Series Solutions of Algebraic and Differential Equations: A Comparison of Linear and Quadratic Algebraic Convergence

Richard J. Fateman
University of California, Berkeley

Abstract

Speed of convergence of Newton-like iterations in an algebraic domain can be affected heavily by the increasing cost of each step, so much so that a quadratically convergent algorithm with complex steps may be comparable to a slower one with simple steps. This note gives two examples: solving algebraic and first-order ordinary differential equations using the MACSYMA algebraic manipulation system, demonstrating this phenomenon. The relevant programs are exhibited in the hope that they might give rise to more widespread application of these techniques.

1 Newton Iteration in a Power Series Domain

Newton iteration is a powerful tool for developing approximations to solutions of various types of equations. Recently, the case of iteration in a power series domain has been studied in some detail as a notion relevant especially in the field of computer-aided algebraic computation.

It has been shown by Kung and Traub [6] that there are fast procedures for finding Taylor-series type expansions for any algebraic function. Based on a reduction of an arbitrary algebraic problem to a "regular" problem, their paper then uses a Newton iteration to solve the problem "fast." The theorem below provides a constructive procedure for computing the expansion of a regular problem. In the next section we describe how to extend this type of iteration to first order differential equations.

0

One of our interests here is to see if the asymptotically "fast" algorithms are, in practice, faster than more naive ones, especially on simple inputs. A second motivation for this paper is to provide simple implementations of linearly and quadratically convergent iterations in a popular algebraic manipulation system (MACSYMA). The theorems we state have been derived and proved elsewhere, and are quoted for reference.

We quote from a reference ([7]),

Theorem 1 *Let $f(x)$ be a polynomial with coefficients in a power series domain $\mathbf{D} = \mathbf{F}[[t]]$. By this we denote the domain of truncated power series in t with coefficients in $\mathbf{F}$. Let α in $\mathbf{F}$ be an $\mathrm{O}(t)$ approximation to a root of $f(x)$. (i.e. $x = \alpha$ is a solution to $f(x) = 0$ when $t = 0$). Furthermore, suppose that α satisfies $f'(\alpha) \neq 0$ when $t = 0$ and where the prime indicates differentiation with respect to x.*

Then the sequence of iterates $x_0, x_1 \cdots$ defined by

$$\begin{aligned} x_0 &:= \alpha \\ x_k &:= \left(x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}\right) \bmod t^{2^k} \qquad k > 0 \end{aligned}$$

has the property that x_k is an $\mathrm{O}(t^{2^k})$ approximation to x.

The next theorem provides what is, at first glance, an inferior procedure.

Theorem 2 *Under the same hypotheses as theorem 1, the sequence of iterates*

$$\begin{aligned} x_0 &:= a \\ x_k &:= \left(x_{k-1} - \frac{f(x_{k-1})}{f'(x_0)}\right) \bmod t^{k+1} \qquad k > 0 \end{aligned}$$

has the property that x_k is an $\mathrm{O}(t^{k+1})$ approximation to x.

That is, the iteration of theorem 1, the Newton iteration, converges quadratically; the iteration of theorem 2, which we will refer to as the Hensel iteration, converges linearly. Note, however, that the Hensel iteration does not require recomputation of f'. Thus each of the (far more numerous) steps is less expensive.

There have been several papers analyzing the choice between linear and quadratic convergence of Hensel, Newton, (also referred to as Zassenhaus) successive approximations. These discussions have involved polynomial greatest-common-divisor and factoring algorithms. This paper provides another related area in which it pays to compare these two iterations. To jump ahead, the quadratically convergent algorithm is only marginally faster than the linear one, and is sometime slower.

The iteration is actually more general than we have indicated. In fact, $f(x)$ need not be a polynomial in x but can include *any* function which can be expanded as a power series in x, as long as the required initial point can be found.

