A simple algorithm for the fast calculation of higher order derivatives of the inverse function

Roozbeh Dargazany *+, Karl Hörnes† and Mikhail Itskov†

† Department of Continuum Mechanics, RWTH Aachen University, Germany
* Department of Materials Science and Engineering, Massachusetts Institute of Technology, USA

Abstract

The paper deals with the calculation of higher order derivatives of the inverse function. A simple and fast recursive procedure is presented and compared with other methods known in literature both with respect to the computation time and memory usage.

1 Introduction

Let \( y(x) \) be a function which can be represented by the Taylor power series around a point \( x_0 \) as

\[
y(x_0 + \xi) = \sum_{i=0}^{\infty} \frac{\xi^i}{i!} y_i(x_0), \quad y_i(x_0) = \left. \frac{d^i y}{dx^i} \right|_{x_0}, \quad i = 1, 2, \ldots \tag{1}
\]

According to the Lagrange inversion theorem this function can be inverted if \( y_1(x_0) \neq 0 \). Then, the series expansion of the inverse function \( x = f^{-1}(y) \) around \( y_0 = y(x_0) \) is given by

\[
x(y_0 + \eta) = \sum_{i=0}^{\infty} \frac{\eta^i}{i!} x_i(y_0), \quad x_i(y_0) = \left. \frac{d^i x}{dy^i} \right|_{y_0}, \quad i = 1, 2, \ldots \tag{2}
\]

For the case \( x_0 = y_0 = 0 \) the derivatives of the inverse function are expressed by the Lagrange inversion formula as

\[
x_i = \left. \frac{d^{i-1}}{dx^{i-1}} \left( \frac{x}{f(x)} \right)^i \right|_{x=x_0}. \tag{3}
\]

Accordingly, the first few derivatives take the form

\[
x_1 = \frac{1}{y_1}, \quad x_2 = -\frac{y_2}{y_1^2}, \quad x_3 = 3\frac{y_2^2}{y_1^3} - \frac{y_2}{y_1^2}, \quad x_4 = -15\frac{y_2^3}{y_1^4} + 10\frac{y_2 y_3}{y_1^2} - \frac{y_4}{y_1^3}, \quad x_5 = 105\frac{y_2^4}{y_1^5} - 105\frac{y_2^2 y_3}{y_1^4} + 10\frac{y_2^2}{y_1^3} + 15\frac{y_2 y_4}{y_1^2} \frac{y_5}{y_1^4}. \tag{4}
\]
The calculation of higher order derivatives becomes excessively difficult and time-consuming since it requires multiple differentiation of the term \( \frac{f(x)}{f(x)} \). For this reason, alternative methods for the calculation of higher order derivatives of the inverse functions have always been the subject of interest.

Thus, Faa di Bruno proposed a formula \[3\] for the \( n \)-th derivative of a composed function which can also be applied to the inverse function. Despite of the simple implementation, the formulation becomes extremely complicated for higher order derivatives.

Later, Chernoff \[2\] introduced a movable strip method to approximate coefficients of the inverse series. The main idea was to reduce the computational efforts by rewriting the differentiation rules in a systematic order.

Following the approach of Ostrowski \[10\], Traub \[12\] derived a compact explicit formulation for the derivatives of the inverse function. Recently, Floater \[4\] presented the same formulation by expanding the Johnson’s combinatoric approach \[6\]. Accordingly,

\[
x_n = \sum_i (-1)^{k_i} \frac{(n + k_i - 1)!}{b_{i,2}! \cdots b_{i,n}!} \left( \frac{y_{2}}{2!} \right)^{b_{i,2}} \cdots \left( \frac{y_{n}}{n!} \right)^{b_{i,n}},
\]

where \( b_{i,2}, b_{i,3}, \ldots, b_{i,n}, i = 1, 2 \ldots \) denote non-negative integer solutions of the Diophantine equation

\[
b_{i,2} + 2b_{i,3} + \ldots + (n - 1)b_{i,n} = n - 1.
\]

For each of the solutions \( i, k_i \) is given by

\[
k_i = \sum_{j=2}^{n} b_{i,j}.
\]

Thus, for every order \( n \) a system of additional equalities and inequalities should be solved. Nevertheless, the Traub method is considerably faster than the classical Lagrange approach but still slow for higher order derivatives.

