To appear in *Optimization Methods & Software*
Vol. 00, No. 00, Month 20XX, 1–15

Comparing high-order multivariate AD methods

Richard D. Neidinger* and Benjamin Altman†

Davidson College, Davidson, NC, USA

(*23 April 2018*)

To compute every high-order multivariate derivative value, interpolation methods will be shown to be less accurate than a direct forward multivariate Taylor series method, becoming significant for degrees higher than ten. As order increases, interpolation methods rely on increasingly ill-conditioned matrices where simply rounding exact rational values produced corresponding error in some resulting derivative values. Both interpolation and direct methods use forward AD (algorithmic differentiation); the direct method propagates multivariate series coefficients of the original function, while interpolation methods propagate univariate series of the function in fixed directions and reconstruct the multivariate values. Such interpolation methods, differing in direction choices and reconstruction methods, have been shown to be theoretically more efficient than the direct method for high order. Four alternatives were implemented in MATLAB (interpreted and using random access arrays) on the same laptop. In our implementations, the direct method was competitive and often faster in run time, in addition to maintaining good accuracy. Since AD tools in compiled languages are much faster, more comparison is needed. Direct method efficiency depends on indexing subsets within the large non-rectangular data structure for multivariate series coefficients. We explain key implementation details of our direct method that uses a global reference array.

Keywords: differentiation; high order; multivariate; series; interpolation; forward

AMS Subject Classification: 65D25; 65Y20; 65D05; 41A63; 41A05

1. Introduction

Forward AD (algorithmic or automatic differentiation) methods using Taylor series are known to be an effective tool for computing arbitrarily high order derivatives. Multivariate Taylor series coefficients (or unique partial derivative values at a point) up to some arbitrary but fixed high order d form a large non-rectangular data structure. For any analytic $f : \mathbb{R}^n \to \mathbb{R}$ at $\mathbf{a} \in \mathbb{R}^n$, terms of the Taylor series are referred to by a multi-index $\mathbf{i} = (i_1, \ldots, i_n)$ of non-negative integers so that

$$f(\mathbf{x}) \approx \sum_{|\mathbf{i}| \le d} F(\mathbf{i})(\mathbf{x} - \mathbf{a})^{\mathbf{i}} \tag{1}$$

where the power is componentwise. The degree of a term is given by the multi-index order $|\mathbf{i}| = i_1 + \cdots + i_n$, so $|\mathbf{i}| \le d$ cuts off the corner at the origin of the n-dimensional box of multi-indices $\le d$ componentwise. For simplicity, we use the term *corner* to refer to the data structure of $F(\mathbf{i})$ (or corresponding derivative value) for $|\mathbf{i}| \le d$, although, more

*Corresponding author. Email: rineidinger@davidson.edu
†Undergraduate research.

precisely, the indices are the lattice points in a d-scaled standard n-simplex. In algorithms, we use a multi-index ordering in terms of order first and then reverse lexical within order. The real coefficient $F(\mathbf{i}) = D_{\mathbf{i}}(f)/\mathbf{i}!$ gives the unique partial derivative for that multi-index, not distinguishing between derivative permutations, divided by the product of each index factorial. Our goal is to evaluate algorithms that compute a full corner of $F(\mathbf{i})$ values (and deduce the derivatives values $D_{\mathbf{i}}(f)$). This can be a massive problem since there are $\binom{n+d}{n}$ entries in a corner. On a laptop, a realistic bound of one million entries restricts (n, d) pairs to under or near a curve through $(4, 67), (11, 11)$, and $(67, 4)$, shaped something like a hyperbola. Somewhat smaller numbers will be used for testing, since extended precision will be used and inefficient methods will be compared. Sparsity will not be considered in this context although, in application, sparsity and contextual structure are very important considerations. We will assume that a seed matrix ([5], p. 311) has already restricted to a smaller subspace of dimension n where all derivatives may be non-zero. We also assume that order d is fairly large, say 4 to 25, so that interpolation can have a theoretical advantage and forward can compete with reverse strategies [4].

Four methods will be compared, all using forward AD to compute either multivariate or univariate series.

The first method is just forward multivariate AD that propagates corners of multivariate Taylor series coefficients, starting with the trivial variables and overloading all operations and functions in a program. Such implementations have been around for over 25 years, as in [2]. Our implementation, similar to [12] (see references list there for other direct multivariate papers), keeps the corner in a one-dimensional array and uses a global structure to store indices of needed subsets of the corner. Section 4 explains details of our indexing scheme that enables efficient implementation in MATLAB. We use the term *Forward Corner* program to refer to this implementation of forward multivariate AD using Taylor series coefficients and indirect referencing.

