

# POLYNOMIAL RECURRENCE, NEWTON CORRECTION AND CONTINUED FRACTIONS

WALTER GANDER AND PEDRO GONNET\*

*Dedicated to Gene H. Golub on occasion of his 75th birthday.*

**Key words.** Three-term recurrence, continued fraction, Newton's method, orthogonal polynomials, Newton-Maehly, avoiding overflow. Classification: 11A55, 65Q05, 33C45, 65H05, 65F15, 65-04, 65F40.

**Abstract.** In this paper we consider pairs of recurrence relations where we are interested in the *ratio* of the iterates rather than in the iterates *per se*. Such recurrences occur i.e. when evaluating continued fractions, orthogonal polynomials or the characteristic polynomial of a symmetric tridiagonal matrix. Although such recurrence relations are compact and straight-forward to evaluate, the function value and the derivative might overflow while the ratio may still be a reasonable machine number. We describe, in this paper, how to avoid overflow and derive a new recurrence relation which allows us to compute the ratio directly.

**1. Introduction.** In this paper we consider pairs of recurrence relations where we are interested in the *ratio* of the iterates rather than in the iterates *per se*.

Such recurrences occur i.e. when evaluating continued fractions, orthogonal polynomials or the characteristic polynomial of a symmetric tridiagonal matrix.

When we evaluate a polynomial by a three-term recurrence relation then this recurrence may also be used to compute the derivative. For computing zeros with Newton's method we need the function value  $f(x)$  and the derivative  $f'(x)$ . For the Newton correction, however, we need only the ratio  $f(x)/f'(x)$ . The function value and the derivative might overflow while the ratio may still be a reasonable machine number.

In this paper we show how to avoid overflow and derive a new recurrence relation which allows us to compute the ratio directly.

**2. Quadrature, Orthogonal Polynomials and Tridiagonal Matrices.** In their famous paper [3] Golub and Welsch showed how to generate Gauss-Quadrature-Rules efficiently.

It is well known (see [2]) that given a nonnegative integrable weight function  $\omega$  on an interval  $[a, b] \subset \mathbb{R}$  there exists a sequence of polynomials  $p_0(x), p_1(x), \dots$  which are orthonormal with respect to  $\omega(t)$  and in which  $p_n(x)$  is of exact degree  $n$  so that

$$\langle p_m, p_n \rangle = \int_a^b p_m(t)p_n(t)\omega(t)dt = \begin{cases} 1 & \text{when } m = n \\ 0 & \text{when } m \neq n \end{cases}$$

The polynomial  $p_n$  has  $n$  real simple roots  $t_i$  which are the nodes in the corresponding Gauss-Quadrature rule:

$$\int_a^b f(t)\omega(t) dt \approx \sum_{j=1}^n w_j f(t_j).$$

The orthogonal polynomials obey a three-term recurrence relation. Initializing  $p_{-1}(x) \equiv 0$ ,  $p_0(x) \equiv 1$ ,  $\beta_0 = 0$ , then

$$p_{k+1}(x) = (x - \alpha_{k+1})p_k(x) - \beta_k p_{k-1}(x), \quad k = 0, 1, 2, \dots \quad (2.1)$$

---

\*Institute of Computational Science, ETH Zurich, gander@inf.ethz.ch, gonnetp@inf.ethz.ch

with

$$\alpha_{k+1} = \frac{\langle x p_k, p_k \rangle}{\|p_k\|^2} \quad \text{and} \quad \beta_k = \frac{\|p_k\|^2}{\|p_{k-1}\|^2},$$

holds for the monic polynomials.

By differentiating the recurrence (2.1) we obtain a second recurrence to compute the derivative  $p'_k(x)$ :

$$\begin{aligned} p'_0(x) &= 0, & p'_1(x) &= 1 \\ p'_{k+1}(x) &= p_k(x) + (x - \alpha_{k+1})p'_k(x) - \beta_k p'_{k-1}(x) \quad k = 1, 2, \dots \end{aligned} \quad (2.2)$$

The Matlab function `evalpol1` (Algorithm 1) computes the function values  $p_n(x)$  and  $p'_n(x)$  for a given  $x$  and returns the ratio  $r = p_n(x)/p'_n(x)$ .

