Horner's Method for Evaluating and Deflating Polynomials

C. Sidney Burrus, James W. Fox, Gary A. Sitton, and Sven Treitel

Rice University

November 26, 2003

Abstract

Horner's method is a standard minimum arithmetic method for evaluating and deflating polynomials. It can also efficiently evaluate various order derivatives of a polynomial, therefore is often used as part of Newton's method. This note tries to develop the various techniques called Horner's method, nested evaluation, and synthetic division in a common framework using a recursive structure and difference equations. There is a similarity to Goertzel's algorithm for the DFT, Z-transform inversion by division, and Padé's and Prony's methods. This approach also allows a straight forward explanation of "stability" or numerical errors of the algorithms. Matlab implementations are given.

1 Polynomials

Polynomials are one of the oldest classes of mathematical functions with an important history in mathematics, science, engineering, and other quantitative fields [1]. Part of the polynomial's appeal comes from the fact that it may be numerically evaluated using a finite number of multiplications and additions. In the processes of factoring and deflating polynomials, Horner's methods are central and are the focus of this note.

Any N^{th} degree polynomial

$$f_N[\boldsymbol{a}, z] = a_0 + a_1 z + a_2 z^2 + \cdots + a_N z^N \tag{1}$$

can be written in a *nested* form as:

$$f_N[\boldsymbol{a}, z] = a_0 + z\ (a_1 + z\ (a_2 + \cdots + z\ (a_{N-1} + z\ (a_N))) \cdots), \tag{2}$$

or nested in a different order as:

$$f_N[\boldsymbol{a}, z] = z\ (\cdots z\ (z\ (z\ (a_N) + a_{N-1}) + a_{N-2}) + \cdots) + a_0, \tag{3}$$

and in a *factored* form as:

$$f_N[\boldsymbol{a}, z] = a_N \prod_{r=1}^{N} (z - z_r) = a_N \prod_{d=1}^{D} (z - z_d)^{Q_d}, \tag{4}$$

where Q_d is the multiplicity of the d^{th} zero, D is the number of distinct zeros, and $\sum_d Q_d = N$. The polynomial can be written in a first order *remainder* form as:

$$f_N[\boldsymbol{a}, z] = q_{N-1}[\boldsymbol{b}, z]\ (z - z_0) + R, \tag{5}$$

where

$$q_{N-1}[\boldsymbol{b}, z] = b_0 + b_1 z + b_2 z^2 + \cdots + b_{N-1} z^{N-1} \tag{6}$$

is the quotient polynomial obtained when $f_N[\boldsymbol{a}, z]$ is divided by $(z - z_0)$ and the remainder is a constant easily seen to be the value of

$$R = f_N[\boldsymbol{a}, z_0]. \tag{7}$$

If z_0 is a zero of $f_N(z)$, then $R = 0$ and $q_{N-1}(z)$ contains the same zeros as $f_N(z)$ except for the one at z_0. A more general formulation of (5) is:

$$f_N[\boldsymbol{a}, z] = q_{N-M}[\boldsymbol{b}, z]\ d_M[\boldsymbol{c}, z] + R(z). \tag{8}$$

There are four polynomial problems[2] that are often posed:

1. Factor: Given the coefficients a_k, find the roots or zeros z_r.
   Matlab command: `roots()`

2. Unfactor: Given the zeros z_r, find the coefficients a_k.
   Matlab command: `poly()`

3. Evaluate: Given the coefficients or zeros and z_0, find the value of the polynomial at $z = z_0$, $f(z_0)$ (or the value of the derivative of the polynomial at $z = z_0$, $f'(z_0)$).
   Matlab command: `polyval()`

4. Deflate: Given a polynomial and one of its roots, find the polynomial containing the other roots.

Horner's methods are important for evaluation and deflation, therefore, for factoring. For many high degree polynomial factoring schemes[2], it is important to use stable evaluation and deflation and to deflate in an order that maximizes the conditioning of the quotient. Unfactoring is simply the multiplying of the factors to obtain the polynomial but, because of round-off error, the order of multiplication is important. As an alternative, the FFT/DFT can be used to unfactor and evaluate. Of the four, only factoring is nonlinear and it is the most difficult [2]. In many strategies, factoring uses the other three. For implementation of any of the four, the number of arithmetic operations required will determine the speed of execution and stability and conditioning will affect the error characteristics.

