Numeric-Symbolic Composite Derivative Calculations

James Daniel Turner

ABSTRACT

Composite derivative calculations arise in many applications in computational science and engineering. Since 1857 the gold standard for computing composite derivatives is the celebrated formula of Faà Di Bruno. The equation is an identity for generalizing the chain rule of calculus to higher dimensions. It is very complicated. Each sub calculation must satisfy two integer constraint equations. An alternative problem formulation, proposed in 1861 by George Scott, is analytically very simple: nevertheless, the requirement for computing hand-generated complex derivatives while enforcing a boundary condition, has limited its application. Symbolic methods are also available, but computationally expensive to embed in application software. This paper combines the best features of symbolic processing and Scott's formulation. The symbolic preprocessor computes (1) derivatives, and (2) enforces the derivative boundary condition appearing in Scott's method. For n requested composite derivatives; the preprocessor generates a lower triangular nxn array that is embedded in the application software for computing the numerical composite derivatives. Unless the number of requested composite work derivatives increases, the preprocessor is only called one time.

Keywords: NA

Classification: FoR Code: 0102

Language: English

Great Britain Journals Press

LJP Copyright ID: 392961
Print ISSN: 2631-8474
Online ISSN: 2631-8482

London Journal of Engineering Research

Volume 25 | Issue 6 | Compilation 1.0

Numeric-Symbolic Composite Derivative Calculations

James Daniel Turner

ABSTRACT

Composite derivative calculations arise in many applications in computational science and engineering. Since 1857 the gold standard for computing composite derivatives is the celebrated formula of Faà Di Bruno. The equation is an identity for generalizing the chain rule of calculus to higher dimensions. It is very complicated. Each sub calculation must satisfy two integer constraint equations. An alternative problem formulation, proposed in 1861 by George Scott, is analytically very simple: nevertheless, the requirement for computing hand-generated complex derivatives while enforcing a boundary condition, has limited its application. Symbolic methods are also available, but computationally expensive to embed in application software. This paper combines the best features of symbolic processing and Scott's formulation. The symbolic preprocessor computes (1) derivatives, and (2) enforces the derivative boundary condition appearing in Scott's method. For n requested composite derivatives; the preprocessor generates a lower triangular nxn array that is embedded in the application software for computing the numerical composite derivatives. Unless the number of requested composite work derivatives increases, the preprocessor is only called one time. The symbolic preprocessor easily scales for handling ten's to hundred's of composite derivatives. A numerical example is provided, where 1..5 composite derivatives are computed.

Author: Amdyn Systems Inc.

I. INTRODUCTION

Composite function calculations arise in all areas of scientific and engineering computing. Low-order applications present no technical problems for manually derived calculations. High-order applications, however, rapidly become cumbersome to manage the bookkeeping involved. Alternatively, symbolic tools are available, such as Maple, Mathematica, Octave (used herein) and others, for generating arbitrary order derivative calculations. Unfortunately, run-time penalties make it is impractical to embed a algebraic symbolic program within an application program. Since *1857* the work of Faa Di Bruno [1],[2],[3] has served as the gold standard for computing composite derivatives. This work combines the power of symbolic computing with the simple elegance of George Scott's *1861* formulation [4]. For five requested composite derivative calculations, the symbolic software generates a *5x5* lower triangular array, that is embedded in the application software for computing numerical derivative values. A numerical example is provided to demonstrate the effectiveness of the proposed algorithm.

1.1 Faa Di Bruno Formulation

Since 1857 scalar composite function calculations have been theoretically handled by invoking the celebrated work of *Faà* Di Bruno [1],[2],[3]. His formula for computing the *nth* derivative of the function $u = F\left(G\left(x\right)\right)$ is given by

London Journal of Engineering Research

$$\frac{d^n}{dx^n}F\left(G\left(x\right)\right)=\sum\frac{n!}{b_{1!}b_{2!}...b_{n!}}F^{(k)}\left(G\left(x\right)\right)\left(\frac{G'}{1}\right)^{b_1}\left(\frac{G''}{1.2}\right)^{b_2}\left(\frac{G'''}{1.2.3}\right)^{b_3}...\left(\frac{G^{(n)}}{1.2...n}\right)^{b_n}$$

where the summation process is subject to the following integer constraint conditions

$$b_1+b_2+b_3+...+b_n=k$$
$$b_1+2b_2+3b_3+...+nb_n=n$$

No additional derivative calculations are required. Not surprisingly the calculation represents a daunting bookkeeping challenge.

