A note on synthetic division and zero of polynomials

Luca Formaggia

AY 2021-22

1 Synthetic division and Horner algorithm

Let

$$p_m(x) = \sum_{i=0}^{m} a_i x^i,$$

be a (real or complex) polynomial of degree at most m, express in terms of the monomial basis. The *Synthetic Division* or *Horner's algoritm* reads

Algorithm 1. *Computes $p(z)$*

```
auto horner(a,z)
     b[m-1]=a[m];
     for k=m-2,...0  b[k]+=b[k+1]*z + a[k+1];
     return {b[0]*z + a[0],b};
```

This version of the algorithm, beside computing $p_m(z)$, returns the coefficients b_j, $j = 0, \dots m-1$ of the so called *associated polynomial*

$$q_{m-1}(x; z) = \sum_{j=0}^{m-1} b_j x^j. \tag{1}$$

It is a polynomial of degree at most m, and the z in $q_{m-1}(x; z)$ indicates that the coefficients b_j are the results of the synthetic division applied in $x = z$ (see Algorithm 1). Indeed, the value of the coefficients b_j depends on the chosen point z.
We have the following *important result*: the polynomial $q_{m-1}(x; z)$ is the quotient of the division of p_m by $x - z$. Which means that

$$\boxed{p_m(x) = q_{m-1}(x; z)(x - z) + p_m(z),} \tag{2}$$

and, consequently,

$$\boxed{\frac{d}{dx}p_m(x) = \frac{d}{dx}q_{m-1}(x; z)(x - z) + q_{m-1}(x; z),} \tag{3}$$

by which,

$$\boxed{\frac{d}{dx}p_m(z) = q_{m-1}(z; z).} \tag{4}$$

Therefore, the synthetic division is a tool to compute not only the value at a point z, but also the derivative at that point, and the quotient of the division with the factor $x - z$. It provides the basic ingredients of the **Newton-Horner** algorithm for computing the zeros of a polynomial. But first we need to illustrate the deflation.

1.1 Deflation

Let $\{\alpha_j,\ j = 1, \dots, n\}$ be a zeroes of the polynomial p_n. Then, $\{\alpha_j,\ j = 1, \dots, n-1\}$ are the zeroes of the polynomial

$$p_{n-1} = p_n/(x - \alpha_n). \tag{5}$$

The proof is trivial. But $p_{n-1}(x) = q_{m-1}(x; \alpha_n)$, according to (2), being $p_n(\alpha_n) = 0$! This allows to set a very efficient algorithm for finding all zeroes of a polynomial.

1.2 Newton-Horner with deflation

The algorithm here implemented is Newton-Horner with deflation:

Algorithm 2. Given the polynomial p_n, the initial point x_0 and tolerances *tolr* (tolerance on residual) and *told* (tolerance on iteration length) do:

```
Set $q = p_n$.
For $k = 1, n$:
    1. Perform the Newton iteration: for $n = 0, 1, \dots$
        (a) Compute $q(x_n)$ and $q'(x_n)$ using synthetic division;
        (b) Compute $x_{n+1} = x_n - q'(x_n)^{-1} q(x_n)$;
        (c) If $|q(x_{n+1})| < tol$ and $|x_{n+1} = x_n|$ terminate iteration and set $\alpha_k = x_{n+1}$ (the $k$-th zero);
    2. Set $q = q/(x - x^{(k)})$ (deflation) using the associated polynomial given by the last synthetic division!;
    3. Set $x_0 = \overline{\alpha_k}$ (to facilitate finding conjugate zeroes);
```

This algorithm exploits the fact that the synthetic division (Horner's algorithm) is able to provide the evaluation at a point of both the polynomial and its derivative and, when the zero is found, also the coefficients of the deflated polynomial!

1.3 Some comments

Newton-Holder algorithm is a very nice and relatively efficient algorithm. However

- We have global convergence (i.e. for all x_0) only for polynomial with real coefficients under some conditions on the zeroes (look at specialized literature if you want more information). Therefore, in general convergence of the basic algorithm depends on the choice of x_0. You may implement techniques like backtracking, or randomization, to better convergence properties. It must be said that in all the tests I have made the basic method worked fine.

- The deflation is only approximate! Indeed, we will "divide" not by the real zero, but by the *approximated zero*. This means that we may accumulate errors that may reduce the accuracy of the last computed zeroes: a small value of the *approximated* deflated polynomial may not imply a small value of the original polynomial. Consequently, the algorithm may terminate at a point far from the real zero. However, this may happen only if you compute all zeroes of a polynomial of high degree, and for the last computed zeroes! Normally, you do not have to worry if you have set the tolerances reasonably tight.

- In real polynomials complex roots come in pairs. Cannot we deflate by $(x - \alpha)(x - \overline{\alpha})$ directly? Indeed we can, it is a modification of the basic algorithm that however I have not implemented here.