2 Evaluate the Polynomial and its Derivatives

From the structure of the nested forms in (2) and (3), one can formulate a recursive relation which is also a linear first order difference equation:

$$x_{k+1} = z\ x_k\ +\ a_{N-1-k}, \quad x_0 = a_N\ , \tag{9}$$

for $k = 0, 1, \ldots, N-1$ with the value of the polynomial at z given by $f_N[\boldsymbol{a}, z] = x_N$ [3, 4, 5]. Here the x_k are the values of the successively evaluated bracketed expressions in (2). Matlab code to evaluate $f_N[\boldsymbol{a}, z_0]$ with coefficients $\boldsymbol{a}$ is given by:

```
m = length(a);          % poly degree plus one: N+1
f = a(1);               % initial condition
for k = 2:m             % iterative Horner's algorithm
  f = z*f + a(k);       % recursive evaluation of f(z)
end
```

Program 1. Forward Evaluation of Polynomial

Remember that Matlab stores polynomial coefficients in the reverse order of the notation used by many writers and the reverse of that shown in (1) and starts indexing with 1 rather than 0.

This formulation is very efficient if implemented with the "multiply-accumulate" single-cycle instruction of several DSP chips and some microprocessors.

The nested forms of (2) and (3), the remainder form of (5), and the recursive form of (9) are all versions of Horner's method [2, 5, 6] or synthetic division. There are also similarities to the Goertzel algorithm to evaluate the DFT, the Padé or Prony method and

inversion of z-transforms by division. Indeed, all of these methods convert an evaluation into a recursion and then into a difference equation which has considerable literature [7]. While in some cases it is possible to reduce the number of multiplications used by Horner in polynomial evaluation, it is generally a small gain and accompanied by an increase in the number of additions and the complexity of the algorithm . If preprocessing of the polynomial coefficients is not used, Horner gives a minimum arithmetic evaluation [3, 5].

If we differentiate (5), we get:

$$f'_N[\boldsymbol{a}, z] = q_{N-1}[\boldsymbol{b}, z] + q'_{N-1}[\boldsymbol{b}, z]\ (z - z_0), \tag{10}$$

which, if evaluated at $z = z_0$, gives:

$$f'_N(\boldsymbol{a}, z_0) = q_{N-1}[\boldsymbol{b}, z_0]. \tag{11}$$

which can be evaluated by a minor variation of (9). Thus Horner's method can evaluate an N^{th} degree polynomial with N multiplications and additions or evaluate it and its derivative with $2N$ multiplications and additions. Note $q_{N-1}[\boldsymbol{b}, z]$ is not the derivative, but is the value of the derivative at z_0. The simple Matlab code for this is:

```
m = length(a);          % N+1
f = a(1); fp = 0;       % initial conditions
for i = 2:m             % iterative Horner's algorithm
    fp = z*fp + f;      % recursive evaluation of f'(z)
    f = z*f + a(i);     % recursive evaluation of f(z)
end
```

Program 2. Forward Evaluation of Polynomial and Its Derivative

This can be made much faster by using the `filter()` function in Matlab as:

```
N = length(a) - 1;
b = filter(1,[1 -z], a);       % Horner by filter: f(z)
f = b(N+1);
c = filter(1,[1 -z], b(1:N)); % Horner by filter: f'(z)
fp= c(N);
```

Program 3. Forward Evaluation of Polynomial and Its Derivative by Filters

From (11) and (6), we see that the value of the derivative of the N^{th} degree polynomial $f_N[\boldsymbol{a}, z]$ is the value of the $(N-1)^{th}$ degree polynomial $q_{N-1}[\boldsymbol{b}, z]$ with coefficients b_k as solutions to the difference equation [7]:

$$b_{k+1} = z\ b_k + a_{N-1-k} \tag{12}$$

which has the same form as (9) but saving the intermediate values of b_k. This means that the solution to the difference equation (12) with the N input values of a_k gives $N-1$ output values of b_k followed by the remainder R_1 which is the value of $f_N[\boldsymbol{a}, z]$.