---

**Algorithm 1** Evaluating a Polynomial and the Derivative

---

```
function r = evalpol1(x,alpha,beta);
% EVALPOL1 evaluates the three-term recurrence with
%           its derivative and returns the ratio
n = length(alpha);
p1s = 0; p1 = 1;
p2s = 1; p2 = x-alpha(1);
for k = 1:n-1
    p0s = p1s; p0 = p1;
    p1s = p2s; p1 = p2;
    p2s = p1 + (x-alpha(k+1))*p1s - beta(k)*p0s;
    p2 = (x-alpha(k+1))*p1 - beta(k)*p0;
end
r = p2/p2s;
```

---

Golub and Welsch use the recurrence relation (2.1) to replace the problem of computing the zeros of orthogonal polynomials with the equivalent problem of computing the eigenvalues of a symmetric tridiagonal matrix. The eigenvalues of such a matrix

$$T = \begin{pmatrix} a_1 & b_1 & & & \\ b_1 & a_2 & \ddots & & \\ & \ddots & \ddots & b_{n-1} & \\ & & & b_{n-1} & a_n \end{pmatrix},$$

are real and simple if  $T$  is unreduced, i.e.  $b_i \neq 0$  for  $i = 1, \dots, n-1$ . The characteristic polynomial

$$f_n(\lambda) = \det(\lambda I - T) \quad (2.3)$$

can be computed by the three-term recurrence

$$\begin{aligned} f_0(\lambda) &= 1, & f_1(\lambda) &= \lambda - a_1 \\ f_j(\lambda) &= (\lambda - a_j)f_{j-1}(\lambda) - b_{j-1}^2 f_{j-2}(\lambda), \quad j = 2, 3, \dots, n \end{aligned} \quad (2.4)$$

which can be derived by the Laplace expansion of the last row of the determinant.

Therefore, three-term recurrences for polynomials occur when computing zeros of orthogonal polynomials and also when computing eigenvalues of tridiagonal matrices by means of the characteristic polynomial.

**3. Newton-Maehly Iteration.** H. J. Maehly [5] describes an implementation of Newton's method which is well suited for computing zeros of polynomials which are evaluated by three-term recurrences.

Maehly uses the technique of *suppression* rather than *deflation*. With suppression computed zeros are implicitly deflated. Suppose that we have already computed the zeros  $x_1, \dots, x_k$ . Then consider the polynomial

$$P_{n-k}(x) := \frac{P_n(x)}{(x-x_1)\cdots(x-x_k)}.$$

Because of

$$\frac{d}{dx} \left( \prod_{i=1}^k (x-x_i)^{-1} \right) = - \prod_{i=1}^k (x-x_i)^{-1} \sum_{i=1}^k (x-x_i)^{-1}$$

the derivative becomes

$$P'_{n-k}(x) = \frac{P'_n(x)}{(x-x_1)\cdots(x-x_k)} - \frac{P_n(x)}{(x-x_1)\cdots(x-x_k)} \sum_{i=1}^k \frac{1}{x-x_i}.$$

Using this expression we obtain for the Newton step:

$$x_{new} = x - \frac{P_{n-k}(x)}{P'_{n-k}(x)} = x - \frac{P_n(x)}{P'_n(x)} \frac{1}{1 - \frac{P_n(x)}{P'_n(x)} \sum_{i=1}^k \frac{1}{x-x_i}}. \quad (3.1)$$

Equation (3.1) defines the *Newton-Maehly* iteration for computing zeros of  $P_n(x)$ .

In [1] Bulirsch and Stoer describe important details of the implementation of the Newton-Maehly method for polynomials with only real roots. They notice that Newton's method for polynomials has a bad global behavior. This because if  $f(x) = a_n x^n + \cdots + a_0$  then  $f'(x) = n a_n x^{n-1} + \cdots + a_1$  and if  $a_n x^n$  is the dominant term then the Newton step is approximately

$$x_{new} = x - \frac{f(x)}{f'(x)} \approx x - \frac{a_n x^n}{n a_n x^{n-1}} = x - \frac{1}{n} x = x \left(1 - \frac{1}{n}\right).$$

They propose to start the iteration on the right of the largest zero and to initially take *double steps*

$$x_{new} = x - 2 \frac{f(x)}{f'(x)}$$

to improve global convergence. It is shown in [1] that the following holds:

We denote by  $\xi$  the next zero and by  $\alpha < \xi$  the next local extremum of  $f$ . For the usual Newton step starting from  $x > \xi$  we obtain  $z$ :

$$z = x - \frac{f(x)}{f'(x)}.$$

For the double Newton step starting from  $x$  we obtain  $w$ :

$$w = x - 2 \frac{f(x)}{f'(x)}.$$

For the Newton step from  $w$  we get

$$y = w - \frac{f(w)}{f'(w)}.$$

The following then holds:

1.  $\alpha < w$ : with the double step iteration we arrive *before* the local extremum.
2.  $\xi \leq y \leq z$ : the backward Newton step lies between the zero  $\xi$  and the simple Newton step  $z$ .

These considerations lead to the following algorithm:

1. Start with the Newton double step iteration *from the right of the first zero* and iterate

$$x := y; \quad y := x - 2 \frac{f(x)}{f'(x)}$$

until numerically  $y \geq x$ .

2. Continue to iterate with normal Newton steps

$$x := y; \quad y := x - \frac{f(x)}{f'(x)}$$

until again monotonicity is lost:  $y \geq x$ . This time we have computed the zero  $\xi = y$  to machine precision.

3. Suppress the new zero and continue the iteration for the next zero with the value  $x < \xi$  found by the Newton double step iteration as initial value.