- You have to avoid division by zero in $q'(x_n)^{-1} p(x_n)$!. This is always a tricky business. What you can try to do is to set a small number at the denominator. For instance, using $p(x_n)/d$ with $d = \epsilon << 1$ if $|q'(x_n)| = 0$ and $d = q'(x_n)$ otherwise.

- Operating with complex numbers makes the algorithm more general. But if x_0 is a pure real number (i.e. zero imaginary part), then the algorithm will find only the real zeroes. Since we do not know how, in general, how many real zeroes we have, the algorithm may fail to find a zero simply because we do not have other real zeroes! Therefore, we need to modify the algorithm to detect non convergence. Here I have decided to keep things simple, but then you have always to start with an x_0 with non-sero imaginary part, unless you know beforehand that *all* zeroes are real (or, at least, the number of real zeroes, since the given algorithm allows to search only for a certain number of zeroes).
- When a complex numer is in fact a pure real number? The answer seems straightforward: when the imaginary part is zero. This is true in the ideal world of complex numbers, not in the real world of *floating point complex numbers*. Roundoff errors will often cause the imaginary part be very small, but not zero. A rule of thumb is to consider zero the imaginary part of the approximated root found by the algorithm if its absolute value is smaller than the given tolerances. Of course, if you know a priori that the root is real you will consider only the real part of the found approximation.

2 Derivatives

Leveraging Equations (3) and (2) we can also compute higher derivatives at point z. Let's first look to the second derivative

$$\frac{d^2 p_m}{dx^2}(x) = \frac{d^2 q_{m-1}}{dx^2}(x;z)(x-z) + \frac{dq_{m-1}}{dx}(x;z) + \frac{dq_{m-1}}{dx}(x;z) = \\ \frac{d^2}{dx^2}q_{m-1}(x;z)(x-z) + 2\frac{d}{dx}q_{m-1}(x;z), \tag{6}$$

Thus,

$$\frac{d^2 p_m}{dx^2}(z) = 2\frac{d}{dx}q_{m-1}(z;z), \tag{7}$$

Using (3),

$$\frac{d}{dx}q_{m-1}(x;z) = \frac{d}{dx}q_{m-2}(x;z)(x-z) + q_{m-2}(x;z), \tag{8}$$

which combined with (7) gives

$$\frac{d^2 p_m}{dx^2}(z) = 2q_{m-2}(z;z). \tag{9}$$

So the second derivative at z is twice the associated polynomial of $q_{m-1}(x;z)$ at the given point. So it can be computed just by using two synthetic divisions. What about the third derivative? We have, from (6)

$$\frac{d^3 p_m}{dx^3}(x) = \frac{d^3}{dx^3}q_{m-1}(x;z)(x-z) + \frac{d^2}{dx^2}q_{m-1}(x;z) + 2\frac{d^2}{dx^2}q_{m-1}(x;z) = \\ \frac{d^3}{dx^3}q_{m-1}(x;z)(x-z) + 3\frac{d^2}{dx^2}q_{m-1}(x;z). \tag{10}$$

Using (8)

$$\frac{d^2}{dx^2}q_{m-1}(x;z) = \frac{d^2}{dx^2}q_{m-2}(x;z)(x-z) + 2\frac{d}{dx}q_{m-2}(x;z), \tag{11}$$

and thus

$$\frac{d^3 p_m}{dx^3}(x) = \frac{d^3}{dx^3}q_{m-1}(x;z)(x-z) + 3\frac{d^2}{dx^2}q_{m-2}(x;z)(x-z)^2 + 6\frac{d}{dx}q_{m-2}(x;z), \tag{12}$$

from which, exploiting again (3), we have

$$\frac{d^3}{dx^3}p_m(z) = 6q_{m-3}(z; z). \tag{13}$$

In general, for any $n \leq m$,

$$\boxed{\frac{d^n}{dx^n}p_m(z) = n!q_{m-n}(z; z).} \tag{14}$$

Therefore, all derivatives at a given point z may be computed with a repeated use of synthetic division.

3 The code in `polyHolder.hpp`

The code in `polyHolder.hpp` contains

- The function polyEval() which uses the basic Holder's algoritm to compute the value of a polynomial at a point;
- The class polyHolder, which is a helper class for the Holder-Newton algorithm and it allows also to compute derivatives of a polynomial (given its coefficients) at a given point.
- The function polyRoots() which computes the roots of a polynomial using the basic *Newton-Holder* procedure.

You may argue that I am not respecting the *single responsibility principle* here: the class `polyHolder` should only provide the synthetic division. The computation of derivatives should be the responsibility of a function (or another class) that uses `polyHolder`. You are right, but the world is not perfect, and the *principles* should be taken with a grain of salt.