A similar argument shows that solving (12) with an input of b_k will give $N-2$ output values of c_k followed by R_2 which is the value of the derivative of $f_N[\boldsymbol{a}, z]$. Using the c_k as an input will give $N-3$ values of d_k and R_3 which is the second derivative. This can be continued N time to evaluate the function and all of its $N-1$ derivatives (see Fig 1). These values of R_k allow the original polynomial to be written as:

$$f_N[\boldsymbol{a}, z] = R_1 + R_2(z - z_0) + R_3(z - z_0)^2 + \cdots + R_{N+1}(z - z_0)^N \tag{13}$$

which is a power series expansion [5] around $z = z_0$ with the coefficients R_k being defined in terms of the derivatives evaluated at $z = z_0$.

The following program evaluates the function and its first and second derivatives in a very compact single loop solution of the basic first order difference equation without saving the coefficients b_k or c_k.

```
m = length(a);          % poly degree + 1
f = 0; fp = 0; fpp=0;   % initial values
for i = 1:m-2           % Horner's algo.
    f = z*f + a(i);     % recursive evaluation of f(z)
    fp = z*fp + f;      % recursive evaluation of f'(z)
    fpp = z*fpp + fp;   % recursive evaluation of f''(z)
end
f = z*f + a(m-1);
fp= z*fp + f;
f = z*f + a(m);         % Finish evaluations
fpp = 2*fpp;
```

Program 4. Forward Evaluation of Polynomial and Two Derivatives
Higher order derivative can be evaluated by a simple extension of this idea.

Describing the algorithm as a flow-graph illustrates the recursion as a feed back path with the feed back parameter being z and shows the structure of evaluating derivatives.

Figure 1. Flowgraph for Horner's Algorithm

3 Polynomial Deflation

If z_0 is a zero of the polynomial $f_N[\boldsymbol{a}, z]$, then $R = 0$ and (5) becomes

$$f_N[\boldsymbol{a}, z] = q_{N-1}[\boldsymbol{b}, z]\ (z - z_0) \tag{14}$$

with q_{N-1} being the $N-1$ degree polynomial (6) having the same zeros at f_N except for the one at $z = z_0$. In other words, Horner's method can also be used to deflate a polynomial with a known zero. A Matlab program that will deflate $f_N[\boldsymbol{a}, z]$ is given by:

```
m = length(a);              % order + one: N+1
b = zeros(1,m-1);
b(1) = a(1);                % initial condition
for k = 1:m-2               % iterative Horner's algorithm
  b(k+1) = z*b(k) + a(k+1); % recursive deflation of f(z)
end
```

Program 5. Forward Deflation of Polynomial

or using the Matlab "filter" command:

```
N = length(a)-1
b = filter(1,[1 -z], a);    % Deflation by filter
b = b(1:N);
```

Program 6. Forward Deflation of Polynomial using Filter

Because multiplication of two polynomials is the same operation as the convolution of their coefficients, one can get the same results by multiplying the discrete Fourier transform (DFT)'s of the polynomial coefficients and taking the inverse DFT of the product. This gives an alternative to Horner's algorithm for deflating polynomials and is the technique use in the Lindsey-Fox polynomial factoring scheme [8, 2]. The use of the FFT for deflation or unfactoring of polynomials has very different error characteristics from Horner's method and is often superior. A Matlab program that deflates a polynomial with coefficients a_k by a divisor with coefficients d_k to give a quotient with coefficients q_k using the FFT is:

```
m = length(a); md = length(d);
q = ifft(fft(a)./fft([d,zeros(1,(m-md))]));
```

Program 7. Deflation of Polynomial using the FFT

A more general formulation of the deflation problem is of the form:

$$f_N[\boldsymbol{a}, z] = q_{N-M}[\boldsymbol{b}, z]\ d_M[c, z] + R(z) \tag{15}$$

If the factors of $d_M(z)$ are all roots of $f_N(z)$, then q_{N-M} contains the others when $R(z) = 0$. This allows the easy deflation by quadratic factors or other factor that are already known.

4 Stability

Horner's method is implemented by the recursive equation (9) or (12) which, in this form, is seen to be a linear first order constant coefficient difference equation. There is a considerable literature on the stability of linear differential and difference equations [7], and an equation of this form is known to be stable[9] for $|z| < 1$ and unstable for $|z| > 1$.

