Efficient Computation of Truncated Power Series: Direct Approach versus Newton's Method

Wolfram Koepf, Department of Mathematics, University of Kassel,
Heinrich-Plett-Str. 40, D-34132 Kassel, Germany,
koepf@mathematik.uni-kassel.de

Abstract

Newton's method is used to approximate the zeros of a real function $f(x)$. In the generic case, Newton's method is quadratically convergent, i. e. in each step the number of correct decimal digits roughly doubles. It is also well-known that Newton's method can be similarly used to compute the Taylor coefficients of a function $x(t)$, given implicitly by $f(t, x(t)) = 0$, in an iterative way such that in each iteration step the number of correct coefficients doubles.

In this paper, we present implementations of higher order iteration schemes generalizing Newton's method which enable us to triple, quadruple, quintuple etc. the number of correct Taylor coefficients of an implicitly given function in each iteration step. These algorithms are of theoretical interest, but in practice they turn out to be not as efficient as the "usual" Newton method since the expression swell generated by the complexity of the formulas is higher than the advantage by the higher order of the method.

We give examples in *Mathematica* and in *Maple* showing that in certain cases Newton's (implicit) method is faster than the direct (explicit) computation.

Keywords: Implicit functions; computation of Taylor polynomials; Newton-Raphson method; quadratically convergent iteration; cubic and quartic iteration; generating functions; Catalan numbers; Lambert's W function.

1 Newton's Method

A general assumption of this article is that the functions occurring are often enough differentiable or else they are members of a suitable differential field so that the statements make sense.

In this section, the function $f : \mathbb{R} \to \mathbb{R}$ should be differentiable. Then to find a zero ξ of f, one can approximate the function by its linear approximation

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) , \tag{1}$$

hence the graph of f is approximated by its tangent. If we are lucky (i. e. if f is not too wild in a neighborhood of ξ), then we will get a better approximation of ξ if we search for a zero of (1) which gives

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} ,$$

which is the zero of the tangent considered. It is "graphically evident" that if one is near enough to ξ then x_1 should be a better approximation of ξ than x_0, see Figure 1.

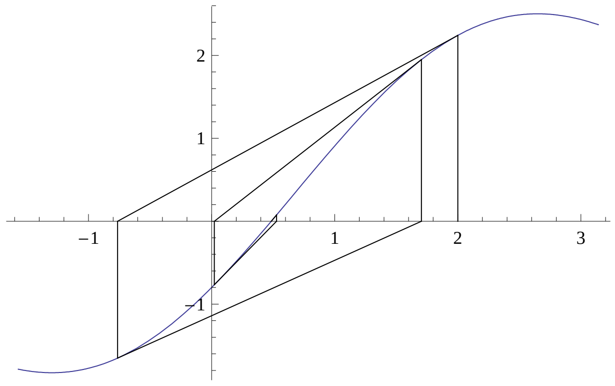

Figure 1: Graphical representation of the Newton-Raphson method starting at $x_0 = 2$.

Iterating this procedure yields the iteration scheme

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} , \tag{2}$$

which is called Newton's method or the Newton-Raphson method. It turns out that this scheme converges locally (i. e. if x_0 is near enough to ξ) towards ξ under very mild restrictions on f and the convergence is even quadratic under still very mild restrictions (see e. g. [11], Satz 10.10). That the convergence is quadratic, means that

$$|x_{n+1} - \xi| \leqq M\,|x_n - \xi|^2$$

for some constant $M \in \mathbb{R}$. Note that this inequality implies that the number of correct decimal digits roughly doubles in each iteration step.

The convergence of Newton's method is generically quadratic if f has a zero of order one. Assuming that $f(x) = (x - \zeta)\,\varphi(x)$, then one obtains (e. g. using a computer algebra system) that

$$x_{n+1} - \zeta = \frac{(x_n - \zeta)^2 \varphi'(x_n)}{\varphi(x_n) - \zeta\varphi'(x_n) + x_n\varphi'(x_n)}\ , \tag{3}$$

which shows the quadratic effect.

For $f(x) = x^2 - a$, Newton's method boils down to Heron's method

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

which is a more than two thousand years old quadratically convergent method for the iterative computation of the square root of $a > 0$. In this particular case the quadratic convergence is given by the formula

$$x_{n+1}^2 - a = \frac{\left(x_n^2 - a\right)^2}{4x_n^2}$$

which is easily established. Table 1 shows Heron's iteration for the approximation of $\sqrt{9}$.