The other three methods, all called interpolation, compute univariate Taylor coefficients $\mathbf{u}_j$ for directional restrictions

$$f(\mathbf{a} + t\mathbf{r}_j) \approx \sum_{k=0}^{d} \mathbf{u}_j(k)\, t^k \tag{2}$$

for some directions $\mathbf{r}_j$ where $j = 1$ to $\binom{n+d-1}{n-1}$ (one for each multi-index of order d) and then reconstruct the multivariate $F(\mathbf{i})$ for all $|\mathbf{i}| = k \leq d$. The second method is *GUW interpolation*, from the original paper [4] (and presented in [5]), where the $\mathbf{r}_j$ are the multi-indices $\mathbf{j}$ with $|\mathbf{j}| = d$, in reverse lexical ordering, and all of these directions are used to compute all $F(\mathbf{i})$ even when the order $|\mathbf{i}| = k < d$. This method is implemented in some AD tools, in particular, ADOL-C and Rapsodia. The last two methods will be called *nested interpolation*, from [8], where $\mathbf{r}_j = (1, \mathbf{q}_j) = (1, w_{j_2}, \ldots, w_{j_n})$ for the jth $\mathbf{j} = (j_1, j_2, \ldots, j_n)$ with $|\mathbf{j}| = d$ and a fixed sequence $w = 0, 1, -1, 1/2, -1/2, 1/4, -1/4, 3/4, -3/4, 1/8, \ldots$. These directions are nested in two ways: first, to compute $F(\mathbf{i})$ for $|\mathbf{i}| = k < d$, only that number of the $\mathbf{r}_j$ are needed; second, when decreasing d, the set of $\mathbf{r}_j$ is just the corresponding initial subset from the larger d. Unlike GUW, this means that global arrays can be computed for a large d and only the needed subset used for smaller d. The two different nested implementations use different reconstruction methods. The *Nested Div-Diff* method does not rely on global arrays but computes a Newton form of the polynomial using divided differences and gathers terms to find $F(\mathbf{i})$. The *Nested LU* method stores LU decompositions of matrices to perform a matrix solve for $F(\mathbf{i})$ values. Only a sense of the method differences is needed now, further practical understanding of the interpolations are described in section 3.

For comparison purposes, all four methods were implemented in MATLAB 2015b on the same laptop running Windows 7 on an Intel Core i7-5600U CPU @ 2.60GHz with 16 GB of RAM. Original implementations were done for an undergraduate thesis [1]. We strove to simply code the methods as described in the corresponding papers, trying to be efficient when implementation details were not specified. However, we did not take advantage of sparsity and did not vectorize the repeated computation of univariate series in different directions. These are prototype implementations that only apply to standard scalar-valued operations and functions in a program (function), not exhaustive MATLAB codes. The prototype Forward Corner code is available at http://www.neidinger.net/publicat.html along with a (less polished) collection of Comparison Test Codes.

2. Comparing accuracy

Since our main point is that the interpolation methods are not as accurate as forward multivariate AD, we first focus on accuracy. Motivated by pure research curiosity, we seek to know if it is possible to accurately compute every individual multivariate Taylor coefficient, or equivalently each mixed partial derivative at a point, up to a high order. "Why should we compute them?" is actually a serious question without an answer in this paper. We do not have a specific application in mind. We will comment on error in the more practical and easier goal of computing an accurate multivariate Taylor polynomial value at a point some distance from the center where coefficients were calculated.

The first test function is a model of the horizontal range of a tennis serve as a function of three variables, initial angle a, speed v, and height h. This simple projectile motion model, from [3] p. 263, solves for where the parabolic path hits height zero. An efficient form of the resulting formula is implemented in the MATLAB function:

```
function f = tennisSI(params)
a = params(1);  % angle in degrees
v = params(2);  % speed in m/sec
h = params(3);  % height in m
rad = a*pi/180;
tana = tan(rad);
vhor = (v*cos(rad))^2 / 9.80665;
f = vhor * (tana + sqrt(tana^2 + 2*h/vhor)); %horizontal range
```

We use initial values $a = 20$, $v = 13.5$, and $h = 2.75$, and compute using structures for $n = 3$ variables up to $d = 25$ order of derivatives (degree of Taylor series). In the Forward Corner program, the function `tennisSI` is run only once using a corner object in place of each of the three variables in `params`; the resulting multivariate Taylor series are converted to derivative values. In the three interpolation methods, `tennisSI` is run multiple times using univariate series class objects (from [9]) initialized for each direction; then partial derivative values are reconstructed (first through multivariate series values in nested methods). The methods are theoretically exact AD methods, so accuracy is limited only by floating point precision, which will be double for all four methods and quadruple for reference (using MATLAB vpa in the most reliable Forward Corner program). Of course, modeling error would be larger but we take the quadruple precision value as the true value for the purpose of comparing error in these methods.

The resulting maximum errors over all 3,276 values are presented in different ways in Table 1. All largest errors in the first row of Table 1 occur on a 25th order partial

Table 1. Maximum errors for tennisSI with $n = 3$ and $d = 25$.