To reduce error accumulation when $|z| > 1$ we nest (1) and (2) in still a different way:

$$f_N[\boldsymbol{a}, z] = z^N[z^{-1}(\cdots z^{-1}(z^{-1}(z^{-1}(a_0) + a_1) + \cdots) + a_{N-1}) + a_N], \tag{16}$$

to give an alternate recursive relationship to (9) which is the following difference equation:

$$x_{k+1} = z^{-1}\ x_k\ +\ a_{k+1}, \quad x_0 = a_0\ , \tag{17}$$

for $k = 0, 1, \ldots, N-1$ with the value of the polynomial at z given by $f_N[\boldsymbol{a}, z] = z^N x_N$. This equation is stable [7, 9] for $|z| > 1$.

Again, remembering the Matlab index convention, the program for this form of Horner's algorithm is:

```
m = length(a);              % N+1
f = a(m);                   % initial condition
for k = 1:m-1               % iterative Horner's algorithm
  f = f/z + a(m-k);         % recursive evaluation of f(z)
end
f = (z^(m-1))*f;            % remove the factor of z^{-m+1}
```

Program 8. Backward Evaluation of Polynomial

which is stable for $|z| > 1$. The same can be done for deflation.

This algorithm is equivalent to reversing the order ("flipping") the polynomial coefficients and applying (9) which will give zeros that are the reciprocal of those of the original polynomial [6, 2]. Using the standard Matlab commands, this can be done by:

```
m = length(a)
f = (z^(m-1))*polyval(a(m:-1:1),1/z);
```

Program 9. Backward Evaluation of Polynomial by using Reversed Order Coefficients

It is easy to see that stability is very important when evaluating or deflating high degree polynomials by considering the effects if the degree is as high as one million[2]. Using this difference equation formulation of Horner's methods allows a more general deflation using a quadratic or higher degree divisor polynomial based on (8) and (15) to be easily analysed for stability.

5 Conclusions

We have seen that polynomial evaluation and deflation can be done by Horner's approach which is the conversion of the problem into a linear difference equation. Indeed, evaluation and first order deflation are accomplished by the same algorithm and the stability is easily controlled.

Having forwards and backwards algorithms allows one to always use a stable evaluation or deflation scheme [10, 11, 5, 12, 6, 2, 9]. If the magnitude of z where the polynomial is being evaluated (or where the polynomial is being deflated) is less than one, use (9), if it is greater than one, use (17).

References

[1] V. Y. Pan. Solving a polynomial equation: some history and recent progress. *SIAM Review*, 39(2):187–220, June 1997.

[2] Gary A. Sitton, C. Sidney Burrus, James W. Fox, and Sven Treitel. Factoring very high degree polynomials. *IEEE Signal Processing Magazine*, 20, November 2003.

[3] Donald E. Knuth. *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*. Addison-Wesley, Reading, MA, third edition, 1997.

[4] E. J. Barbeau. *Polynomials*. Springer-Verlag, New York, 1989.

[5] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 1996. Chapter 5 on polynomials.

[6] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in Fortran: The Art of Scientific Computing*. Cambridge University Press, Cambridge, second edition, 1992.

[7] Saber N. Elaydi. *An Introduction to Difference Equations*. Springer-Verlag, New York, second edition, 1999. First edition in 1996.

[8] J. P. Lindsey and James W. Fox. A method of factoring long z-transform polynomials. *Computational Methods in Geosciences, SIAM*, 78–90, 1992. Reprinted in Seismic Source Signature Estimation and Measurement, edited by Osman Osman and Enders Robinson, Society of Exploration Geophysicists, Geophysics Reprint Series no. 18, 712–724, 1996.

[9] James W. Fox. Improving the accuracy of polynomial evaluation, deflation, and root finding. *draft manuscript*, January 14 2002.

[10] Daniela Calvetti and Lothar Reichel. On the evaluation of polynomial coefficients. *Numerical Algorithms*, 33:153–161, 2003.

[11] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1963. Now published by Dover, 1994.

[12] Forman S. Acton. *Numerical Methods that Work*. The Mathematical Association of America, Washington, DC, 1990.