2 Taylor Series of Implicit Functions

Now assume that $f : \mathbb{R}^2 \to \mathbb{R}$, and the function $x : \mathbb{R} \to \mathbb{R}$ is given implicitly by the equation

$$f(t, x(t)) = 0$$

n	x_n
0	5.000000000000000000000000000000000000000000000
1	3.400000000000000000000000000000000000000000000
2	3.023529411764705882352941176470588235294117647
3	3.000091554131380178530556191348134584573128862
4	3.000000001396983862248478425258881977121085389
5	3.000000000000000000325260651745651330219868255
6	3.000000000000000000000000000000000000017632415
7	3.000000000000000000000000000000000000000000000

Table 1: Heron's iteration gives a 45 digits approximation of $\sqrt{9}$ starting with $x_0 = 5$.

with one given point $x(0) = x_0$ such that $f(0, x_0) = 0$,[1] hence we search again for a zero x of f with initial value $x_0(t) = x_0$. Assume that $x(t)$ is often enough differentiable so that it has a Taylor polynomial approximation around the origin

$$x(t) = \sum_{k=0}^{N} a_k t^k + O(t^{N+1}) \tag{4}$$

or else $x(t) \in \mathbb{K}[[t]]$ for a suitable field $\mathbb{K}$. We can now use Newton's method again by setting

$$x_{n+1}(t) = x_n(t) - \frac{f(t, x_n(t))}{\frac{\partial f}{\partial x}(t, x_n(t))} + O(t^{2^n}) \tag{5}$$

with constant starting value

$$x_0(t) := x_0 \; .$$

Note that (5) is to be understood as an approximation in $\mathbb{K}[[t]]$, hence the fraction occurring there is converted to a Taylor polynomial of degree 2^n. Therefore the order of the Taylor approximation in the nth iteration step is 2^n, hence doubling the number of correct coefficients in each step. It turns out that this algorithm is correct [1, 6], i. e., the nth iteration step indeed generates 2^n correct initial coefficients. This is essentially shown by (3). Note that already Newton himself

[1]If $x(t_0) = x_0$, then one considers a Taylor expansion around $t = t_0$.

might have used this method to compute the power series of the inverse function, see e. g. [4], Chapter 8.

Note that a decimal approximation is also of type (4) with $t = 1/10$. The reason why the number of decimal places in the nth iteration step only doubles "approximately" is given by the fact that since $a_k \in \{0, \ldots, 9\}$ in decimal arithmetic carries occur. This is not the case for Taylor expansions, where $a_k \in \mathbb{K}$ for some field $\mathbb{K}$, e. g. $\mathbb{K} = \mathbb{R}$ or $\mathbb{K} = \mathbb{Q}$. Here the contributions for the coefficients of t^n are completely independent of the contributions to t^{n+1}. This is unfortunately not so for the decimal case. Therefore in this sense Newton's method is "simpler" in the Taylor case than in the decimal case.

We will give an example for this method in Section 4.

3 Higher Order Iteration Schemes

Next, we will develop a cubic scheme. Rather that linearizing f as we did for Newton's method, we can use a second-order approximation

$$f(x) \approx f(x_0) + f'(x_0)\,(x - x_0) + \frac{f''(x_0)}{2}\,(x - x_0)^2 \;, \tag{6}$$

hence the graph of f is now approximated by a fitting parabola at x_0. To linearize the computation again, we can approximate the quadratic term $(x - x_0)^2$ in (6) by $-(x - x_0)\,\frac{f(x_0)}{f'(x_0)}$, hence replacing one of these factors by Newton's formula (2). Therefore we solve the equation

$$0 = f(x_0) + f'(x_0)\,(x - x_0) - \frac{f''(x_0)}{2}\,\frac{f(x_0)}{f'(x_0)}\,(x - x_0) \;,$$

which gives

$$x_1 = x_0 - \frac{2\,f(x_0)\,f'(x_0)}{2\,f'(x_0)^2 - f''(x_0)\,f(x_0)} \;,$$

and iteratively

$$x_{n+1} = x_n - \frac{2\,f(x_n)\,f'(x_n)}{2\,f'(x_n)^2 - f''(x_n)\,f(x_n)} \;. \tag{7}$$

Note that this iteration formula is more complicated than Newton's (2). This method is attributed to Halley [8]. Its cubic convergence for $f(x) = (x - \zeta)\,\varphi(x)$

results from the computation

$$x_{n+1} - \zeta = (x_n - \zeta)^3 \left(\varphi(x_n)\varphi''(x_n) - 2\varphi'(x_n)^2\right) \Big/$$

$$\Big(- 2\varphi'(x_n)^2 x_n^2 + \varphi(x_n)\varphi''(x_n)x_n^2 + 4\zeta\varphi'(x_n)^2 x_n - 2\varphi(x_n)\varphi'(x_n)x_n$$

$$-2\zeta\varphi(x_n)\varphi''(x_n)x_n - 2\varphi(x_n)^2 - 2\zeta^2\varphi'(x_n)^2 + 2\zeta\varphi(x_n)\varphi'(x_n) + \zeta^2\varphi(x_n)\varphi''(x_n)\Big) \,.$$