Maximum Absolute	Forward Corner	GUW Interpol.	Nested Div-Diff	Nested LU
Derivative error	6.68e-06	7.93e+06	1.72e+05	9.65e+04
Taylor error	3.55e-15	1.80e-12	3.55e-15	3.55e-15
Relative error	4.63e-06	7.34e+18	5.11e+15	1.23e+16
Order relative	6.66e-15	7.91e-03	1.71e-04	9.63e-05

derivative and, except for Nested Div-Diff, on the largest absolute derivative $D_{(0,0,25)}f \approx$ `1.0025e+9`. We divide all row one values by this largest value (even though Nested Div-Diff error `1.72e+05` occurs on a much smaller value) to get the measure of error in the last row. We call this *order relative* error since we can fix any order and look at the maximum absolute error of that order relative to the largest value of that order. In the worst case (highest order 25) in this order relative sense, we see that Forward Corner achieves effectively full numerical accuracy, while the interpolation methods give a few meaningful digits. As fixed order grows from 1 to 25, the order relative error steadily grows for the interpolation methods as shown in Figure 1. To show that this phenomenon does not depend on the absolute derivative values growing large, we simply change units from meters to feet (where tennis dimensions originate) and then the maximum absolute derivative values stay small, around 10^{-4} or 10^{-5} for order ≥ 5. Figure 1 was actually generated using units of feet (with 32 for acceleration of gravity and initial values $a = 20$, $v = 44$, and $h = 9$). The same plot using SI units looks very similar except that maximum values grow to 9, for 10^9 value.

The maximum absolute Taylor coefficient errors in row two of Table 1 are small enough that all of the methods are reliable for computing a Taylor polynomial value at a point close to the series center. For example, at $(20 + h, 13.5 + h, 2.75 + h)$ for $h = 0.5$, all of the methods give full double precision accuracy, while the difference between polynomial and function value is 10^{-23} (truncation error); so such high-order was not really needed. However, larger h need more accuracy and by $h = 2$ the GUW interpolation error is significant, resulting in absolute error in the polynomial value of 10^{-3} on a true value of around 25.0161456 where the polynomial truncation error is of order 10^{-7}. For this example, the nested interpolation methods still give full double precision in the true polynomial value. In an example in Section 2.1 (analyzed by *Mathematica*, in addition to this MATLAB implementation), all of the interpolation methods result in significant polynomial value error. The Forward Corner implementation had the most accurate order of error in all polynomial value tests, usually 10^{-15} relative error or better.

The maximum relative errors in row three of Table 1 look at each individual term regardless of size. The computed relative errors are the same for derivative or Taylor values, as expected since all computations for the same term are scaled by the same factorials. The scalings are huge so that overflow or underflow could cause a difference for larger d, but no difference was observed up through $d = 25$. Maximum relative error for the Forward Corner program shows that every one of the corner values has at least five good digits, while the interpolation methods can have ridiculously large relative error, as could be expected for relatively small values in the linear solve. For example, the largest derivative error for Nested Div-Diff `1.72e+05` occurs on the tiny value $D_{(0,24,1)}f \approx$ `4.74e-11`, yielding something close to the largest relative error. For order > 13, all interpolation methods have some value with relative error > 1, meaning that the value would be useless. Actually, many small values are computed well by the methods; the smallest abs derivative $\left|D_{(24,0,0)}f\right| \approx$ `7.65e-16` is computed with over 13 good digits in all methods except GUW Interpolation. Still, in some values, the GUW

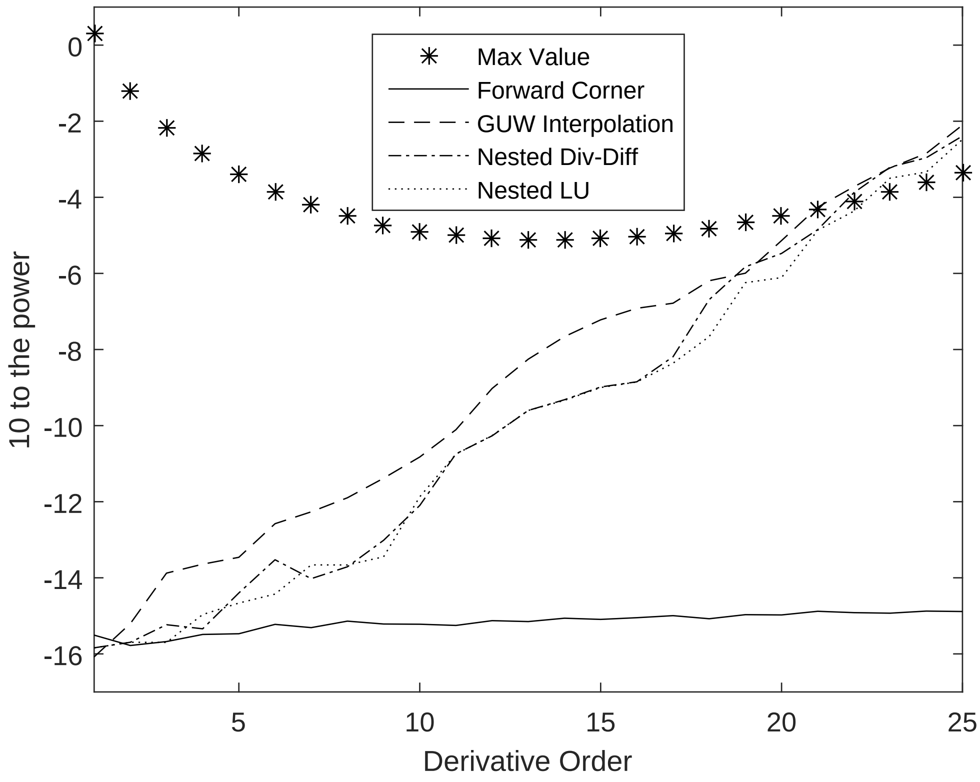

Figure 1. Order relative error, log(max error / max value), with max value when using feet.