1.2 Scott's Composite Derivative Formulation

In 1861 George Scott [4] proposed the following alternative composite function derivative formulation

$$\frac{d^n}{dx^n}F\left(G\left(x\right)\right)=\sum_{k=1}^{n}\frac{F^{(k)}\left(G\left(x\right)\right)}{k!}\left\{\frac{d^n}{dx^n}G^k\left(x\right)\right\}\Bigg|_{G=0} \tag{1}$$

which is analytically much simpler than *Faà* Di Bruno's identity. Unfortunately, the requirement for generating hand-derived derivatives for $\frac{d^n}{dx^n}G^k\left(x\right)\Big|_{G=0}$, where n denotes the nth derivative and k denotes derivative summation index, has limited its broader application in the computational research community.

II. NUMERIC-SYMBOLIC COMPOSITE DERIVATIVE FORMULATION

This paper presents a composite derivative algorithm that combines: (1) the simplicity of Scott's formulation [4], and, (2) the power of algebraic symbolic manipulation. Only two function routines are required (see Figure 1 and Appendix A). Symbolic function derivatives are computed for a generic *G(x)* function. The use of a generic function (see Figure 1), eliminates the need for embedding a symbolic program in the application software. The algorithm consists of two parts. First, a symbolic preprocessor (see Figure A) computes the derivatives for $\frac{d^n}{dx^n}G^k\left(x\right)\Big|_{G=0}$ and enforces a *G=0* boundary condition on each derivative. The derivative calculations become very complicated as the requested number of composite derivatives increase. Nevertheless, after the boundary condition *G=0* is enforced, the elements of the output derivative array are very simple. The symbolically generated derivative array consists of a lower triangular *nxn* array; which is copied into the application software for numerical derivative calculations. The elements of the array consist of components of the vector array {V1 V2 ...Vn}, where Vi denotes $\frac{d^i}{dx^i}G$. The array is only recomputed if the number of requested derivative increases.

Second, the application program embeds the computational function routine appearing in *Figure 1*, for numerically computing the composite derivatives. An *elegantly* simple double summation algorithm generates the numerical results for the composite derivatives.

For *nd = five* requested composite derivatives, the symbolic preprocessor, presented in Appendix A, generates the following *dGdx* lower triangular *nd* x *nd* array:

$$dGdx = \begin{bmatrix} V_1 & & & & \\ V_2 & 2V_1^2 & & & \\ V_3 & 6V_1V_2 & 6V_1^3 & & \\ V_4 & 8V_1V_3+6V_2^2 & 36V_1^2V_2 & 24V_1^4 & \\ V_5 & 10V_1V_4+20V_2V_3 & 60V_1^2V_3+90V_1V_2^2 & 240V_1^3V_2 & 120V_1^5 \end{bmatrix} \tag{2}$$

where $Vi = \frac{d^i G}{dx^i}\Big|_{G=0}$. The *dGdx* array of Eq. 2 enables 1 through 5 composite derivatives to be evaluated. Elements of the array are nonlinear, for example, the element *dGdx*(4,2) = $8V_1V_3+6V_2^2$. During the preprocessing operations, the following change of variables eliminates potential problems with Processing the nonlinear terms:

$$\{G' \quad G'' \quad G''' \quad G^{(4)} \quad G^{(5)}\} = \{V_1 \quad V_2 \quad V_3 \quad V_4 \quad V_5\}$$

The change of variables substitution starts with the highest order derivative and continues until the lowest order derivative variable has been replaced (see Appendix A).

An additional benefit of this approach is that for applications only requiring three composite derivatives, one only uses the sub portion of the array dGdx(1:3,1:3) for all numerical derivative calculations.

III. COMPOSITE DERIVATIVE CALCULATIONS

Taking the symbolically generated *dGdx* array of Eq. 2 as input for the application software; composite derivatives are computed with following *function routine*:

Function Composite = Derivative(nd, x, *DF*, *DG*, *Rfac*)

%% This function computes composite derivatives for nd derivatives.
%% The Maximum number of derivatives is five(5), otherwise the
%% symbolic preprocessor is recomputed to generate a new *dGdx* array.