For $f(x) = x^2 - a$, the method boils down to a method given by Dedekind [3]

$$x_{n+1} = \frac{x_n\,(x_n^2 - 3a)}{3x_n^2 + a}$$

which is a cubic iterative method for the computation of the square root of a. In this particular case the cubic convergence is given by the formula

$$x_{n+1}^2 - a = \frac{\left(x_n^2 - a\right)^3}{(3x_n^2 + a)^2} \,.$$

Table 2 shows Dedekind's iteration for the approximation of $\sqrt{9}$.

n	x_n
0	5.000000000000000000000000000000000000000000000
1	3.095238095238095238095238095238095238095238095
2	3.000022888270905574438379052654467218273995491
3	3.000000000000000333066907387546980616016966545
4	3.000000000000000000000000000000000000000000000

Table 2: Dedekind's iteration for a 45-digits approximation of $\sqrt{9}$ starting with $x_0 = 5$.

We can now iterate the above method to get algorithms of higher order. In the next step, we use a cubic approximation of f

$$f(x) \approx f(x_0) + f'(x_0)\,(x - x_0) + \frac{f''(x_0)}{2}\,(x - x_0)^2 + \frac{f'''(x_0)}{6}\,(x - x_0)^3 \,. \qquad (8)$$

To linearize the computation again, we can approximate the quadratic term $(x-x_0)^2$ in (8) by $-(x-x_0)\,\frac{2\,f(x_n)\,f'(x_n)}{2\,f'(x_n)^2-f''(x_n)\,f(x_n)}$, hence replacing one of these factors by the formula (7) of the last iteration, and similarly for the remaining cubic factor in (8). Solving for x leads to the iteration

$$x_{n+1} = x_n - \frac{3f(x_n)\left(2f'(x_n)^2 - f(x_n)f''(x_n)\right)^2}{2f'(x_n)} \cdot \tag{9}$$

$$\frac{1}{6f'(x_n)^4-9f(x_n)f''(x_n)f'(x_n)^2+f(x_n)^2f'''(x_n)f'(x_n)+3f(x_n)^2f''(x_n)^2}.$$

Note that this iteration formula is again much more complicated than the previous one (7). That this method is generically quartic convergent is shown in a similar way as before with a lengthy result of the form

$$x_{n+1} - \zeta = (x_n - \zeta)^4\, H(\varphi, x_n, \zeta)\ .$$

For a different fourth order scheme (Householder's method) see [18], [9], [17] and [21].

n	x_n
0	5.000000000000000000000000000000000000000000000
1	3.023529411764705882352941176470588235294117650
2	3.000000001396983862248478425258881977121085390
3	3.000000000000000000000000000000000000000001763241
4	3.000000000000000000000000000000000000000000000

Table 3: Fourth order iteration for a 45-digits approximation of $\sqrt{9}$ starting with $x_0 = 5$.

For $f(x) = x^2 - a$, the above method was considered in [13], and gives the iteration scheme

$$x_{n+1} = \frac{x_n^4 + 6a\,x_n^2 + a^2}{4x_n\,(x_n^2 + a)}$$

which is a quartic iterative method for the computation of the square root of a. In this particular case the quartic convergence is given by the formula

$$x_{n+1}^2 - a = \frac{(x_n^2 - a)^4}{16x_n^2(x_n^2 + a)^2}\ .$$

Table 3 shows this quartic iteration for the approximation of $\sqrt{9}$.

Iterating the above procedure yields algorithms of higher and higher order. The formulas connected with these algorithms get more and more involved though.[2] In the next section we will consider these algorithms for the computation of Taylor approximations.

4 Generating the Catalan Numbers using Newton's Method

The well-known Catalan numbers C_k are generated by the function

$$x(t) = \sum_{k=0}^{\infty} C_k \, t^k$$

satisfying the implicit equation

$$f(t,x) = t\,x^2 - x + 1 = 0 \tag{10}$$

with the initial value $x(0) = 1$. Of course, in this specific example one can easily solve the quadratic equation (10) and gets the explicit representation

$$x(t) = \frac{\sqrt{1-4\,t}-1}{2t}$$

of the generating function $x(t)$ from which one can deduce the series representation

$$x(t) = \frac{\sqrt{1-4\,t}-1}{2t} = \sum_{k=0}^{\infty} \frac{1}{k+1} \binom{2k}{k} t^k$$

(using the binomial series, or algorithmically, see e. g. [10]) so that the representation

$$C_k = \frac{1}{k+1} \binom{2k}{k} \tag{11}$$

for the Catalan numbers is valid.