Interpolation is better than the nested interpolation methods, so it is hard to generalize except to say that the Forward Corner program is much more reliable.

Similar errors were also experienced by ADOL-C (which uses the GUW-interpolation method). To get easily comparable errors, the tennis function (using feet) $t(a, v, h)$ was embedded (in a meaningless way) into a normalized function that has derivative 1 for every possible higher-order derivative at (1,1,1). Series coefficients for t are computed but then divided by themselves in

$$g(x, y, z) = \exp\left(\frac{t(x, y, z)}{t(x, y, z)}(x - 1) + (y - 1) + (z - 1)\right).$$

The maximum error for a 20th derivative was `5.6e-16` for Direct Forward but `1.7e-04` for our GUW Interpolation and `6.0e-05` for ADOL-C implementation.

To understand what causes these errors and to confirm that they come from the methods, and not the specific example or implementation, we now study an extremely simple example

$$f(x, y) = \exp(x + y) \text{ at } (0, 0), \tag{3}$$

and compute $D_{\mathbf{i}}(f) = 1$ up through $|\mathbf{i}| \leq d = 9$ and $d = 25$. Relative and absolute errors are the same, since the exact value is one. Our Forward Corner implementation gives full numerical accuracy, with maximum absolute error of `2.2e-16` for $d = 9$ and `5.6e-16` for $d = 25$, so the method does not introduce any error. The interpolation methods will be implemented by matrix operations in *Wolfram Mathematica*, where we can use exact arithmetic (simulating what might be done by hand) to show the source of numerical error.

2.1 *Matrix analysis of GUW interpolation error*

GUW interpolation can be described as multiplication by matrices in

$$H_k \ [\mathbf{u}_j(k) : \text{directions } \mathbf{r}_j \text{ are each } |\mathbf{j}| = d] = [D_{\mathbf{i}}(f) : |\mathbf{i}| = k] \tag{4}$$

for each order k. The entries in the matrices H_k are given by the explicit formula (17) in [4] which results in rational number entries. Depending only on n and d, the matrices H_k are $m_k \times m_d$ where

$$m_k = \binom{n+k-1}{n-1}$$

is the number of multi-indices of order k. Unlike nested interpolation, increasing d for the same n means starting over with new H_k's and directions. We will assume absolutely no error in the $[\mathbf{u}_j(k)]$ matrix of series coefficients which is especially simple for (3). For $d = 9$, $\mathbf{r}_j = (9,0), (8,1), \ldots, (0,9)$, so $f((0,0) + t\mathbf{r}_j) = \exp(9t)$ for all j. Thus $\mathbf{u}_j(k) = 9^k/k!$ for all j and k. In *Mathematica*, entries in H_k are computed with the same program (following formula (17) from [4]) but with three different exact rational or numerical double results. Using exact rational inputs, H_k will denote the exact rational matrix result; then $N[H_k]$ will denote conversion of H_k to numerical doubles; finally, `NHk` will denote the result of numerical double inputs to the *Mathematica* program, resulting in floating point computations throughout generation of the matrix. Since all derivatives should be 1, errors from (4) are easy to calculate and the resulting maximum absolute errors are in Table 2. The first row verifies correct implementation of this perfect method

Table 2. Maximum absolute errors using H_k to compute derivatives of $\exp(x + y)$ at (0,0).

Same *Mathematica* using	$d = 9$ error	$d = 25$ error
H_k exact rationals	0	0
$N[H_k]$ exact to double	7.0e-14	4.4e-07
`NHk` doubles throughout	1.5e-11	1.6e+01
Our MATLAB GUW Interpol.	1.1e-11	1.4e+00

in the absence of roundoff. The third and fourth rows show that floating point creation of H_k explains errors experienced by our GUW Interpolation in MATLAB (which does create and multiply by global H_k matrices), even ignoring any error in univariate series. The second row further isolates the problem to the matrix, not implementation details.

The computed derivative values were used in the 25th order multivariate Taylor polynomial $p(h, k)$ approximating e^{h+k}. While all methods were accurate for the true value of $p(2, 2)$, all interpolation methods struggle with $p(-2, 4)$ resulting in absolute error of order about 10^{-8} in the MATLAB implementations. In the *Mathematica* computations, the error in $p(-2, 4)$ was about 10^{-10} using the coefficients from $N[H_k]$, and almost 10^{-8} using coefficients from `NHk`. These are significant since true $|p(-2, 4) - e^2| \approx 10^{-19}$. Massive cancellations lead to this tiny truncation error but are not really responsible for the interpolation method errors, since $p(2, 4)$ has about the same interpolation errors and (they are not as significant since) the truncation error is almost 10^{-6}. In all these cases, the Forward Corner implementation gives full numerical accuracy in the polynomial value.