2 Differential Equations

We can solve differential equations with a similar iteration. Consider solving the equation $g(x', x, t) := x' - f(x, t) = 0$. We can expand g in a two-dimensional Taylor series in the variables x' and x at the "point" (x_k, x'_k):

$$\begin{aligned}0 &= g(x', x, t)\\ &= x' - f(x, t)\\ &= x'_k - f(x_k, t) + (x' - x'_k) + (x - x_k)(-f_x(x_k, t) + \cdots\end{aligned}$$

Solving for $e_k = x - x_k$ provides a basis for a Newton iteration for the solution of differential equations. Geddes [5] proceeds in a somewhat more general fashion, but we abstract the simpler

Theorem 3 *Let the initial-value problem*

$$x' - f(x) = 0; \quad x(0) = \alpha$$

be such that $f(x)$ is a polynomial in the domain $\mathbf{D}[x]$ with coefficients in a power series domain $\mathbf{D} = \mathbf{F}[[t]]$ and where $\alpha \in \mathbf{F}$. Then the sequence of iterates $x_0 := \alpha, \quad x_1, \ldots, x_{k+1} := x_k + e_k$, converges to a formal power series solution to $x' = f(x)$ where $e_k \in \mathbf{F}[[t]]$ is the solution of the linear ordinary differential equation

$$e'_k - f'(x_k)e_k = -x'_k + f(x_k) \mod (t^{2^k}) \qquad e_k(0) = 0.$$

Furthermore, x_k is an $\mathrm{O}(t^{2^k})$ approximation to x.

Note that the first-order linear differential equation

$$y' + g(t)y = h(t)$$

satisfying $y(0) = \alpha$ can be solved by the integrating factor method. Let

$$\mu(t) = e^{\int_0^t g(\tau)d\tau}.$$

Then

$$y(t) = \frac{1}{\mu(t)}\left(\int_0^t \mu(\tau)h(\tau)d\tau\right).$$

These computations can be done in a power series domain. Details in the form of a program are given in the next section.

A much simpler but only linearly convergent iteration is described in the next theorem, based on Picard iteration, as discussed in any elementary differential equations text.

Theorem 4 *Under the same hypotheses as theorem 3, the sequence of iterates, $x_0 := \alpha, x_1 \ldots, x_{k+1} := \int f(x_k)dt \mod t^{k+2}$ converges to a formal power series solution to $x' = f(x)$. Furthermore, x_k is an $O(t^{k+1})$ approximation to x.*

Again, details in the form of a quite simple program, are given in the next section.

3 Programs

These programs are perhaps less than self-explanatory, but the basic notational conventions used in the current MACSYMA top-level language resemble Algol-60.

Nearly all statements are treated as expressions and thus have values. The value of an assignment statement is its right-hand-side. The functions used and defined below evaluate all arguments if they have values, otherwise use their arguments literally.

Procedures are defined by :=, and local variables are declared within square brackets inside a procedure `block`. The assignment operator is a colon (:). The command `subst(a=b,e)` substitutes `b` for `a` in `e`. The parallel substitution of several values is done by `subst([a=b,c=d],e)`. The command `taylor(a,b,c,d)` expands `a` in a series in the variable `b` about the point `c` up to degree `d`. For example, `taylor(sin(x),x,0,3)` produces the expression $x - x^3/6 + \cdots$.

The command `horner` converts an expression to a "Horner's Rule" arrangement (e.g. `horner(x^2-x+1)` $= x(x-1)+1$). Since we are using substitution as a simple implementation of composition of Taylor series, it seems appropriate to make this chore somewhat less expensive by using `horner`. An asymptotically faster technique is discussed in reference 3. Undoubtedly some significant factor could be obtained by coding this more closely to the representation.

The symbol # is used for "$\neq$" and comments are delimited by /* and */. Finally, we found it convenient, and a particular time-saver, to write a short program to tinker with the Macsyma system data-type used to represent Taylor series. At critical places in the program we must convert the truncation level of a series to another (higher) precision. This is done by a 3-line Lisp program (not shown in the listings) which we called `tweek`. The command `tweek(s,n)` changes the Taylor series `s` to precision `n`, which is presumed to be higher than its current precision. If `s` is already a Taylor series in `t` about 0, then `tweek(s,n)` is equivalent to `taylor(s,t,0,n)`, but takes essentially no time. It modifies an existing data structure rather than constructing a new one. The use of `tweek` on some simple examples provided a factor of 3 speedup. Timings must be dealt with carefully: although we have tried to use the simplest correct programs, the underlying complexity of algebraic manipulation systems suggest that another factor of 3 could easily be hidden in some redundant computation. Such discoveries have altered our recommendation of the choice of algorithm twice during the research reported on here.