Nevertheless, to demonstrate the method we would like to compute the Catalan sequence iteratively using the quartic method given in Section 3. To compute the

[2]The printout of the fifth order iteration $x_{n+1} = g(x_n)$ is already half a page long.

first 64 Catalan numbers using this method needs only $\log_4 64 = 3$ iteration steps. Below this computation is given using *Mathematica* [20].[3]

We define

```
In[1]:= n = 63
Out[1]= 63

In[2]:= x0 = 1
Out[2]= 1

In[3]:= f = t x^2 - x + 1
Out[3]= t x^2 - x + 1

In[4]:= F[z_] = Normal[(f/.x → z) + O[t]^(n+1)]
Out[4]= t x^2 - x + 1
```

and the quartic Newton term (9):

```
In[5]:= quartic :=
          Normal[(# - (3 F[#]
                         ((2 F′[#]^2-)
                            F[#] F″[#])^2)/
                     (2 F′[#]
                         (6 F′[#]^4-
                            9 F[#] F′[#]^2 F″[#]+
                            3 F[#]^2 F″[#]^2+
                            F[#]^2 F′[#]
                            F‴[#])))&[approx]]
```

With

```
In[6]:= approx = x0
Out[6]= 1
```

we can now start the first iteration:

```
In[7]:= k = 1
Out[7]= 1
```

We apply the quartic Newton term (9):

[3]For the computations in this paper, we used *Mathematica* 5.2 and *Maple* 11.

```
In[8]:= approx = quartic
```

Out[8]= $$1 - \frac{3\,t\,(2\,(2\,t-1)^2 - 2\,t^2)^2}{2\,(2\,t-1)\,(12\,t^4 - 18\,(2\,t-1)^2\,t^2 + 6\,(2\,t-1)^4)}$$

In the given case, this yields a rational function. In the first iteration step this function must be converted towards a Taylor polynomial of termination order 4^1:

```
In[9]:= approx = Normal[approx + O[t]^(4^k)]
```

Out[9]= $5\,t^3 + 2\,t^2 + t + 1$

The second iteration yields:

```
In[10]:= k = 2
Out[10]= 2
```

```
In[11]:= approx = quartic;
```

In this step, the termination order is 4^2:

```
In[12]:= approx = Normal[approx + O[t]^(4^k)]
```

Out[12]= $$9694845\,t^{15} + 2674440\,t^{14} + 742900\,t^{13} + 208012\,t^{12} + 58786\,t^{11} + 16796\,t^{10} + 4862\,t^9 + 1430\,t^8 + 429\,t^7 + 132\,t^6 + 42\,t^5 + 14\,t^4 + 5\,t^3 + 2\,t^2 + t + 1$$

Finally the third iteration gives the requested 64 correct starting coefficients:

```
In[13]:= k = 3
Out[13]= 3
```

```
In[14]:= approx = quartic;
```

```
In[15]:= approx = Normal[approx + O[t]^(4^k)]
```

Out[15]=
$$\begin{aligned}
&942958505587719797879353849463801 25\,t^{63}+\\
&24139737743045626825711458546273312\,t^{62}+\\
&6182127958584855650487080847216336\,t^{61}+\\
&1583850964596120042686772779038896\,t^{60}+\\
&405944995127576985730643443367112\,t^{59}+\\
&104088460289122304033498318812080\,t^{58}+\\
&26700952856774851904245220912664\,t^{57}+\\
&6852456927844873497549658464312\,t^{56}+\\
&1759414616608818870992479875972\,t^{55}+\\
&451959718027953471447609509424\,t^{54}+
\end{aligned}$$