We obtain the Matlab function `newtonmaehly` (Algorithm 2).

As an example we compute the eigenvalues of the one dimensional Laplace operator (a symmetric tridiagonal matrix with elements  $-2$  in the diagonal and  $1$  in the sub-diagonal). The exact eigenvalues are

$$-4 \sin^2 \left( \frac{j\pi}{2(n+1)} \right), \quad j = 1, \dots, n$$

```
n = 400
xi = newtonmaehly(@evalpol1,-2*ones(n,1),ones(n-1,1),n,4);
ex = sort(-4*sin([1:n]'*pi/2/(n+1)).^2);
norm(sort(xi)-ex)
T = diag(ones(n-1,1), -1) +diag(ones(n-1,1), 1) +diag(-2*ones(n,1));
lamb = sort(eig(T));
norm(lamb-ex)
```

We obtain the result

```
n =
    400
ans = 6.048660967720208e-15
ans = 1.545531485323698e-14
```

which shows that we get results that are slightly better than those obtained by the Matlab `eig` function. However, for  $n = 500$  `newtonmaehly` fails, because `evalpol1` produces an overflow in computing the determinant!

---

**Algorithm 2** Newton-Maehly Algorithm

---

```
function xi = newtonmaehly(evalpol, alpha,beta,K,x0);
% NEWTONMAEHLY computes K zeros of the polynomial defined
%           by the recurrence coefficients alpha and beta.
%           Uses EVALPOL. The first start value x0 must be
%           right of all real zeros.
for k = 1 : K,
    y = x0; finished =0;
    m = 2; % Newton double step
    while ~finished,
        x = y;
        r = evalpol(x,alpha,beta);
        % Maehly correction for suppressing zeros
        s = 0;
        if k>1
            s = sum(1./(x-xi(1:k-1)));
        end
        y = x - m*r/(1-r*s);
        if y >= x,
            if m == 1, finished = 1;
            end
            if ~finished; % stop double step
                x0 = x; m = 1;
            end
        end
    end
    xi(k) = y;
end
xi = xi(:);
```

---

**4. Continued Fractions.** In Rutishauser's Algol 60 book [6] (an insider tip for numerical algorithms!) a continued fraction is defined by

$$\cfrac{f_1}{g_1} + \cfrac{f_2}{g_2} + \cfrac{f_3}{g_3} + \dots + \cfrac{f_m}{g_m} = \cfrac{f_1}{g_1 + \cfrac{f_2}{g_2 + \cfrac{\dots}{g_2 + \cfrac{f_m}{\dots + \cfrac{f_m}{g_m}}}}} \quad (4.1)$$

where  $f_k$  and  $g_k$  are given functions of the subscript  $k$ .

A continued fraction (4.1) can be evaluated by *backward recurrence* which uses the fact that the relation  $s^{(k)} = f_k/(g_k + s^{(k+1)})$  holds between the values

$$s^{(k)} = \cfrac{f_k}{g_k} + \cfrac{f_{k+1}}{g_{k+1}} + \dots + \cfrac{f_m}{g_m}, \quad k = 1, 2, \dots, m.$$

Given the vectors of nominators  $\mathbf{f}$  and denominators  $\mathbf{g}$  the Matlab function `cf` evaluates the continued fraction by backwards recurrence (Algorithm 3).

Usually we do not know the value  $m$ , i.e. where to start the backwards recurrence. We would prefer to truncate an infinite continued fraction for some *a priori* unknown  $m$  when it approximates the limit to some given tolerance  $\tau$ . Backward recurrence is

---

**Algorithm 3** Continued Fraction by Backwards Recurrence

---

```
function y = cf(f,g)
% CF evaluates a continued fraction using
% backwards recurrence
s = 0;
for k = length(f):-1:1
    s = f(k)/(s+g(k));
end
y = s;
```

---

therefore not appropriate for this case. It is, however, well known (see [7]) that a *forward recurrence* exists for successive nominators  $A_k$  and denominators  $B_k$  of the  $k$ -th convergent of a continued fraction such that its value is obtained as  $\lim_{k \rightarrow \infty} (A_k/B_k)$ . With the initialization

$$B_0 = 1, \quad A_0 = 0, \quad A_1 = f_1, \quad B_1 = g_1$$

the numerators  $A_k$  and the denominators  $B_k$  are computed by the same recurrence

$$\begin{aligned} A_k &= g_k A_{k-1} + f_k A_{k-2}, \\ B_k &= g_k B_{k-1} + f_k B_{k-2}, \end{aligned} \quad k = 2, 3, \dots \quad (4.2)$$

until two convergents match the tolerance

$$\left| \frac{A_k}{B_k} - \frac{A_{k-1}}{B_{k-1}} \right| < \tau.$$

A translation of the Algol program of Rutishauser ([6], page 117) is the Matlab function `forwrec1` (Algorithm 4).