3.1 Algebraic and Transcendental Root-finders

```
/* Newtonroot and Henselroot each finds x(t)
such that f(x)=0 mod (t^(n+1)).
Aroot, (given) must satisfy f(aroot)=0
and f'(aroot)#0 mod (t).
Newtonroot uses a quadratically convergent
iteration, Hensel uses a linearly convergent
iteration.  */

/* --------------NEWTONROOT---------------*/

newtonroot(f,x,t,aroot,n):=block([s,deg,ex,df],

/* check initial conditions of theorem */

if (subst([x=aroot,t=0],f)#0) or
   (subst([x=aroot,t=0],(df:diff(f,x)))=0)
      then return(
         print("can't expand", f, "at" , aroot)),

      /*set up iteration */

ex:horner(x-f/df),
/* note: ex will be recomputed in the loop*/
s:aroot, deg:0,
while deg < n d do  /*s is a mod(deg) approx */
    (deg:min(2*(deg+1)-1,n),
    /* promote s to higher approx. */
     s:taylor(subst(x=tweek(s,deg),ex),t,0,deg)),
return(s));

/* --------------HENSELROOT---------------*/
henselroot(f,x,t,aroot,n):= block([s,deg,ex,df],

/* check initial conditions of theorem */

if (subst([x=aroot,t=0],f)#0) or
   (df:subst([x=aroot,t=0],diff(f,x)))=0
      then return(
       print("can't expand", f, "at ",aroot)),

ex:horner(x-f/subst(aroot,x,df)),
  /* note: ex is fixed*/
s:aroot, deg:0,
while deg < n do
   (deg: deg+1,
    s: taylor(subst(x=tweek(s,deg),ex),t,0,deg)),
return(s));
```

Here's a sample problem: $f(x) = x^2+3x+2+t$. For the Newton iteration, we use the following formula for successive approximations `s` in the program above:

$$s := \frac{s^2-t-2}{2s+3}$$

Realizing that these iterations are done in a power-series domain, they work out, to degree 7, as follows (this is a trace produced by Macsyma):

$$s \text{ set to } -1$$

$$s \text{ set to } -1-t+\cdots$$

$$s \text{ set to } -1-t-t^2-2t^3+\cdots$$

$$s \text{ set to }$$
$$-1-t-t^2-2t^3-5t^4-14t^5-42t^6-132t^7+\cdots$$

For the Hensel iteration, linear convergence is the rule, so that to produce the degree 7 result, 7 iterations are used instead of 3. Each iteration tacks on one more term. However, the iteration is the much simpler formula

$$s := (-s-2)s-t-2$$

The table below gives the times taken by MACSYMA to use each of the algebraic programs to solve the equation $f(x) = x^2+3x+2+t$ for x as a power series in t about $t=-1$ The command, for degree `n` looks like

```
newtonroot(x^2+3*x+2+t,x,t,-1,n).
```

Degree	Time in Seconds	
	Newton	Hensel
2	0.20	0.17
3	0.22	0.22
4	0.38	0.32
5	0.40	0.40
6	0.42	0.53
7	0.47	0.65
8	0.78	0.80
9	0.87	0.97
10	0.88	1.12
11	0.90	1.33
12	0.95	1.53
13	0.98	1.80
14	1.03	2.02
15	1.07	2.32
16	1.98	2.73
17	2.00	2.88

Note that the Newton iteration is generally *superior* to Hensel, although not by as much as the quadratic convergence might lead you to think. Also note that as the number of terms increases beyond a power-of-two degree, the cost of the Newton iteration nearly doubles. (e.g. note the difference between degree 7 and 8, or 15 and 16).

The times given above are for a VAX 8600 computer running Macsyma on the UNIX 4.3BSD operating system, Franz Lisp Opus 42. Lisp garbage collection times have been excluded from the timings.

3.2 First Order ODE solvers

```
/* Each of the two programs below solve
      x'=f(t,x), x(0)=a0 */

/*---NEWTONFODE (first order ODE) ---*/


newtonfode(f,x,t,a0,n):=block([s,deg,hh,uu],

deg:0, s:a0,
while deg < n do
 /*s is a mod(t^deg) approx  to x. */
 (deg:min(2*deg+1,n),
  s:tweek(s,deg),
  hh:-diff(s,t)+taylor(subst(x=s,f),t, 0, deg),
  uu:taylor(%e^integrate(
   taylor(
      subst(x=s,-diff(f,x)),t,0,deg),
        t),t,0,deg),
  s:s+1/uu*(integrate(uu*hh,t))),
 return(s));

 /*--PICARD (linearly convergent) ---*/
```