$116157871455782434250553845880\,t^{53}+$
$29869166945772625950142417512\,t^{52}+$
$7684785670514316385230816156\,t^{51}+$
$1978261657756160653623774456\,t^{50}+$
$509552245179617138054608572\,t^{49}+$
$131327898242169365477991900\,t^{48}+$
$33868773757191046886429490\,t^{47}+$
$8740328711533173390046320\,t^{46}+$
$2257117854077248073253720\,t^{45}+$
$583300119592996693088040\,t^{44}+$
$150853479205085351660700\,t^{43}+$
$39044429911904443959240\,t^{42}+$
$10113918591637898134020\,t^{41}+$
$2622127042276492108820\,t^{40}+$
$680425371729975800390\,t^{39}+$
$176733862787006701400\,t^{38}+$
$45950804324621742364\,t^{37}+$
$11959798385860453492\,t^{36}+$
$3116285494907301262\,t^{35}+$
$812944042149730764\,t^{34}+$
$212336130412243110\,t^{33}+55534064877048198\,t^{32}+$
$14544636039226909\,t^{31}+3814986502092304\,t^{30}+$
$1002242216651368\,t^{29}+263747951750360\,t^{28}+$
$69533550916004\,t^{27}+18367353072152\,t^{26}+$
$4861946401452\,t^{25}+1289904147324\,t^{24}+$
$343059613650\,t^{23}+91482563640\,t^{22}+$
$24466267020\,t^{21}+6564120420\,t^{20}+$
$1767263190\,t^{19}+477638700\,t^{18}+129644790\,t^{17}+$
$35357670\,t^{16}+9694845\,t^{15}+2674440\,t^{14}+$
$742900\,t^{13}+208012\,t^{12}+58786\,t^{11}+$
$16796\,t^{10}+4862\,t^{9}+1430\,t^{8}+429\,t^{7}+$
$132\,t^{6}+42\,t^{5}+14\,t^{4}+5\,t^{3}+2\,t^{2}+t+1$

Note, however, that to compute the first $1024 = 4^5 = 2^{10}$ Catalan numbers using five iterations of this quartic scheme is more time consuming than 10 iterations of the "normal" Newton method. This is evident since the iteration step given by formula (9) needs more than the double amount of computations as iteration step (2). Unfortunately, none of the regarded higher order methods is therefore symbolically more efficient than the original version. Nevertheless, in our opinion, the designed higher order algorithms are of theoretical interest. With regard to computations with decimal arithmetic, if the underlying formulas are compiled, the higher order schemes should be more efficient. Details about timings with *Mathematica* [20] and *Maple* [16] can be found in Tables 4–5.[4]

	Newton (2)	quartic (9)	formula (11)	hypergeom.
$n = 2^{10}$	2.16	4.0	0.14	0.03
$n = 2^{12}$	217	438	6.5	0.06

Table 4: Several methods to compute the first n Catalan numbers using *Mathematica*

	Newton (2)	quartic (9)	formula (11)	hypergeom.
$n = 2^{10}$	3.83	19.0	2.47	0.078
$n = 2^{12}$	$\diamond$	$\diamond$	71.6	1.74

Table 5: Several methods to compute the first n Catalan numbers using *Maple*

It is interesting to note that in *Maple* the "closed formula" (11) is not much more efficient than the Newton iteration! The reason for this failure is that the computation of the binomial coefficients is not efficient enough. This can be resolved by a "hypergeometric computation" (see [12]) of the truncated series $\sum_{k=0}^{n} C_k\, t^k$ using, e. g., the code

```
catalansum:=proc(n)
local tmp,summ,k;
  tmp:=1; summ:=1;
  for k from 1 to n do
```

[4]All timings are in seconds and were done with Mathematica 5.2 / Maple 11 and a laptop with Intel Core Duo T 2600, 2.16 GHz CPU and 2 GB RAM. The sign $\diamond$ indicates that the computation took longer than one hour.

```
    tmp:=2*(2*k-1)/(k+1)*tmp;
    summ:=summ+tmp*t^k
    end do;
summ;
end proc:
```

This computation uses exclusively rational arithmetic, hence no factorials and only simple gcd computations are necessary.

This gives the last column of Table 5. A similar computation in *Mathematica* using `Apply` gives the last column in Table 4.

Nevertheless, if no "closed form" is available – and under certain circumstances even with closed form – then Newton's method is a very efficient method to compute the Taylor coefficients of an implicitly given function. This will be shown in the next section.

5 Further Examples

If we write the steps of Newton's method as a complete *Mathematica* procedure, we get [15]

```
In[16]:= FastImplicitTaylor[f_, x_[t_], x0_, n_] :=
             Module[{F, z, approx},
                 F[z_] :=
                   Normal[(f/.x → z) + O[t]^(n + 1)];
                 approx = x0;
                 Do[
                   approx =
                     Normal[# - F[#]/F′[#]&[approx]
                         +O[t]^(2^k)],
                   {k, 1, Log[2, n + 1]}];
                 approx + O[t]^(n + 1)
               ]/; IntegerQ[Log[2, n + 1]]
```

where—for simplicity—we assume that $n+1$ is a power of 2.[5] The corresponding *Maple* code used for the Newton scheme is given as

[5]Similary in the quartic scheme we assume that $n+1$ is a power of 4, e. g.

```
FastImplicitTaylor:=proc(f,x,t,x0,n)
local G,i,approx;
if not(type(simplify(log[2](n+1)),integer))
then return 'procname(args)'
end if;
G:=z->normal(subs(x=z,x-f/diff(f,x)));
approx:=x0;
for i from 1 to simplify(log[2](n+1)) do
approx:=convert(series(G(approx),t=0,2^i),polynom);
end do;
convert(series(approx,t=0,n+1),polynom);
end proc:
```