**4.1. Numerical Examples.** As a first example consider the infinite continued fraction

$$e^x = \cfrac{1}{1} - \cfrac{x}{1} + \cfrac{x}{2} - \cfrac{x}{3} + \cfrac{x}{2} - \cfrac{x}{5} + \cfrac{x}{2} \mp \dots \quad (4.3)$$

For  $x = 1$  the coefficients become

$$g_1 = 1, \quad \left. \begin{aligned} f_k &= (-1)^{k+1} \\ g_{2k+1} &= 2 \\ g_{2k} &= 2k - 1 \end{aligned} \right\} k = 1, 2, 3, \dots$$

With the Matlab statements

```
m = 8; n = 2*m; v=1;
for k = 1:n
    g(k) = 2; f(k) = v; v=-v;
end
g(1) = 1;
for k=1:m
    g(2*k) = 2*k-1;
end
y = cf(f,g)
[yy k]= forwrec1(f,g,n,3,1e-12)
```

---

**Algorithm 4** Continued Fraction Evaluation by Forward Recurrence

---

```
function [cf, k]= forwrec1(ff,gg,kmax,r,tau)
% FORWREC1 evaluates the continued fraction given by
%       the two vectors ff and gg by forward recurrence.
%       every r step it is checked if two consecutive
%       partial fractions match to tolerance tau.
p=1; q=2; nl = 2;
a(nl+1) = 0; b(nl+1) =1;
a(nl-1) = ff(1); b(nl-1) = gg(1);
for k=2:kmax
    f = ff(k); g = gg(k);
    a(nl+p) = g*a(nl-p) + f*a(nl+p);
    b(nl+p) = g*b(nl-p) + f*b(nl+p);
    if k==q*r | k==kmax
        cf = a(nl+p)/b(nl+p);
        cg = a(nl-p)/b(nl-p);
        if abs(cf-cg)<tau
            break
        end
        q = q+1;
    end
    p = -p;
end
```

---

the constant  $e$  is computed to machine precision with both recurrences. The continued fraction (4.3) converges quickly. It may happen that some  $B_i = 0$  and then the convergent  $A_i/B_i$  does not exist. In this example indeed  $B_2$  becomes zero. This is not crucial, we just have to skip the convergence test and continue with the computation.

A second example is a continued fraction for  $\pi$  given by Lange [4]

$$\pi = 3 + \cfrac{1}{6} + \cfrac{9}{6} + \cfrac{25}{6} + \cfrac{49}{6} + \cfrac{81}{6} + \cfrac{121}{6} + \dots \quad (4.4)$$

The continued fraction for  $\pi - 3$  has remarkably simple coefficients

$$\left. \begin{array}{l} f_1 = 1 \\ f_k = (2k - 1)^2 \\ g_k = 6 \end{array} \right\} k = 1, 2, 3, \dots$$

however, it converges very slowly. For  $n = 100$ , running the Matlab statements

```
n = 100
f = 1:n; f = ones(size(f)) + 2*f; f = [1 f.*f];
g = 6*ones(size(f));
y = cf(f,g)
[yy,k] = forwrec1(f,g,n+1,3,1e-10)
```

we get for both recurrences the result 0.14159288914208 which has about 6 correct decimal digits. If we increase  $n = 150$  the backward recurrence delivers 0.14159272477434 (about 7 digits) while with the forward recurrence we get NaN because of overflow.

**5. Rutishauser's Solution.** To overcome the problem of overflow in continued fraction evaluations, Rutishauser suggests to check, from time to time, the size of  $A_k$ ,  $B_k$ ,  $A_{k-1}$ ,  $B_{k-1}$  and simply rescale them if needed.

Algorithm 5 is a translation of the ALGOL program on page 127 of [6] to Matlab. Rescaling should not only avoid overflow but underflow as well.

---

**Algorithm 5** Continued Fraction Evaluation by Forward Recurrence with Rescaling

---

```
function [cf, k]= forwrec2(ff,gg,kmax,tau)
% FORWREC2 evaluates the continued fraction given by
%         the two vectors ff an gg by forward recurrence.
%         Convergence and rescaling are checked for
%         every 10 steps
p=1; nl = 2;
a(nl+1) = 0; b(nl+1) =1;
a(nl-1) = ff(1); b(nl-1) = gg(1);
for k=0:10:kmax-11
    for j=2:11
        f = ff(j+k); g = gg(j+k);
        a(nl+p) = g*a(nl-p) + f*a(nl+p);
        b(nl+p) = g*b(nl-p) + f*b(nl+p);
        p = -p;
    end
    cf = a(nl+1)/b(nl+1);
    cg = a(nl-1)/b(nl-1);
    if abs(cf-cg) < tau
        break
    end
    maxx = abs(a(nl+1))+abs(a(nl-1))+abs(b(nl+1))+abs(b(nl-1));
    if maxx>1e20
        d = 1e-20;
    elseif maxx<1e-20
        d = 1e20;
    else
        d = 1;
    end;
    a(nl+1) = d*a(nl+1); b(nl+1)= d*b(nl+1);
    a(nl-1) = d*a(nl-1); b(nl-1)= d*b(nl-1);
end
```