Apostol \[1\] further proposed the following simple expression

\[
x_n = \frac{P_n}{y_1^{n-1}},
\]

where

\[
P_{n+1} = y_1 P'_n - (2n - 1)y_2 P_n.
\]

Here, \( P_1 = 1 \) while \( P'_n \) is the derivative of \( P_n \) with respect to \( x \). Accordingly, the formulation requires a recursive differentiation of \( P_i \) which represent polynomials in \( y_j, (j = 1, 2, \ldots, i) \) with integer coefficients. Explicit expressions of these polynomials grow up and increase in complexity as \( n \) increases. Thus, the calculation of higher order derivatives becomes numerically expensive and extremely time consuming.

Johnson further simplified this approach by introducing a combinatorial argument for the representation of \( x_n \) \[6\]. Accordingly

\[
x_{n+1} = \sum_{k=0}^{n} y_1^{-n-k-1} (-1)^k R_{n,k}(y_2, \ldots, y_{n-k+2}),
\]

where

\[
R_{n,k}(y_2, \ldots, y_{n-k+2}) = \frac{1}{k!} \sum_i \left( \begin{array}{c} n + k \\ b_{i,1}, \ldots, b_{i,k} \end{array} \right) y_{b_{i,1}} y_{b_{i,2}} \cdots y_{b_{i,k}}.
\]
The summation is carried out over each set of integer solutions $b_{i,1}, b_{i,1}, \ldots, b_{i,k}$, $i = 1, 2 \ldots$ of the following Diophantine equation
\begin{equation}
 b_{i,1} + \ldots + b_{i,k} = n + k, \quad b_{i,j} \geq 2.
\end{equation}

Recently, Liptaj [9] proposed a solution for derivatives of the inverse function in a recursive limit form as
\begin{equation}
 x_n = \lim_{\Delta x \to 0} n! \frac{x - \sum_{i=1}^{n-1} \frac{x^i}{i!} \left( \sum_{j=1}^{n} \frac{x_j (\Delta x)_j^i}{j^i} \right)^i}{\left( \sum_{i=1}^{n} \frac{y_i (\Delta x)_i^i}{n} \right)^n}, \quad x_1 = \frac{1}{y_1}.
\end{equation}

Unlike the conventional methods the procedure of Liptaj does not require the successive differentiation and is relatively fast for lower order derivatives. For higher orders, however, the calculation of limits becomes very difficult and time consuming.

We further developed the procedure by Liptaj and proposed a simple recursive formula for the calculation of Taylor coefficients of the inverse function $y$. In this contribution, we shortly present this formula and additionally provide more insight into the computational aspects of the calculation. The advantages and capabilities of the proposed formulation are then illustrated in comparison with the procedures by Traub/Floater, Apostol, Johnson and Liptaj discussed above.

\section{Proposed formulation}

In view of (1) and (2) we can write
\begin{equation}
 x_0 + \xi = x (y (x_0 + \xi)) = x (y (x_0) + \eta) = x_0 + \sum_{j=1}^{n} \frac{\eta^j}{j!} x_j + O \left( \eta^{n+1} \right),
\end{equation}
where in comparison to (2) the series for $\xi$ is truncated after $n$ terms. Eliminating $x_0$ we further obtain
\begin{equation}
 \xi = \sum_{j=1}^{n-1} \frac{\eta^j}{j!} x_j + \frac{\eta^n}{n!} x_n + O \left( \eta^{n+1} \right).
\end{equation}
Assuming further that all $x_i$, $(i = 1, 2, \ldots, n - 1)$ are known the above equation can be rewritten by
\begin{equation}
 x_n = n! \frac{\xi - \sum_{j=1}^{n-1} \frac{\eta^j}{j!} x_j - O \left( \eta^{n+1} \right)}{\eta^n}.
\end{equation}
Keeping (1) and (2) in mind and utilizing a representation for power series raised to powers (see e.g. [2], p. 17), we get
\begin{equation}
 \eta^j = \left( \sum_{i=0}^{\infty} \frac{\xi^i}{i!} \right)^j = \xi^j \left( \sum_{i=0}^{\infty} \frac{\xi^i}{(i + 1)!} y_{i+1} \right)^j = \sum_{k=j}^{\infty} P_{j,k} \xi^k, \quad j = 1, 2, \ldots,
\end{equation}
where the coefficients are expressed in the recursive manner by
\begin{equation}
 P_{j,j} = y_j, \quad j = 1, 2, \ldots,
\end{equation}
\begin{equation}
 P_{j,k} = \frac{1}{(k - j) y_1} \sum_{l=1}^{k-j} (lj - k + j + l) \frac{y_{l+1}}{(l + 1)!} P_{j,k-l}, \quad k = j + 1, j + 2, \ldots
\end{equation}
Substituting (17) into (16) and enforcing $\xi \to 0$, we further obtain