In this section, we would like to give some more examples for its use. We are interested particularly in examples for which this method is *faster* than the direct computation of the Taylor coefficients of an *explicitly given function.* Although this seems to be a rather venturous wish, it nevertheless can happen. To compute the Taylor coefficients of an explicitly given function Taylor approximations for all subexpressions are computed and combined. Hence Newton's method will be most efficient if the subexpressions of the implicit representation have easier computable Taylor approximations than the subexpressions for the given explicit expression. We will show that for this reason under certain circumstances Newton's method can be faster.

As an example of this type we compute the Taylor polynomial for

$$x(t) = \tan(\sin t) = t + \frac{t^3}{6} - \frac{t^5}{40} + \cdots$$

up to order 2^8. We rewrite the equation

$$x = \tan(\sin t) \tag{12}$$

implicitly as

$$\arctan x - \sin t = 0 \tag{13}$$

and using Newton's method we compute therefore

```
In[17]:= Timing[
            ser1 = FastImplicitTaylor[
               ArcTan[x] - Sin[t], x[t], 0,
               2^8 - 1]; ]
Out[17]= {1.875 Second, Null}
```

which takes about two seconds. However, the direct computation, using (12)

```
In[18]:= Timing[
           ser2 = Series[Tan[Sin[t]],
              {t, 0, 2^8 - 1}]; ]
Out[18]= {106.36 Second, Null}
```

which – of course – gives the same result

```
In[19]:= ser1 - ser2
Out[19]= O(t^256)
```

has a much longer computation time. The higher the order, the larger the gap between these two methods. The reason is the complicated Taylor representation (see e. g. [19], p. 472)

$$\tan t = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{2^{2k}(2^{2k}-1)}{(2k)!} B_{2k}\, t^{2k-1} \tag{14}$$

which (at least asymptotically) cannot be computed very efficiently since the Bernoulli numbers B_k are difficult to compute [5]. On the other hand, all Taylor approximations of the implicit equation (13), namely $\arctan x$ and $\sin t$ have hypergeometric representations

$$\arctan x = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} x^{2k+1} ,$$

$$\sin t = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} t^{2k+1} ,$$

and can be computed very efficiently. For the tangent function such a simple representation does not exist (see e. g. [15], Satz 10.11). Even worse is the situation for the example $x(t) = \tan(e^t - 1)$, see Table 6.

Note, however, that in *Maple* the situation is different since *Maple* computes these explicit Taylor polynomials much faster, see Table 6. Whereas *Maple*'s `series` command needs 1.8 seconds to compute the Taylor polynomial of $\tan t$ of order 2^{10} using probably (14), *Mathematica* needs 87 seconds for the same purpose. This proves that *Mathematica* funnily seems not to use (14) for the computation of the tangent series. But even in *Maple*, nevertheless, for $x = \tan(\sin t)$ asymptotically the direct method should be slower than Newton. However, this cannot be tested since already for $n = 2^{10}$ *Maple* the Newton computation is out of memory.

However, there are also examples for which *Maple*'s `series` command is very weak, and Newton wins the game. This is so in particular if algebraic functions occur. We have considered the examples $x(t)^3 - 1 - \ln(1+t) = 0$ and $(1-t^3)\,x(t)^2 - (1+t^2) = 0$ which show this effect, see Table 6.

$x(t)$	$\tan(\sin t)$	Newton	$\tan(e^t - 1)$	Newton
Maple	1.0	3.9	2.1	7.2
Mathematica	106	1.9	525	6.9
$x(t)$	$\sqrt[3]{1+\ln(1+t)}$	Newton	$\sqrt{\frac{1+t^2}{1-t^3}}$	Newton
Maple	54	10.3	392	11.0
Mathematica	1.0	5.0	2.4	11.7

Table 6: Explicit and implicit computation of Taylor approximations of order 2^8 (last example order 2^{10}).