---

In Algorithm 5, convergence and the need for rescaling are checked for every 10th iteration. Rescaling indeed works and we are now able to compute the partial fraction for  $\pi$  for larger  $n$ . With the statements

```
n = 460
f = 1:n; f = ones(size(f)) +2*f; f = [1 f.*f];
g = 6*ones(size(f));
[yy,k] = forwrec2(f,g,n+1,0)
```

we get the result 0.14159265103806. However, for  $n = 500$  again the result is NaN because of overflow! This happens because the  $f_i$  grow too quickly and scaling only every 10th iteration is not enough to prevent overflow.

Rescaling is also possible when evaluating the polynomials  $f$  and  $f'$  (`evalpol2`, Algorithm 6).

Replacing `evalpol` in `newtonmaehly` by Algorithm 6 `evalpol2` we can now compute the eigenvalues of the Laplace operator for much larger  $n$  without suffering from overflow. Some examples are given in Table (5.1) which shows the norm of the difference of the exact eigenvalues and the ones computed by the functions `newtonmaehly`

---

**Algorithm 6** Evaluating a Polynomial and the Derivative with Rescaling

---

```
function r = evalpol2(x,alpha,beta);
% EVALPOL2 evaluates the three-term recurrence with
%           its derivative with rescaling and returns
%           the ratio
n = length(alpha);
p1s = 0; p1 = 1;
p2s = 1; p2 = x-alpha(1);
for k = 1:n-1
    p0s =p1s; p0=p1;
    p1s = p2s; p1 =p2;
    p2s = p1 + (x-alpha(k+1))*p1s - beta(k)*p0s;
    p2 = (x-alpha(k+1))*p1 - beta(k)*p0;
    maxx = abs(p2)+abs(p2s);
    if maxx > 1e20;
        d = 1e-20;
    elseif maxx < 1e-20
        d = 1e20;
    else
        d = 1;
    end
    p1 = p1*d; p2 =p2*d;
    p1s = p1s*d; p2s = p2s*d;
end
r = p2/p2s;
```

---

and eig.

$n$	newtonmaehly	eig
600	$7.1149e-15$	$1.9086e-14$
1000	$8.9523e-15$	$2.8668e-14$
5000	$2.0565e-14$	$9.3223e-14$
10000	$2.8705e-14$	---

(5.1)

It is quite impressive that we are able to compute the zeros of a polynomial of degree  $n = 10'000$  using Newton's method to such a high accuracy this way.

**6. New Algorithm for Continued Fractions.** We are interested in computing the convergents  $A_k/B_k$ , yet so far we used recurrences for numerator and denominator separately. The question remains whether we can find recurrences which compute the ratio directly, thus avoiding problems with overflow right from the beginning.

Using the recurrence relations (4.2) we obtain

$$r_k := \frac{B_k}{A_k} = \frac{g_k B_{k-1} + f_k B_{k-2}}{g_k A_{k-1} + f_k A_{k-2}}$$

We will compute the inverse of the convergent since it requires less divisions than computing the convergent directly. Dividing both the numerator and the denominator

by  $A_{k-1}$  we get

$$r_k = \frac{B_k}{A_k} = \frac{g_k \frac{B_{k-1}}{A_{k-1}} + f_k \frac{B_{k-2}}{A_{k-1}}}{g_k + f_k \frac{A_{k-2}}{A_{k-1}}} = \frac{g_k r_{k-1} + f_k v_{k-1}}{g_k + f_k u_{k-1}}$$

where we have defined the auxiliary quantities

$$u_k := \frac{A_{k-1}}{A_k} \quad \text{and} \quad v_k := \frac{B_{k-1}}{A_k}.$$

Using again the recurrences (4.2) and reducing by  $A_{k-1}$  we get

$$u_k = \frac{A_{k-1}}{A_k} = \frac{A_{k-1}}{g_k A_{k-1} + f_k A_{k-2}} = \frac{1}{g_k + f_k u_{k-1}}$$

Similarly for  $v_k$  again reducing by  $A_{k-1}$  we obtain

$$v_k = \frac{B_{k-1}}{A_k} = \frac{B_{k-1}}{g_k A_{k-1} + f_k A_{k-2}} = \frac{r_{k-1}}{g_k + f_k u_{k-1}}$$

We have thus derived recurrence relations for  $r_k$ ,  $u_k$  and  $v_k$ . To use them we need initial values. The recurrences for nominators and denominators (4.2) allow us to compute

$k$	0	1
$A_k$	0	$f_1$
$B_k$	1	$g_1$
$r_k = B_k/A_k$		$g_1/f_1$
$u_k = A_{k-1}/A_k$		0
$v_k = B_{k-1}/A_k$		$1/f_1$

With these initial values we obtain the Matlab function `forwrec3` (Algorithm 7).