The second row in Table 2, from roundoff only late in the process of GUW interpolation, is the inevitable error as now suggested by condition number. In our analyses of this example, the condition number is effective in predicting the maximum relative (and also absolute since exact is always 1) error in derivative values computed using $N[H_k]$ (and $N[M_k^{-1}]$ in the next section). The effectiveness of the condition number is perhaps unreasonable, since H_k is not used in a linear solve, and the norm of H_{25} is about one. Nevertheless, the condition numbers $\text{cond}(H_9) \approx 10^3$ and $\text{cond}(H_{25}) \approx 10^{10}$ when multiplied by machine epsilon around 10^{-16} are close to the errors in the second row. (In this example, the $1, 2$, and ∞ -norms all produce a condition number with the same order of magnitude as reported here. Only H_9 and H_{25} are square and different H_k for $k \leq 9$ are used for $d = 9$ and $d = 25$.) While the errors do depend on the function example, the condition numbers do not, so such loss of significance is perhaps inevitable. By postponing any conversion to doubles until after perfect rationals in H_k, it seems that any implementation of GUW implementation using double precision floating point arithmetic will have at least this much error. Indeed, we see more error in the last rows.

One caveat is that more accuracy can be obtained by using the accurate sum and dot product as described in [11]. This technique uses double precision computation to effectively achieve quadruple precision arithmetic. In MATLAB, switching from intrinsic matrix multiplication to naive loops of arithmetic could destroy the efficiency. However, the technique is very efficiently implemented in C for cross-derivatives in [6]. Cross-derivatives have at most one differentiation in each variable, so that there is only one highest order $d = n$ cross-derivative. With a comparison goal similar to this paper, but for this restricted structure, [6] shows that interpolation can be more efficient than a direct method for $d = n \geq 15$, although accurate-arithmetic errors still grow to around 10^{-12} for $d = n = 19$.

2.2 *Matrix analysis of nested interpolation error*

Nested interpolation is amenable to similar analysis using matrices M_k that originate from using directions $\mathbf{r}_j = (1, \mathbf{q}_j)$ for $j = 1, \ldots, m_k$. Using multivariate series (1) to find univariate:

$$\begin{aligned} f(\mathbf{a} + t\mathbf{r}_j) &= \sum_{\mathbf{i}} F(\mathbf{i})(t(1, \mathbf{q}_j))^{\mathbf{i}} \\ &= \sum_{\mathbf{i}} \mathbf{q}_j^{(i_2,\ldots,i_n)} F(\mathbf{i}) t^{|\mathbf{i}|}. \end{aligned}$$

Thus, univariate series coefficients in (2) are given by

$$\mathbf{u}_j(k) = \sum_{|\mathbf{i}|=k} \mathbf{q}_j^{(i_2,\ldots,i_n)} F(\mathbf{i}). \tag{5}$$

Defining the $m_k \times m_k$ matrix $M_k = [\mathbf{q}_j^{(i_2,\ldots,i_n)} : j = 1, \ldots, m_k \text{ and } |\mathbf{i}| = k]$, yields $M_k [\mathbf{F}(\mathbf{i}) : |\mathbf{i}| = k] = [\mathbf{u}_j(k) : \ j = 1, \ldots, m_k]$ so that nested interpolation amounts to solving a linear system with this matrix. We consider three solution methods. The Nested LU method saves global L_k and U_k matrix decompositions of each M_k, enabling the efficient and reliable triangular matrix solution method. (Unlike H_k, these global matrices can be stored up to say $d = 25$ and reused for smaller computations such as $d = 9$.) The Nested Div-Diff method does not store large global arrays but recognizes (5) as a multivariate polynomial interpolation (powers of $\mathbf{q}_j$) and uses divided-differences to compute coefficients of a Newton form of the polynomial which are then gathered to find the $\mathbf{F}(\mathbf{i})$ coefficients. In order to analyze the error, we consider the inverse matrix formulation

$$M_k^{-1}\ [\mathbf{u}_j(k) : \text{directions } \mathbf{r}_j = (1, \mathbf{q}_j) \text{ for } j = 1, \ldots, m_k] = [\mathbf{F}(\mathbf{i}) : |\mathbf{i}| = k]\,.$$

As before, we use *Mathematica* to compute three different exact rational or numerical double results, all assuming exact values for the univariate $\mathbf{u}_j(k)$ values. (Since $n = 2$ in our simple example, $f((0,0) + t(1, w_j)) = \exp((1 + w_j)t)$ and $\mathbf{u}_j(k) = (1 + w_j)^k/k!$, where $w = 0, 1, -1, 1/2, -1/2, 1/4, -1/4, 3/4, -3/4, 1/8, \ldots$. The w sequence was chosen in hopes of keeping condition number down in general.) Perfect M_k^{-1} will denote the exact rational matrix; then $N[M_k^{-1}]$ will denote conversion of this rational inverse to numerical doubles; finally, `InvNMk` will denote converting M_k to doubles before inverting. After multiplying times $[\mathbf{u}_j(k)]$ to get $[\mathbf{F}(\mathbf{i}) : |\mathbf{i}| = k]$, we multiply by $\mathbf{i}!$ to compute $D_{\mathbf{i}}(f)$ which should be 1. The absolute error $|D_{\mathbf{i}}(f) - 1|$ is (theoretically and numerically) the same as the relative error in $\mathbf{F}(\mathbf{i})$. The resulting maximum absolute errors are in Table 3. The first row verifies correct implementation of this perfect method in the absence

Table 3. Maximum absolute errors solving $M_k\,[\mathbf{F}(\mathbf{i})] = [\mathbf{u}_j(k)]$ to compute derivatives of $\exp(x + y)$ at (0,0).