%% *INPUT*:

%% nd number of composite derivatives requested
%% x evaluation point for the composite derivative

%% $DF = \{F' \quad F'' \quad F''' \quad F^{(4)} \quad F^{(5)}\}\Big|_{F|_{G|=x}}$

%% $DG = \{G' \quad G'' \quad G''' \quad G^{(4)} \quad G^{(5)}\}\Big|_{G(x)}$

%% where *x* denotes the evaluation point for the derivatives
%% *DF & DG* are *user supplied nd x 1* arrays of derivatives numerically
%% evaluated at the application points
%% *Rfac* = reciprocal factorial array *nd x 1*

%% *OUTPUT*:

%% *Composite* denotes the *nd x 1* array of requested composite derivatives
%%==========

%%==========

$$V_1 = DG(1);\ V_2 = DG(2); V_3 = DG(3); V_4 = DG(4); V_5 = DG(5);$$

%% Introduce Current G(x) Derivative values in the following *dGdx* array

%% *dGdx* is copied from the symbolic preprocessor of *Appendix A*

$$dGdx(1,\ 1) = V_1,\ dGdx(2,\ 1) = V_2,\ dGdx(3,\ 1) = V_3,\ dGdx(4,\ 1) = V_4,\ dGdx(5,\ 1) = V_5$$
$$dGdx(2,\ 2) = 2V_1^2$$
$$dGdx(3,\ 2) = 6V_1V_2$$
$$dGdx(4,\ 2) = 8V_1V_3 + 6V_2^2$$
$$dGdx(5,\ 2) = 10V_1V_4 + 20V_2V_3$$
$$dGdx(3,\ 3) = 6V_1^3$$
$$dGdx(4,\ 3) = 36V_1^2V_2$$
$$dGdx(5,\ 3) = 60V_1^2V_3 + 90V_1V_2^2$$
$$dGdx(4,\ 4) = 24V_1^4$$
$$dGdx(5,\ 4) = 240V_1^3V_2$$
$$dGdx(5,\ 5) = 120V_1^5$$

```
%% Begin Main Derivative Summation Loop
for n = 1:nd     %% Number of requested composite derivatives
    s = 0;       %% initialize each derivative summation
   for k = 1:n   %% Number of summation terms for each derivative
       s = s  +  DF(k)*dGdx(n,k)*Rfac(k);
    endfor       %% Sum current Derivative Series Expansion
    Composite(n) = s; %% Save nth value of the composite derivative
endfor %% double summation algorithm
endfuction
```

Figure 1: Numerical Composite Derivative Function Routine

This double summation algorithm is extremely simple when compared with Faa Di Bruno's formulation [1],[2],[3].

IV. NUMERICAL APPLICATION

This Section presents a numerical composite derivative example, where *five(5)* composite derivatives are requested. The assumed composite derivative function is given by:

$$\frac{d^n}{dx^n} F\left(G(x)\right) = \tan\left(\cosh(x)\right)\Big|_{x=0.5} \tag{3}$$

There are two numerical evaluation points for the function appearing in Eq. (3). Namely,
Evalpt = 0.5 for G(Evalpt), and
FEvalpt = cosh(Evalpt)

The user is assumed to have provided numerical values for the following analytic derivative arrays for *F & G:*

$$DF = \left\{ F' \quad F'' \quad F''' \quad F^{(4)} \quad F^{(5)} \right\}\Big|_{\cosh(x)} \tag{4}$$

$$DG = \left\{ G' \quad G'' \quad G''' \quad G^{(4)} \quad G^{(5)} \right\}\Big|_{x} \tag{5}$$

These numerical arrays are processed in the double summation algorithm presented in *Figure 1*. Assuming that the evaluation point for the composite derivative is *x=0.5*, the above derivative arrays have the following truncated numerical values

$$DF = \begin{bmatrix} 3.5495\text{e+}00 & 1.1335\text{e+}01 & 6.1395\text{e+}01 & 4.3746\text{e+}02 & 3.9112\text{e+}03 \end{bmatrix}$$

$$DG = \begin{bmatrix} 1.5056\text{e-}01 & 1.0113\text{e+}00 & 1.5056\text{e-}01 & 1.0113\text{e+}00 & 1.5056\text{e-}01 \end{bmatrix}$$

Using the double summation algorithm presented in *Figure 1*, the five requested composite derivatives are given by:

$$CD = \begin{bmatrix} 5.3442\text{e-}01 & 3.8464\text{e+}00 & 5.9215\text{e+}00 & 4.8062\text{e+}01 & 1.8572\text{e+}02 \end{bmatrix}$$

which have been crossed checked with purely symbolic derivative calculations.