Degree	Time in Seconds	
	Newton	Hensel
2	0.47	0.17
3	0.52	0.27
4	0.92	0.40
5	1.00	0.58
6	1.05	0.78
7	1.13	1.03
8	1.87	1.30
9	1.97	1.60
10	2.03	1.90
11	2.13	2.30
12	2.18	2.65
13	2.28	3.10
14	2.37	3.53
15	2.48	4.03
16	4.02	4.60
17	4.23	5.20

```
picard(f,x,t,a0,n):=
block([s,deg],
 deg:0,s:a0,
 while deg < n do
 (s:tweek(s,deg:deg+1),
  s:integrate(
      taylor(subst(x=s,f),t,0,deg),t)+a0,
 return(taylor(s,t,0,n)));
```

The next table provides some comparison times for solving first order differential equations. Here we are solving a Ricatti differential equation

$$x' = 1 + t^2 - 2xt + x^2, \quad x(0) = 1$$

whose solution is $x = t + 1/(1-t)$, or in series,

$$1 + 2t + t^2 + t^3 + t^4 + \cdots + t^k + \cdots.$$

Note that the linear Picard iteration is often faster, although quadratic convergence catches up eventually.

The Newton iteration again increases in cost sharply across the power-of-two degree transition points, but eventually becomes less expensive than Picard iteration.

The times listed above are for the same VAX 8600 computer system.

4 Problems and Further Techniques

In an earlier paper [4], we ignored the details of overcoming some problems in using iterative techniques for algebraic problems, referring the reader to Kung and Traub [6]. Let us return to an example similar to one in [4] which, as we pointed out, was not amenable to Newton-iteration solution because it did not satisfy the two conditions on f and f':

$$f(x) = x^3 + t^3(x+1) = 0.$$

The equation $(f(x) \bmod t) = 0$ has the solution $x = 0$. But $f'(x) \bmod t \equiv 3x^2 = 0$ when $x = 0$. Yet a power series type solution for this equation exists, (in fact, three!) one of which is

$$x = t - \frac{t^2}{3} + \frac{t^4}{81} + \cdots$$

The essential step in solving this problem is to use the Newton Polygon procedure [6] to convert this problem to

$$\bar{x}^3 + t\bar{x} + 1 = 0.$$

which has a solution

$$\bar{x} = -1 + \frac{t}{3} - \frac{t^3}{81} - \frac{t^4}{243} + \frac{4t^6}{6561} + \frac{5t^7}{19683} + \cdots.$$

This solution is related to the original one by

$$x = -t\bar{x}.$$

Also available routinely are the other two solutions which can be described using the symbol ω for either of the two primitive roots of $\omega^3 + 1$. (The third root is $\omega = -1$.) These solutions are:

$$x = \omega + \frac{(\omega-1)t}{3} + \frac{\omega t^3}{81} - \frac{(\omega-1)t^4}{243} - \frac{4\omega t^6}{6561} + \frac{(5\omega-5)t^7}{19683}.$$

Unfortunately, the Newton Polygon procedure is rather complex and has one component which is normally expressed graphically! Except in simple somewhat contrived cases, it generally requires algebraic number computation. A proprietary version of this algorithm has been coded and included in the Symbolics Corp. version of MACSYMA, in the command "Taylorsolve" [3].

Kung and Traub in [6] also settle other issues of relevance by dealing with multivalued expansions, expansions at points other than the origin, and degenerate cases where $f(x)$ is not irreducible.

A complete resolution of all the special checks by means of a computer program appears to be, with the exception of algebraic number computation, practical and desirable.

5 Notes on Tools and Timings

As far as the tools in MACSYMA or similar programs for handling series, it appears that systems should provide appropriate access to the data structures on power series for the operations of change of order, composition, differentiation, and integration. Efficient versions of these operations are *not* conveniently available in MACSYMA. For change of order we wrote a special program called "tweek." For composition in MACSYMA we used "subst" even though it does not use the efficient Taylor-series representation. An internal program `pscsubst` uses the Taylor-series representation and is faster but could not be simply adapted for general use. (We experimented with `pscsubst`: Let ex be a 20th degree expansion of e^x, and st a 20th degree expansion of $\sin t$. Substituting st for x in ex took 11.6 seconds in a naive fashion using `subst`, but only 9.3 seconds if we first restructure ex using `horner`. By constrast, it takes 8.7 seconds using the internal `pcsubst`. Since the size and density of this test exceeds that of any of the sub-problems in our earlier timings, we do not expect we would get a great speedup by using `pscsubst`. After fixing a minor bug, differentiation of Taylor series in MACSYMA worked as intended; integration, however, did not maintain the Taylor-series format as it should.