Same *Mathematica* using	$d = 9$ error	$d = 25$ error
M_k^{-1} exact rationals	0	0
$N[M_k^{-1}]$ exact to double	9.8e-13	7.0e-03
`InvNMk` inverse of doubles	1.2e-11	2.4e+02
Our MATLAB Nested LU	1.3e-12	2.3e+01
Our MATLAB Nested Div-Diff	5.5e-11	1.4e+02

of roundoff. The last three rows show that floating point computations throughout the different matrix solve methods all result in about the same error, with LU being just slightly better. Again, the second row further isolates the problem to the matrix, not implementation details or function examples.

As before, the second row in Table 3 is explained by condition number. The condition numbers $\text{cond}(M_9) \approx 10^4$ and $\text{cond}(M_{25}) \approx 10^{13}$ when multiplied by machine epsilon around 10^{-16} are close to the errors in the second row. (In this example, we report the 2-norm condition number; 1 and ∞ -norms are just enough to round to one higher order of magnitude.) While the errors do depend on the function example, the condition numbers do not, so such loss of significance is to be always expected. By postponing any conversion to doubles until after perfect rationals in M_k^{-1}, it appears that any implementation of Nested interpolation using double precision floating point arithmetic will have at least this much error.

3. Comparing Efficiency

The interpolation methods were invented for efficiency, with fewer flops than forward multivariate AD, but in our MATLAB implementations the Forward Corner program is very competitive, and often the fastest. Such comparison may only be relevant for an interpreted language using efficient matrix and vector primitives for many of the operations, and for reasonably sized problems with relatively efficient random (indirect) access to the arrays. We desire high-order d and report results only for problems with less than 15,000 entries in the corner to be computed, so we are restricted to $n \leq 8$ for $d \geq 8$. We can handle up to 1 million entries in Forward Corner implementation, but are more limited in order to complete comparative computation of all methods and variable precision for true values, in particular. Some AD tools, such as ADOL-C, are much faster than timings reported here for MATLAB and can handle larger problems, by using a compiled language and simultaneous propagation of univariate Taylor coefficients. Still, it is interesting to compare all the methods in one language on the same platform.

By going out to the 25th derivative in the tennis serve function, the theoretical advantage of interpolation showed slightly in the timings of Table 4. All 3,276 partial derivatives are computed within the evaluation time in the first row of the table, assuming that global reference arrays exist for $n = 3$ and $d = 25$. (It has been reported that ADOL-C takes only 0.006 seconds compared to the 0.52 seconds in row one of Table 4 using C on a different platform, so clearly these timings are only relative to our straight-forward implementations in MATLAB on this laptop.) These global arrays are H_k matrices for GUW Interpolation (section 2.1) and the L_k and U_k matrices for Nested LU (section 2.2). The global array for Forward Corner is the sub-box indexing array described in section 4. The Nested Div-Diff method does not use a large global array but pays for this in the Evaluation time. These global arrays are computed once and permanently saved for later derivative or series computation to 25th order for any function of three variables. In the last two rows of Table 4, the one-time cost of globals can be justified assuming multiple runs, and is relatively insignificant for Forward Corner. Size could be an issue for large problems, though all except Div-Diff are of a similar order of magnitude in MB, which is a couple or orders of magnitude bigger than the `0.025` MB space taken by one corner array of double values (size of the answer), in this case. Actually, the Nested LU space only accounts for L_k matrices but, since they are stored as square matrices, the U_k matrices could be stored in the same space.

Table 4. Efficiency of tennis serve function for $n = 3$ and $d = 25$.

	Forward Corner	GUW Interpol.	Nested Div-Diff	Nested LU
Evaluation time (sec)	0.55	0.52	1.18	0.48
One-time globals (sec)	0.31	9.47	0.0008	2.75
Largest global (MB)	5.96	8.77	0.005	5.46

Dimensions $n = 8$ and $d = 8$, and more operations, are tested by finding derivatives at $(1, 2, \frac{1}{2}, 3, \frac{1}{3}, 4, \frac{1}{4}, 5)$ for the following damped oscillation:

$$t = x_1^2 + 2x_2^2 + 3x_3^2 + 4x_4^2 + 5x_5^2 + 6x_6^2 + 7x_7^2 + 8x_8^2,$$
$$f(\mathbf{x}) = \exp\left(-\sqrt{t}\right) \sin(t \ln(1 + t)).$$

Timing results in Table 5 show much more efficiency for the Forward Corner. In this case, one corner array of 12,870 double values (size of the answer) is `0.098` MB. The global arrays for interpolation are much larger and do not pay off. For this example, the maximum relative error was about 10^{-11} for Forward Corner and 10^{-3} for the interpolation methods. The maximum error over maximum value for fixed order 8 was a respectable 10^{-14} for Forward Corner and 10^{-13} for the interpolation methods (compare Figure 1).

The evaluation times in Tables 4 and 5 are somewhat surprising, given theoretical flop comparisons which say that interpolation methods should beat Forward Corner, with the fastest being Nested Div-Diff, then Nested LU, then GUW Interpolation. Interpolation has two phases, first computing univariate series in each of the directions and then reconstructing the multivariate values (using the global arrays). The first univariate series

Table 5. Efficiency of the damped oscillation function for $n = 8$ and $d = 8$.