Using Algorithm 7 we can evaluate the continued fraction (4.4) without overflow problems for large  $n$ :

```
n = 1000
f = 1:n; f = ones(size(f)) + 2*f; f = [1 f.*f];
g = 6*ones(size(f));
[yy,k] = forwrec3(f,g,n+1,0)
```

We obtain the following results:

$n$	<code>forwrec3</code>	$A_n/B_n - (\pi - 3)$
500	0.14159265556597	$-1.9761e-9$
1000	0.14159265383830	$2.4850e-10$
10000	0.14159265359004	$-2.4688e-13$
15000	0.14159265358986	$-6.6890e-14$

For  $n = 15000$  we are near machine precision.

**7. The New Algorithm for Newton Correction.** We consider the recurrence for the polynomials  $f_n$

$$f_n(x) = (x - \alpha_n)f_{n-1}(x) - \beta_{n-1}f_{n-2}(x)$$

and their derivatives

$$f'_n(x) = f'_{n-1}(x) + (x - \alpha_n)f''_{n-1}(x) - \beta_{n-1}f'_{n-2}(x)$$

---

**Algorithm 7** Direct Computation of  $A_k/B_k$ 

---

```
function [y,k] = forwrec3(f,g,kmax,tau);
% FORWREC3 computes successive convergents for the
% continued fraction given by the nominators f and
% denominators g.
%
r(1) = g(1) / f(1);
u(1) = 0;
v(1) = 1 / f(1);
for k=2:kmax
    u(k) = 1 / (g(k) + f(k)*u(k-1));
    r(k) = (g(k)*r(k-1) + f(k)*v(k-1)) * u(k);
    v(k) = r(k-1) * u(k);
    if abs(1/r(k)-1/r(k-1))<tau
        break
    end
end;
y = 1/r(k);
```

---

with the initializations

$$f_{-1}(x) = 0, \quad f_0(x) = 1, \quad f'_0(x) = 0, \quad f'_1(x) = 1.$$

We will start by computing the ratio

$$r_n(x) = \frac{f'_n(x)}{f_n(x)}$$

which is the inverse of the ratio needed for Newton's Method. As with the recursion for the continued fractions, we will use the inverse since in its final derivation, it requires less divisions than computing  $f_n(x)/f'_n(x)$  and is hence more efficient.

To simplify computations we replace, in the following,  $\alpha_i := x - \alpha_i$ . We expand both the numerator and the denominator of the right hand side and obtain

$$r_n = \frac{f_{n-1} + \alpha_n f'_{n-1} - \beta_{n-1} f'_{n-2}}{\alpha_n f_{n-1} - \beta_{n-1} f_{n-2}}. \quad (7.1)$$

If we divide both the numerator and the denominator in Equation (7.1) with  $f_{n-1}$  we obtain

$$r_n = \frac{1 + \alpha_n \frac{f'_{n-1}}{f_{n-1}} - \beta_{n-1} \frac{f'_{n-2}}{f_{n-1}}}{\alpha_n - \beta_{n-1} \frac{f_{n-2}}{f_{n-1}}}. \quad (7.2)$$

Since we want to compute the  $r_i$  iteratively, we can replace the first fraction in the numerator with  $r_{n-1}$ . We also introduce the two terms

$$u_n = \frac{f_{n-1}}{f_n} \quad (7.3)$$

and

$$v_n = \frac{f'_{n-1}}{f_n}. \quad (7.4)$$

Substituting these into Equation (7.2) we obtain

$$r_n = \frac{1 + \alpha_n r_{n-1} - \beta_{n-1} v_{n-1}}{\alpha_n - \beta_{n-1} u_{n-1}}. \quad (7.5)$$

which now only depends on the terms  $r_{n-1}$ ,  $u_{n-1}$  and  $v_{n-1}$  of the previous iteration.

In order to compute  $u_n$  and  $v_n$ , we expand Equations (7.3) and (7.4) much in the same way as we did  $r_n$ :

$$\begin{aligned} u_n &= \frac{f_{n-1}}{\alpha_n f_{n-1} - \beta_{n-1} f_{n-2}} \\ &= \frac{1}{\alpha_n - \beta_{n-1} \frac{f_{n-2}}{f_{n-1}}} \\ &= \frac{1}{\alpha_n - \beta_{n-1} u_{n-1}} \end{aligned} \quad (7.6)$$

and

$$\begin{aligned} v_n &= \frac{f'_{n-1}}{\alpha_n f_{n-1} - \beta_{n-1} f_{n-2}} \\ &= \frac{\frac{f'_{n-1}}{f_{n-1}}}{\alpha_n - \beta_{n-1} \frac{f_{n-2}}{f_{n-1}}} \\ &= \frac{r_{n-1}}{\alpha_n - \beta_{n-1} u_{n-1}} \end{aligned} \quad (7.7)$$