Another tool which would be useful is a published complete implementation of the Newton Polygon procedure.(e.g. [3], [1]).

6 Conclusions and Future Work

Although we have found that in some contexts, expansions can be performed more rapidly, to low order at least, by methods already in MACSYMA, the techniques in [6] provide a set of tools, and a rather substantial theoretical basis for truncated power series expansions as solutions to algebraic equations. By means of approximating polynomials, one can solve certain transcendental equations also. Note that in [6], the Newton-based algorithms are shown to be asymptotically fast, yet this may not be particularly relevant in practice. Asymptotically-many terms in a power series may not be interesting, and in fact many applications require only a very few terms – enough to provide one term in the final result.

Extensions of the ordinary differential equations programs for expansions in other frameworks also provides a fruitful area for investigation. For example, expansions of solutions around ordinary points, regular singularities, and irregular singularities of unit rank of second order ODE's are developed in two of the references ([4] [8]). Providing a complete complement of tools ranging from closed form to series solutions to numerical solutions, should be one of the goals of algebraic manipulation systems in the next few years. (Since the first draft of this paper was written in 1980, in fact, some progress has been made on closed form solutions: See, for example, [2], [9]) In summary: these iterative tools should be more widely used than they are; their applicability can and should be extended to help solve iterative algebraic and transcendental equations, and differential equations. While we have illustrated only the simplest forms, it is straightforward to extend these examples to computations over systems of equations, and even to exploit parallel processors.

7 Acknowledgments

This work was supported in part by the following grants: National Science Foundation under grant number CCR-8812843; Army Research Office, grant DAAG29-85-K-0070, through the Center for Pure and Applied Mathematics, University of California at Berkeley; the Defense Advanced Research Projects Agency (DoD) ARPA order #4871, monitored by Space & Naval Warfare Systems Command under contract N00039-84-C-0089, through the Electronics Research Laboratory, University of California at Berkeley; the IBM Corporation; a matching grant from the State of California MICRO program.

References

[1] M. Anderson. An implementation of the Newton polygon procedure, Term Project for CS 292s, Winter, 1980 (University of California, Berkeley), 1980.

[2] J. Della Dora, C. Dicrescenzo, E. Tournier. An algorithm to obtain formal solutions of a linear homogeneous differential equation at an irregular singular point, *Proc. EUROCAM '82, Lecture Notes in Computer Science*, 144, Springer Verlag, 1982.

[3] W. G. Dubuque. Series Solution of Functional Equations:Taylor-solve, *MACSYMA Newsletter 3* no. 3 July, 1986, 18-19.

[4] R. Fateman. Some comments on series solutions, *Proc. 1977 Macsyma Users' Conf.*, NASA CP-2012 (July 1977) 43-52.

[5] K. O. Geddes. Convergence Behaviour of the Newton iteration for First Order Differential Equations, *Symbolic and Algebraic Computation, Lecture Notes in Computer Science 72*, Springer Verlag, 1979, 189-199.

[6] H. T. Kung and J. F. Traub. All algebraic functions can be computed fast, *J. ACM*, 25, 2 (April 1978), 245-260. See also: R.P. Brent and J. F. Traub. On the Complexity of Composition and Generalized Composition of Power Series, *SIAM. J. on Comp*, 9, 1 (Feb. 1980), 54-66.

[7] J. D. Lipson. Newton's method: A great algebraic algorithm, *Proc. 1976 ACM Symp. on Symbolic and Algebraic Computation*, R.D. Jenks, ed., 260-270.

[8] Y-M Shen. SOODESIS, Term Project for CS 292s, Winter, 1980 (University of California, Berkeley) 20 pp.

[9] M. F. Singer. Liouvillian solutions of n-th order homogeneous linear differential equations, *Amer. J. Math.* 103, (1981) 661-681.

Appendix

This is a listing of the "tweek" program.

```
;; tweek(h,n) changes the truncation upper degree
;; of a taylor series h to degree n.
;; e.g.  if f:taylor(1+x,x,0,10);
;; f: tweek(f,20); then
;; f is same as taylor(1+x,x,0,20);

(defun $tweek (h n)
  (cond
   ((atom h)); do nothing
   ((memq 'trunc (car h))
    ;; patch Macsyma's taylor series
    ;;representation in 2 places.
    (rplaca (car (cadaar(cdddr(cdar h)))) n)
    (or (numberp (cadr h))
     ;; don't change a constant powerseries
(rplaca (car (cadddr h)) n))
    )) h)

(setq taylor_simplifier nil)
```