$$x_n = \lim_{\xi \to 0} n! \frac{\xi - \sum_{j=1}^{n-1} \frac{x_j}{j!} \sum_{k=j}^{\infty} P_{j,k} \xi^k}{\sum_{k=n}^{\infty} P_{n,k} \xi^k}.$$  (19)

For the case $n = 1$ it yields the well-known result

$$x_1 P_{1,1} = 1.$$  (20)

The limit in (22) exists and is final if

$$\sum_{j=1}^{k} \frac{x_j}{j!} P_{j,k} = 0, \quad k = 2, 3, \ldots, n - 1.$$  (21)

Indeed, in this case (22) can be expressed by

$$x_n = -n! \lim_{\xi \to 0} \frac{\xi^n \sum_{j=1}^{n-1} \frac{x_j}{j!} P_{j,n} + \xi^{n+1} \sum_{j=1}^{n-1} \frac{x_j}{j!} P_{j,n+1} \cdots}{\xi^n P_{n,n} + \xi^{n+1} P_{n,n+1} \cdots},$$  (22)

which leads to the final representation as follows

$$x_n = -\frac{n!}{y_1^n} \sum_{j=1}^{n-1} \frac{x_j}{j!} P_{j,n}, \quad n = 2, 3, \ldots.$$  (23)

It is seen that this solution satisfies the conditions (21) for all $n$. Indeed, (23) can be rewritten as

$$x_n = -\frac{n!}{y_1^n} \sum_{j=1}^{n} \frac{x_j}{j!} P_{j,n} + \frac{x_n}{y_1^n} P_{n,n}, \quad n = 2, 3, \ldots,$$

which in view of (18) immediately implies (21).

3 Computational aspects and comparison with other methods

For the calculation and comparison of different methods we used the MAPLE program which includes a tool for the analytical differentiation. It was necessary both for the Apostol and Liptay procedure, where the limit can be obtained by l'Hôpital's rule.

Both the Traub/Floater and Johnson procedures require a solution of the Diophantine equation (6) and (12), respectively. For lower orders, all of the multiple solutions can easily be found by trial and error, as proposed by Floater [4]. For higher orders, however, this procedure becomes highly inefficient. For this reason, we apply here a new faster recursive solution algorithm presented in Appendix.

A peculiarity of the Diophantine equation (12) used by Johnson are equal coefficients one. Thus, any permutation of a solution $b_{i,1}, b_{i,1}, \ldots, b_{i,k}$ represents again a solution of this equation. The dominator $k!$ in the Johnson equation (11) represents the
number of all such permutations in the case of pairwise distinct roots $b_{i,1}, b_{i,1}, \ldots, b_{i,k}$. In the case of multiple roots equation (11) can thus be rewritten by

$$R_{n,k}(y_2, \ldots, y_{n-k+2}) = \frac{1}{s_{i,1}! \cdots s_{i,k}!} \sum_i \left( \frac{n+k}{b_{i,1}, \ldots, b_{i,k}} \right) y_{b_{i,1}} y_{b_{i,2}} \cdots y_{b_{i,k}},$$

(24)

where $s_{i,j}$ represents the multiplicity of the root $b_{i,j}$, $(j = 1, 2, \ldots, k)$. By this means, every solution is considered only one time, which significantly reduces computation expenses. Furthermore, we applied an efficient solver algorithm developed by Riha & James [11]. The so-obtained modified Johnson procedure is very fast in comparison to other similar algorithms.