Using Equations (7.5), (7.6) and (7.7) we can compute  $r_n$  iteratively using

$$\begin{aligned} u_n &:= (\alpha_n - \beta_{n-1} u_{n-1})^{-1} \\ r_n &:= (1 + \alpha_n r_{n-1} - \beta_{n-1} v_{n-1}) u_n \\ v_n &:= r_{n-1} u_n \end{aligned} \quad (7.8)$$

**8. Numerical Considerations.** In Equation (7.8) we compute the ratio  $f'_n/f_n$  without ever evaluating any of the  $f_i$  or  $f'_i$ . Although we avoid potential overflows, we are prone to two new problems:

- If any of the  $f_i$ ,  $i < n$  are zero, the resulting  $u_i$  will cause a division by zero.
- If one of the  $f_i$ ,  $i < n$  is small compared to the previous values  $f_{i-1}$ , the subtraction in computing  $u_i$  will lose precision due to cancellation, resulting either in a loss of precision in the result or ultimately a division by zero.

If we detect a small value in  $f_{n-1}$ , we can simply skip the  $(n-1)^{\text{st}}$  terms in the computation of relation by using the expanded recursions.

$$\begin{aligned} f_n &= \alpha_n (\alpha_{n-1} f_{n-2} - \beta_{n-2} f_{n-3}) - \beta_{n-1} f_{n-2} \\ &= (\alpha_n \alpha_{n-1} - \beta_{n-1}) f_{n-2} - \alpha_n \beta_{n-2} f_{n-3} \end{aligned} \quad (8.1)$$

---

**Algorithm 8** Direct Computation of the Newton Correction

---

```
function res = evalpol3(x, alpha, beta )
% EVALPOL3  evaluates the ratio of the  three term recurrence
%           and its derivative without evaluation of f and f'
tol = 1.0e-5;
n = length(alpha);
alpha = x - alpha;
r = [0,1/alpha(1),0];
u = [0,1/alpha(1),0];
v = [0,0,0];
skip = 0;
for i = 2:n
    t_0 = alpha(i) - beta(i-1)*u(2);
    if ~skip & (abs(t_0) <= tol)
        skip = 1;
    elseif skip
        t_1 = alpha(i-1) - beta(i-2)*u(1);
        t_2 = 1 / ( alpha(i)*t_1 - beta(i-1) );
        r(3) = ( t_1 + alpha(i) + (alpha(i)*alpha(i-1) ...
                - beta(i-1))*r(1) -alpha(i)*beta(i-2)*v(1) ) * t_2;
        u(3) = (alpha(i-1) - beta(i-2)*u(1)) * t_2;
        v(3) = (1 + alpha(i-1)*r(1) - beta(i-2)*v(1)) * t_2;
        skip = 0;
    else
        u(3) = 1 / t_0;
        r(3) = (1 + alpha(i)*r(2) - beta(i-1)*v(2)) * u(3);
        v(3) = r(2) * u(3);
    end;
    r(1) = r(2); r(2) = r(3);
    u(1) = u(2); u(2) = u(3);
    v(1) = v(2); v(2) = v(3);
end;
if skip
    res = (alpha(i) - beta(i-1)*u(2)) / ...
        (1 + alpha(i)*r(2) - beta(i-1)*v(2));
else
    res = 1 / r(3);
end;
```

---

and

$$\begin{aligned} f'_n &= (\alpha_{n-1}f_{n-2} - \beta_{n-2}f_{n-3}) + \alpha_n (f_{n-2} + \alpha_{n-1}f'_{n-2} - \beta_{n-2}f'_{n-3}) - \beta_{n-1}f'_{n-2} \\ &= (\alpha_{n-1} + \alpha_n) f_{n-2} - \beta_{n-2}f_{n-3} + (\alpha_n\alpha_{n-1} - \beta_{n-1}) f'_{n-2} - \alpha_n\beta_{n-2}f'_{n-3} \quad (8.2) \end{aligned}$$

It should be noted that it is safe to assume that  $f_{n-2}$  will not be zero and hence have the same problems, since the zeros of such polynomials are simple and interlacing. If  $f_{n-1}(x)$  is zero, then  $f_n(x)$  and  $f_{n-2}(x)$  are not.

We can use these two expressions to construct  $r_n$

$$\begin{aligned}
r_n &= \frac{(\alpha_{n-1} + \alpha_n) f_{n-2} - \beta_{n-2} f_{n-3} + (\alpha_n \alpha_{n-1} - \beta_{n-1}) f'_{n-2} - \alpha_n \beta_{n-2} f'_{n-3}}{(\alpha_n \alpha_{n-1} - \beta_{n-1}) f_{n-2} - \alpha_n \beta_{n-2} f_{n-3}} \\
&= \frac{(\alpha_{n-1} + \alpha_n) - \beta_{n-2} \frac{f_{n-3}}{f_{n-2}} + (\alpha_n \alpha_{n-1} - \beta_{n-1}) \frac{f'_{n-2}}{f_{n-2}} - \alpha_n \beta_{n-2} \frac{f'_{n-3}}{f_{n-2}}}{(\alpha_n \alpha_{n-1} - \beta_{n-1}) - \alpha_n \beta_{n-2} \frac{f_{n-3}}{f_{n-2}}} \\
&= \frac{(\alpha_{n-1} + \alpha_n) - \beta_{n-2} u_{n-2} + (\alpha_n \alpha_{n-1} - \beta_{n-1}) r_{n-2} - \alpha_n \beta_{n-2} v_{n-2}}{(\alpha_n \alpha_{n-1} - \beta_{n-1}) - \alpha_n \beta_{n-2} u_{n-2}} \quad (8.3)
\end{aligned}$$