Whereas, in the previous examples we considered composite functions, in some instances, even the computation of the Taylor polynomial of a "primitive" function via Newton's method may be faster than the "direct" computation. In *Mathematica*, this is so, for example, for the inverse error function `InverseErf`, the inverse of the error function

$$\operatorname{erf}(t) = \frac{2}{\sqrt{\pi}} \int_0^t e^{-x^2}\,dx$$

as is shown by the computations

```
In[20]:= Timing[
           ser1 = FastImplicitTaylor[Erf[x] - t,
              x[t], 0, 2^7 - 1];]
Out[20]= {13.953 Second, Null}

In[21]:= Timing[
           ser2 = Series[InverseErf[t],
              {t, 0, 2^7 - 1}];]
Out[21]= {21.515 Second, Null}

In[22]:= ser1 - ser2
Out[22]= O(t^128)
```

In this research I realized that computer algebra systems like *Mathematica* and *Maple* seem to use quite different methods to compute Taylor polynomials of "primitive" expressions that represent $\mathbb{Q}[[x]]$ series. As seen above, well-known formulas are sometimes just ignored so that the computation times are indisputable. Table 7 gives some more examples showing this effect.[6]

$x(t)$	$\sin t$	$\arcsin t$	$\tan t$	$\arctan t$	$\cot t$	$\operatorname{arccot} t$
Maple	0.2	0.2	1.8	2.2	2.2	7.2
Mathematica	0.02	0.02	87	0.05	126	◇
$x(t)$	$\operatorname{erf}(t)$	$\operatorname{erf}^{-1}(t)$	$t\,e^t$	$W(t)$		
Maple	0.7	47.4	0.0	0.4		
Mathematica	0.05	◇	0.01	0.4		

Table 7: Computation of Taylor approximations of order 2^{10} for some inverse functions.

Of course, $\cot t$ is not a $\mathbb{Q}[[x]]$ series but has a pole of order one. This, however, is not essential since there is a formula similar to (14) (see e. g. [19], p. 472)

$$\cot t = \frac{1}{t} + \sum_{k=1}^{\infty} (-1)^k \frac{2^{2k}}{(2k)!} B_{2k}\, t^{2k-1} \; .$$

Mathematica's problem with $\operatorname{arccot} t$ is the branch cut of the principal value at the origin. This leads for example to the output

```
In[23]:= Series[ArcCot[t], {t, 0, 5}]
```

$$\texttt{Out[23]=} \left(-t + \frac{t^3}{3} - \frac{t^5}{5} + O(t^6) \right) + \frac{1}{2} (-1)^{\lfloor \frac{2\operatorname{Arg}(t)+\pi}{2\pi} \rfloor} \pi$$

Therefore, the computation times for $\arctan t$ and for $\operatorname{arccot} t$ are quite different with *Mathematica* although for positive t-values the relation

$$\arctan t + \operatorname{arccot} t = \frac{\pi}{2}$$

is valid. *Maple* decides instead for the specific branch given by $\operatorname{arccot} 0 = \frac{\pi}{2}$. Therefore the computation times are much better. However, to give a fair comparison between the timings of *Maple* and *Mathematica* we should note that *Maple*'s

[6]Note that the inverse error function $\operatorname{erf}^{-1}(t)$ is denoted by `RootOf(-erf(_Z)+t)` in *Maple*.

`series` command does not ensure the requested order. It just computes the Taylor polynomials of the parts of the input expression and combines them. Therefore as result of the command

```
series(sin(x^10)/x^10,x=0,21);
```

we get for example

$$1 + O(x^{11})\ .$$

Whereas the requested order is 20, the output has order 10. Similarly `series(tan(sin(t^5))/sin(tan(t^5)),t=0,15);` has order $O(t^{10})$. On the other hand *Mathematica* uses an adaptive approach producing the correct result

```
In[24]:= Series[Sin[x^10]/x^10, {x, 0, 20}]
Out[24]= 1 - x^20/6 + O(x^21)
```

$$\texttt{In[24]:=}\ \mathrm{Series}\left[\frac{\mathrm{Sin}[x^{10}]}{x^{10}}, \{x, 0, 20\}\right]$$

$$\texttt{Out[24]=}\ 1 - \frac{x^{20}}{6} + \mathrm{O}\left(x^{21}\right)$$

Of course *Mathematica*'s method is inherently more time consuming. Whereas *Maple*'s method is easy to implement, but does not generally give the requested answer, *Mathematica*'s implementation ensures always the required order.

Some other special functions are better implemented in *Mathematica*. The computation of the Taylor polynomial of the inverse function of $x = t\,e^t$, the so-called Lambert W function[7], is quite efficient as can be also seen from Table 7. Note that for this example obviously the formula (see e. g. [2])

$$W(t) = \sum_{k=1}^{\infty} (-1)^{k-1} \frac{k^{k-1}}{k!}\, t^k$$

is utilized in both systems. Therefore, the explicit form is of course more efficient than Newton's implicit method. However, by changing the "primitive function" slightly and considering the implicit equation $x\,e^{x^2} - t = 0$ and therefore

$$x(t) = \frac{\sqrt{2}\,t}{2\sqrt{\frac{t^2}{W(2t^2)}}}\ ,$$

Newton's method gains again and wins in *Maple*, see the last entry of Table 8.

