + a_2 \\ c_0 &= c_1 x + b_1 \\ b_0 &= b_1 x + a_1 \\ c_{-1} &= c_0 x + b_0 \\ b_{-1} &= b_0 x + a_0 \end{aligned}$$

concluding with $P(x) = b_{-1}$, $P'(x) = c_{-1}$.

Exercise 4. *Design a procedure with two arguments, an array and a number x, which returns $[P(x)\ \ P'(x)]$, where the polynomial P is defined by the array argument. It should use Horner's method to do this, preferably in the most efficient version. (Hint: evaluatng the derivative requires one less step than evaluating the polynomial. This can be dealt with by initializing c to 0 in the steps shown above. In this way, c and b can be handled in the same number of steps, and writing the loop becomes simpler.)*

4. Evaluating Bernstein polynomials

nialsaBéBernsteib:4 The Bernstein polynomials are the generalizations of the Bézier cubic polynomials, polynomials of the form

$$B_y(t) = (1-t)^n y_0 + n(1-t)^{n-1} t y_1 + \frac{n(n-1)}{2}(1-t)^{n-2}t^2 y_2 + \cdots + t^n y_n \ .$$

They are used frequently in computer graphics, for reasons explained elsewhere. A procedure to evaluate one has two arguments, the number t and the array of the y_i, and returns $B_y(t)$. The basic principle here will be as in Horner's algorithm, except that here the polynomial coefficients must be calculated as we go along. Fix n and let

$$C_k = \frac{n(n-1)\ldots(n-(k-1))}{k!}\, t^k$$

so that

$$B_y(t) = \sum_{k=0}^{n} C_{n-k} y_{n-k} s^k \quad (s = 1-t) \ .$$

In evaluating $B_y(t)$ by Horner's method, the coefficients C_k must be evaluated on the fly. This is done by the inductive process

$$\begin{aligned} C_0 &= 1 \\ C_{k+1} &= C_k \cdot t \cdot \frac{n-k}{k+1} \ . \end{aligned}$$

We run Horner's algorithm with variables P (the current value of the polynomial), k (an index), and C (equal to C_k at all times). We start with $P = y_0$, $k = 1$, and $C = C_1 = nt$. Then the appropriate variant of Horner's algorithm repeats n times, setting in each loop

$$\begin{aligned} P &:= P \cdot s + C \cdot y_k \\ C &:= C \cdot \frac{n-k}{k+1} \cdot t \\ k &:= k+1 \end{aligned}$$

and end with $P = B_y(t)$. It seems plausible that this procedure will be evaluated often, so efficiency is a major concern. For this reason, in the following code, no dictionary is used. This is one of the more complicated code segments in which I do this—note the heavy use of `index` to retrieve the values of y, n, s, and t. This is especially reasonable since they remain constant throughout the calculation once stored. (There is no storage operator to match the retrieval operator `index`.)

```
% t y=[ y0 y1 ...  yn ]
/bernstein {  % t y
  % constants y n t s=1-t
  % variables k C P
  dup length          % t y n+1
  1 sub               % t y n
  3 -1 roll 1         % y n t 1
  1 index sub         % y n t s
```

```
      % constants in place
     1                   % y n t s k
     3 index 3 index mul  % y n t s k C=nt
     5 index 0 get        % y n t s k C P=y0
     5 index {            % y n t s k C P
       % P -> P* = s.P + C.y[k]
       % C -> C* = C.t.(n-k)/(k+1)
       % k -> k* = k+1
       3 index mul        % y n t s k C P.s
       1 index            % y n t s k C P.s C
       7 index            % y n t s k C P.s C y
       4 index get mul add   % y n t s k C P.s+C.y[k]=new P
       3 1 roll           % y n t s P* k C
       5 index            % y n t s P* k C n
       2 index sub mul    % y n t s P* k C.(n-k)
       1 index 1 add div  % y n t s P* k C.(n-k)/(k+1)
       4 index mul        % y n t s P* k C*
       3 1 roll 1 add     % y n t s C* P* k*
       3 1 roll           % y n t s k* C* P*
     } repeat
     7 1 roll 6 { pop } repeat
} def
```

5. Code

horner:5 The file `horner.inc` has one procedure `horner` with arguments x and a that returns $A(x)$. The file `bernstein.inc`
bernstein:5 has a procedure `bernstein` with arguments t and y that returns $B_y(t)$.