For reasons which will become obvious later, we re-group the expressions as

$$r_n = \frac{(\alpha_{n-1} - \beta_{n-2} u_{n-2}) + \alpha_n + (\alpha_n \alpha_{n-1} - \beta_{n-1}) r_{n-2} - \alpha_n \beta_{n-2} v_{n-2}}{\alpha_n (\alpha_{n-1} - \beta_{n-2} u_{n-2}) - \beta_{n-1}} \quad (8.4)$$

The denominator only becomes zero when  $f_n$  is zero, which is not the case since we are skipping a zero at  $f_{n-1}$ .

Similarly, the extended equations for  $f_n$  and  $f'_n$  can be used to compute

$$\begin{aligned}
u_n &= \frac{\alpha_{n-1} f_{n-2} - \beta_{n-2} f_{n-3}}{(\alpha_n \alpha_{n-1} - \beta_{n-1}) f_{n-2} - \alpha_n \beta_{n-2} f_{n-3}} \\
&= \frac{\alpha_{n-1} - \beta_{n-2} \frac{f_{n-3}}{f_{n-2}}}{(\alpha_n \alpha_{n-1} - \beta_{n-1}) - \alpha_n \beta_{n-2} \frac{f_{n-3}}{f_{n-2}}} \\
&= \frac{\alpha_{n-1} - \beta_{n-2} u_{n-2}}{\alpha_n (\alpha_{n-1} - \beta_{n-2} u_{n-2}) - \beta_{n-1}} \quad (8.5)
\end{aligned}$$

and

$$\begin{aligned}
v_n &= \frac{f_{n-2} + \alpha_{n-1} f'_{n-2} - \beta_{n-2} f'_{n-3}}{(\alpha_n \alpha_{n-1} - \beta_{n-1}) f_{n-2} - \alpha_n \beta_{n-2} f_{n-3}} \\
&= \frac{1 + \alpha_{n-1} \frac{f'_{n-2}}{f_{n-2}} - \beta_{n-2} \frac{f'_{n-3}}{f_{n-2}}}{(\alpha_n \alpha_{n-1} - \beta_{n-1}) - \alpha_n \beta_{n-2} \frac{f_{n-3}}{f_{n-2}}} \\
&= \frac{1 + \alpha_{n-1} r_{n-2} - \beta_{n-2} v_{n-2}}{\alpha_n (\alpha_{n-1} - \beta_{n-2} u_{n-2}) - \beta_{n-1}}. \quad (8.6)
\end{aligned}$$

Using these equations for  $r_n$ ,  $u_n$  and  $v_n$ , we get Algorithm 8. Replacing `evalpol3` instead of `evalpol2` in `newtonmaehly` we obtain the results

$n$	<code>newtonmaehly</code>	<code>eig</code>
600	$7.1384e-15$	$1.9086e-14$
1000	$8.9651e-15$	$2.8668e-14$
5000	$2.0577e-14$	$9.3223e-14$
10000	$2.8704e-14$	---

(8.7)

The results are (up to rounding errors) the same results as in Table (5.1). Of course the recurrences require more operations to compute, but the overflow is now eliminated right from the beginning.

## REFERENCES

- [1] J. STOER AND R. BULIRSCH, *Introduction to Numerical Analysis*, Springer, 1991.
- [2] WALTER GAUTSCHI, *Orthogonal Polynomials: Computation and Approximation*, Oxford Univ Press, August 2004.
- [3] G. H. GOLUB AND J. H. WELSCH, *Calculation of Gauss Quadrature Rules*, Math. Comp., 23 (1969), pp. 221–230.
- [4] L. J. LANGE, *An elegant continued fraction for  $\pi$* , The American Mathematical Monthly, 106 (1999), pp. 456–458.
- [5] H. J. MAEHLY, *Zur iterativen Auflösung algebraischer Gleichungen*, ZAMP (Zeitschrift für angewandte Mathematik und Physik), (1954), pp. 260–263.
- [6] H. RUTISHAUSER, *Description of Algol 60*, Springer, 1967.
- [7] H. S. WALL, *Analytic Theory of Continued Fractions*, AMS Chelsea Publishing 1948; 433 pp; reprinted 1973; first AMS printing 2000.