[7]named `LambertW` in *Maple* and `ProductLog` in *Mathematica*

$x(t)$	$W(t)$	Newton	$\frac{\sqrt{2}t}{2\sqrt{\frac{t^2}{W(2t^2)}}}$	Newton
Maple	0.02	4.5	5.3	0.7
Mathematica	0.02	6.5	0.9	1.2

Table 8: Explicit and implicit computation of Taylor approximations of order 2^8.

Conclusion

In this paper we showed how Newton's method can be extended to higher order schemes for the computation of truncated power series such that the number of correct coefficients doubles, triples etc. in each iteration step.

Then we indicated how Newton's method can be used for the computation of truncated power series in *Maple* and in *Mathematica*, systems that have collected know-how and experience for 28 and 20 years, respectively. Nevertheless as a result it turns out that in many instances the implicit use of Newton's method is faster than the built-in direct computations.

Acknowledgment

I would like to thank Peter Larcombe who was interested in the iterative computation of the Catalan numbers which initiated the current research.

References

[1] Brent, P. P., and Kung, H. T.: Fast algorithms for manipulating formal power series. Journal of the ACM **178**, 1978, 581–595.

[2] Corless, R. M., Gonnet, G. H., Hare, D. E. G., Jeffrey, D. J., and Knuth, D, E.: On the Lambert W function. Advances in Computational Mathematics **5**, 1996, 329–359.

[3] Dedekind, R.: *Stetigkeit und irrationale Zahlen.* Vieweg, Braunschweig, 1965.

[4] Edwards, C. H.: *The Historical Development of the Calculus.* Springer, New York, Berlin, 1979.

[5] Fee, G. and Plouffe, S.: An efficient algorithm for the computation of Bernoulli numbers. arXiv:math/0702300v2 [math.NT]

[6] von zur Gathen, J., Gerhard, J.: *Modern Computer Algebra.* Cambridge University Press, Cambridge, 1999.

[7] Graham, R. L., Knuth, D. E. and Patashnik, O.: *Concrete Mathematics. A Foundation for Computer Science.* Addison-Wesley, Reading, Massachussets, second edition, 1994.

[8] Halley, E.: A new, exact, and easy method of finding roots of any equations generally, and that without any previous reduction (Latin). Philos. Trans. Roy. Soc. London **18**, 1694, 136–148. English translation: Philos. Trans. Roy. Soc. London **3**, 1809, 640–649.

[9] Kalantaria, B., Kalantari, I. and Zaare-Nahandici, R.: A basic family of iteration functions for polynomial root finding and its characterizations. J. Comp. Appl. Math. **80**, 1997, 209–226.

[10] Koepf, W.: Power series in computer algebra. J. Symbolic Computation **13**, 1992, 581–603.

[11] Koepf, W.: *Mathematik mit DERIVE.* Vieweg, Braunschweig/Wiesbaden, 1993.

[12] Koepf, W.: Efficient computation of Chebyshev polynomials. In: M. Wester (Ed.): *Computer Algebra Systems: A Practical Guide.* John Wiley, Chichester, 1999, 79–99.

[13] Koepf, W.: *DERIVE für den Mathematikunterricht.* Vieweg, Braunschweig/Wiesbaden, 1998.

[14] Koepf, W. and Schmersau, D.: *Die reellen Zahlen als Fundament und Baustein der Analysis.* Oldenbourg, Munich, 2000.

[15] Koepf, W.: *Computeralgebra.* Springer, Berlin–Heidelberg–New York, 2006.

[16] Monagan, M. B., Geddes, K. O., Heal, K. M., Labahn, G., Vorkoetter, S. M., McCarron, J., DeMarco, P.: Maple 9: *Advanced Programming Guide.* Maplesoft, Waterloo, 2003.

[17] Sebah, P, Gourdon, X.: Newton's method and high order iterations, `http://numbers.computation.free.fr/Constants/Algorithms/newton.html`, 2001.

[18] Snyder, R. W.: One more correction formula. Amer. Math. Monthly **62**, 1955, 722–725.

[19] Stöcker, H.: *Taschenbuch mathematischer Formeln und moderner Verfahren.* Harri Deutsch, Frankfurt, 4. Auflage, 1999.

[20] Wolfram, St.: *The Mathematica Book.* Wolfram Media und Cambridge University Press. Fourth Edition, Cambridge, 1999.

[21] Wikipedia: Householders's Method. `http://en.wikipedia.org/wiki/Householder's_method`, 2007.

[22] Ypma, T. Y.: Historical development of the Newton-Raphson method. SIAM Review **37**, 1995, 531–551.