	Forward Corner	GUW Interpol.	Nested Div-Diff	Nested LU
Evaluation time (sec)	1.65	14.29	12.25	11.64
One-time globals (sec)	0.79	312.07	0.0002	218.38
Largest global (MB)	6.99	631.81	0.0005	433.99

phase is the same time cost for all the interpolation methods. This cost is proportional to the number of operations (*, /, transcendental) in the function being differentiated. For each operation, this cost is only a fraction $q(d, n)$ (defined in [4]) of the cost for such an operation on a Forward Corner, with $q(25, 3) \approx 0.17$ and with $q(8, 8) \approx 0.39$. For the reconstruction phase, the number of nonzeros (and hence required multiplications) in the H_k matrices are roughly ($1\times$ for $(25, 3)$ and $5\times$ for $(8, 8)$) the number multiplications for one Forward Corner operation, supposedly making GUW faster. The Nested LU matrices are smaller than the H_k matrices, and the Nested Div-Diff has been shown to have even fewer multiplications and divisions including Newton and collect terms phases [8]. However, our implementations use full matrix storage and operations in MATLAB and the nested loops in Div-Diff take their toll.

For better efficiency comparisons in the future, all these methods should be implemented in a compiled language, handling sparsity and taking advantage of simultaneous propagation of univariate Taylor series in the needed directions. Still the interpolation error problem persists. The accuracy of the Forward Corner program, and its efficiency at least in MATLAB, make it worth considering how to implement the algorithm.

4. Forward Corner index reference scheme

The Forward Corner program relies on an efficient multiplication of corners of Taylor series coefficients, as in $h = u * v$ where known series coefficients for u and v are combined to compute those for h. All standard transcendental functions $h = f(u(\mathbf{x}))$ are then given by one or two differential equations of the form $h_{x_i} = v * u_{x_i}$, where the subscript is a partial derivative, so that a similar multiplication of corners performs a recurrence relation for the function. Details of every multivariate Taylor recurrence relation for the standard functions and operations are found in [10]. A conceptual geometric understanding of the n-dimensional relationships shows what must be accomplished. For a simple multiplication of two corners, the resulting product value at an index $\mathbf{i}$ is given by pulling out the sub-box of all coefficients with index $\mathbf{j} \leq \mathbf{i}$ componentwise from corners for u and v and doing a dot product of the two sub-boxes, but with one in reverse order. The analogous operation for the differential equation version strips off one (opposite) face of each box and dots parallel slices of each, also times the index of that slice. This section will clarify these operations and describe implementation. The key idea is that we can store a 2-dimensional reference matrix for each multi-index, where each row of the 2D matrix contains addresses for the elements in each slice of the corresponding sub-box.

For practicality, all corners of Taylor coefficients are stored in a linear array corresponding to increasing order $|\mathbf{i}|$, and in reverse lexical ordering of multi-indices within the same order. A simple algorithm for incrementing from one multi-index to the next is Algorithm 2 in [8]. It is convenient to save a, relatively small, global table of the multi-index $\mathbf{i}$ for each linear index i, to interpret results if nothing else. This table works like a vector valued function $\text{multi}(i) = \mathbf{i}$. We avoid mapping in the other direction which requires a search or binomial formulas. Algorithms for operations on corners will loop through each linear index, assuming all corner values with previous linear indices, including all terms of lower order, have been computed.

Let's illustrate with specifics in an example. We focus on $n = 3$ variables and multi-index $(3, 0, 2)$ which occurs at linear index 41 for any $d \geq 5$. Consider multiplying $h(x, y, z) = u(x, y, z) * v(x, y, z)$, where u and v have previously computed corners of Taylor series coefficients, and suppose you want to compute the coefficient $\mathbf{H}((3, 0, 2))$ of $(x - a_1)^3(y - a_2)^0(z - a_3)^2$ in the product h. In the corners for u and v, contributions come only from terms that correspond to the sub-box of multi-indices $\leq (3, 0, 2)$ componentwise, and these are multiplied by the complementary term in the other sub-box. The sub-box of indices are shown in Table 6. In general, for reasons to be explained in

1 (0,0,0)	4 (0,0,1)	10 (0,0,2)
2 (1,0,0)	7 (1,0,1)	16 (1,0,2)
5 (2,0,0)	13 (2,0,1)	26 (2,0,2)
11 (3,0,0)	23 (3,0,1)	41 (3,0,2)

Table 6. Linear indices and multi-indices in the sub-box for multi-index (3,0,2).

the next paragraph, each sub-box of indices (of any dimension) is organized into a 2D reference matrix M_i where indices are placed into rows according to a *privileged coordinate* defined as the (first of) smallest nonzero coordinate in the multi-index $\mathbf{i}$. For each $\mathbf{j} \leq \mathbf{i}$, the linear indices j are listed in rows corresponding to values of this privileged coordinate in $\mathbf{j}$, with increasing linear index within each row. For $\mathbf{i} = (3, 0, 2)$, the third coordinate is privileged, so that this case is just a transpose of Table 6:

$$M_{41} = \begin{bmatrix} 1 & 2 & 5 & 11 \\ 4 & 7 & 13 & 23 \\ 10 & 16 & 26 & 41 \end{bmatrix}.$$