V. CONCLUSION

This work links (1) the power of symbolic processing for computing derivative values for $\frac{d^n}{dx^n} G^k(x)\Big|_{G=0}$ and the simplicity of Scott's algorithm presented in Eq. (1). Numerical composite derivative calculations are preformed using the elegantly simple double summation algorithm presented in *Figure 1*. Up to five composite derivatives are handled by the algorithm. If more composite derivatives are required; the symbolic preprocessor of Appendix A is recomputed to generate an updated derivative array for *dGdx*. The symbolic preprocessor performs derivative calculations and enforces a boundary condition on each derivative. The output of the symbolic preprocessor consists of a lower triangular 5x5 matrix. The elements *dGdx* functionally depend on the nonlinear derivatives of generic function *G(x)* . The derivative array *dGdx* is embedded in function routine presented in *Figure 1* for computing the numerical composite derivatives. Assuming that the upper limit for the requested number of composite derivatives does not change; the preprocessor is only executed one time. The algorithm scales for tens to hundreds of composite derivatives. A numerical example is presented that demonstrates the effectiveness of the proposed methodology.

REFERENCES

1. Faa di Bruno, "Noet sur une nouvelle formule de calcul differntiel," Quart. J. Pure Appl. Math. 1 (1857) pp. 359-360.
2. S. Roman, "Faa di Bruno's formula," The Mathematical Association of America, Monthly, Vo. 87, No. 10. (Dec., 1980), pp. 805-809.
3. Warren P. Johnson, "The Curious History of Faá di Bruno's formula," The Mathematical Association of America, Monthly 109, March 2002, pp. 217-234.
4. George Scott, "Formulae of successive differentiation," Quaterly J. Pure Appl. Math. Vol. 4, (1861), pp. 77-92.

London Journal of Engineering Research

APPENDIX A

Symbolic Constrained Derivative Preprocessor

This function routine uses symbolic processing to compute and enforce derivative boundary conditions. A generic function is used to generate the derivatives and enforce boundary conditions on $\left.\frac{d^n}{dx^n} G^k(x)\right|_{G=0}$. The use of a generic function *G(x)* eliminates the need for embedding a symbolic manipulation tool in the application software. The generation of symbolic derivatives is straight forward. Enforcing the *G=0* boundary condition of the derivative expressions requires an indirect approach to avoid a potential loss of nonlinear terms. This is handled by introducing a change of variables to eliminate the explicit derivative expressions in the symbolic derivative results. The calculation is started by eliminating the highest derivatives first. The calculation generates a lower triangular array *dGdx*, that is inserted in the application software as the function routine appearing in *Figure 1*. This array is computed only one time, and is only recomputed if the number of requested composite derivative increases.

```
function dgk = Sym_dGdk_preprocessor ( nd )
%% Symbolic PreProcessor for composite derivatives
%% Math Model:  phi = f( g(x) ), @ x = evaluation point
%% INPUT: nd number of implicit derivatives required on output
%%=====================================================
%% INPUT:      nd Number of implicit derivatives required
%% OUTPUT: dgk nxn lower triangular array of derivatives @ G = 0
%% COPYRIGHT (2025) James D. Turner Allrights reserved
syms G(x);       %% Declare variables to be symbolic
syms k;
%% Create change of variables array to eliminate analytic dervs:
symbols = sym( zeros( 1, nd ) ); % Preallocate symbolic array
for i = 1:nd
   symbols(i) = sym( sprintf('v%d', i));
   % Create symbols = { v1, v2, ..., vn }
End  %% => ( G' = V1, G'' = V2, ...,G(n) = Vn )

old   = sym ( zeros( nd,1) );
new = sym ( zeros( nd,1) );     %% allocate memory for derivative transformations

for i = 1:nd
 old(i,1)  = diff( G(x), x, i );     %% Load G derivatives
 new(i,1) = symbols(i) ;          %% Load { v1, v2, ..., vn }
endfor                           %% Store derivative transformation equations
oldr  =  old( end:-1:1 );
newr =  new( end:-1:1 );           %% Reverse order of derivative & V arrays
dgk  = sym( zeros( nd, nd ) );     %% allocate array for enforcing G = 0 BC
    %%================================
    %% Symbolic Preprocessor Loop
    %%================================
for n  = 1:nd                      %% n-th derivative loop
 for k = 1:n                       %% summation terms for each derivative
```

```
   deriv = diff( G(x)^k, x, n );      %% compute nth derivative ( vary complicated )
   ans   = expand( deriv );           %% ans = { G^(n)*f0 + G^(n-1)*f1 + ... + fn }
   ans   = subs( ans, oldr, newr );  %% change of variables for higher derivatives
   G_BC  = subs( ans, G(x), 0);    %% enforce G = 0 Boundary Condition
   dgk( n, k ) = [ G_BC ];           %% save nk th element of constrained derivative array
    endfor
endfor
endfunction
```