For the comparison of the algorithms discussed above we used the Langevin function

$$f(x) = 1 - \coth x,$$

(25)

whose derivatives are given by

$$y_i(0) = \frac{2^{i+1} B_{i+1}}{i+1}, \quad i = 1, 2, \ldots,$$

(26)

where $B_n$, $n = 1, 2, \ldots$, denote the Bernoulli numbers. The inverse of (25) appears in the non-Gaussian theory of rubber elasticity as the chain force and plays an important role in polymer physics (see, e.g., [7]). However, the inverse Langevin function cannot be expressed in a closed form and requires an approximation.

The comparison criteria were

- computation time, and
- computer resources in particular the memory usage of the algorithm.

The stop threshold of the computation was set to 300MB of memory usage and 50s of calculation time. Figs. 1 and 2 illustrate respectively the computation time and the memory usage versus the order of the derivative calculated by the different algorithms. Accordingly, Apostol’s and Liptaj’s procedures could calculate 51 derivatives before running out of time while Johnson’s algorithm was able to obtain 71 derivatives before using all its allocated memory.

![Figure 1: Computation time required by different algorithms for the calculation of derivatives of the inverse function plotted versus the order of the derivative](image-url)
In Johnson’s approach, the number of solutions of the Diophantine equation \((12)\) grows rapidly with the order of the derivative. For instance, for \(n = 36\) the number of solutions is 17,977, increases to 37,338 for \(n = 40\) and goes over 200,000 for \(n = 50\). Thus, even by implementing a fast solver like for example that one by Riha & James [11], the computational efficiency of this method falls drastically at higher orders (see Fig. 2). In the Traub/Floater algorithm, the similar scenario takes place, where for \(x_{51}\) the Diophantine equation \((6)\) has more than 200,000 solutions. In Apostol’s algorithm, successive differentiations of lower order terms and in Liptaj’s algorithm, the limit equations with multiple summations lead to very complex and time consuming formulation for higher order derivatives.

In contrast, the proposed procedure does not require any analytical formulation like for example analytical differentiation or limit derivation. The higher memory usage of our algorithm at lower orders (see Fig. 2) is due to the initial memory allocation for the numerical matrix \(P_{i,j}\). At higher orders the amount of the allocated memory slightly increases due to the growing size of this matrix. Nevertheless, our procedure is considerably more effective in comparison to other above discussed algorithms both with respect to the required computer time and memory usage especially for higher order derivatives.

4 Conclusion

We have presented a simple and fast recurrent algorithm for the calculation of derivatives of the inverse function and compared it to other procedures known in literature. Accordingly, our algorithm appears to be highly advantageous both with respect to the calculation time and memory requirements.

Acknowledgment

The authors thank the Deutsche Forschungsgemeinschaft (DFG) for the financial support of this work.
References


Appendix A

The recursive algorithm for calculation of the solutions of the Diophantine equation given in Eq. (6).

```matlab
function sol = diophantine(n)
    % solves the diophantine equation b_2+2*b_3...+(n-1)*b_n = n-1 and stores results in solution matrix sol
    global row; global col; global sol;
    row = 0; col = 1; retcount = 0; A = [1:n-1]; c = n-1;
    for i = 1:n
        if (c-A(end)*(i-1))>=0
            ccounter = diophantine_recursive(A(1:(end-1)),c-A(end)*(i-1),n);
            retcount = retcount + ccounter;
            col = col + 1;
            for blockcount = row-ccounter+1:row
                sol(blockcount,col)=i-1;
            end
        else
            break;
        end
    end
end
```
function retcount = diophantine_recursive(A,c,n)
% recursive algorithm called during solution of diophantine equation
global row; global col; global sol;
retcount = 0;
if length(A) == 1
    row = row + 1;
    col = 1;
    sol(row,col) = c;
    retcount = retcount + 1;
else
    for i = 1:n
        if (c-A(end)*(i-1))>=0
            ccounter = diophantine_recursive(A(1:(end-1)),c-A(end)*(i-1),n);
            retcount = retcount + ccounter;
            col = col + 1;
            for blockcount = row-ccounter+1:row
                sol(blockcount,col)=(i-1);
            end
        else
            break;
        end
    end
end