The M_i are always 2D matrices of different sizes. (In our MATLAB implementation, they are in a cell array, so M_i is `linearsliceS{i}`.) For simple multiplication of corners, we really need only a 1D list of these sub-box entry addresses, and any raveling of M_{41} (by rows, columns, or increasing linear index) into one row will match appropriate complements in the following pseudocode:

$$\text{boxi} = \text{ravel}(M_{41})$$
$$h\,[41] = u\,[\text{boxi}] * v\,[\text{reverse}(\text{boxi})]^T.$$

Here, $h\,[41]$ indexes into the desired value in the corner of Taylor series coefficients for h and indexing in with vectors returns vectors of u and v coefficients. In MATLAB, ravel can be accomplished by `(:)'` and reverse can be accomplished by `fliplr`.

For transcendental functions, we need the 2D matrices. Consider propagating series through $h(x, y, z) = \exp(u(x, y, z))$. To compute $h\,[41] = \mathbf{H}((3, 0, 2))$, we isolate the variable z corresponding to the privileged coordinate and use $h_z = u_z * h$. Coefficients of u_z come from the Taylor series coefficients for u, adjusted for the power of z, given by the row of M_{41}. First row coefficients are are dropped, since terms are constant with respect to z; the last row coefficients, corresponding to quadratic terms of u, are multiplied by

2. Using MATLAB notation for rows,

$$h[41] = \frac{1}{2}\left(\begin{array}{c} 1 * u[M_{41}(2,:)] * h[\text{reverse}(M_{41}(2,:))]^T \\ +2 * u[M_{41}(3,:)] * h[\text{reverse}(M_{41}(1,:))]^T \end{array}\right). \tag{6}$$

In this formula, the u coefficients times row-number correspond to the coefficients of u_z up to multi-index $(3,0,1)$. Combining with these known complementary h coefficients results in the coefficient for h_z at multi-index $(3,0,1)$. Finally, integration with respect to z will simply divide by 2 to give the coefficient for h at $(3,0,2)$ as in (6). Mathematically, $h_x = u_x * h$ would work just as well, so the privileged coordinate could be from any nonzero in $\mathbf{i}$. However, choosing x would mean dropping the first row of Table 6 and result in 9 multiplications as opposed to the 8 in (6). Choosing the smallest nonzero reduces the necessary computation for transcendental functions, which curiously is less than required for simple multiplication.

Formula (6) generalizes for any i and M_i (and n) and applies to any derivative relationship $h' = u' * v$. If M_i has $k+1$ rows, the fraction out front is $\frac{1}{k}$, row multipliers are 1 to k for rows 2 to $k+1$ in u, and rows of v are used in reverse order k to 1. This is called a ddot (for derivative dot) operation in [10] and is used to implement every Taylor coefficient recurrence formula for all standard functions. The nonzero privileged coordinate insures $k \geq 1$; the degenerate case of only one nonzero entry in $\mathbf{i}$ makes M_i into a one column matrix (to compute univariate series).

4.1 *Efficient generation of the global matrices of sub-box indices*

Efficient generation of the global array of all M_i matrices is done iteratively by combining previous matrices. This was found to be much, much faster than directly searching through a listing of all multi-indices and their addresses for the needed $\mathbf{j} \leq \mathbf{i}$. When the program generating M_i reaches linear index $i = 41$, the formula (7) explained below returns the addresses $d_1 = 26$ and $d_3 = 23$ (above and to the left in Table 6) corresponding to reducing the multi-index $\mathbf{i} = (3,0,2)$ by one separately in each respective nonzero entry. Then M_{41} is a kind of union of the entries from

$$M_{26} = \begin{bmatrix} 1 & 4 & 10 \\ 2 & 7 & 16 \\ 5 & 13 & 26 \end{bmatrix} \text{ and } M_{23} = \begin{bmatrix} 1 & 2 & 5 & 11 \\ 4 & 7 & 13 & 23 \end{bmatrix}.$$

The sorted union (see `union` or `unique` in MATLAB) of entries in these matrices (up to n of them in general) can be loaded into the appropriate row of M_{41} by looking up the privileged (third in this example) coordinate in multi(j) corresponding to each linear index entry j in the union. Actually, we can build on M_{23} since it corresponds to multi-index $(3,0,1)$, down one in the (same) privileged coordinate. Then other addresses with privileged coordinate value 2, are augmented as a new row to make M_{41} which ends with 41. One caveat if $\mathbf{i}$ has 1 in some coordinate (which is then privileged): the matrix you build on is first flattened into one row. For example, consider forming M_{65} corresponding to multi-index $\mathbf{i} = (3,1,2)$. Formula (7) returns linear indices $d_1 = 44$, $d_2 = 41$, and $d_3 = 40$, corresponding to separately down one from $\mathbf{i}$ in each coordinate left to right. Since the second coordinate is privileged in this case, we build on M_{41} but, since all entries have second coordinate zero, this is sorted into the first row of

$$M_{65} = \begin{bmatrix} 1 & 2 & 4 & 5 & 7 & 10 & 11 & 13 & 16 & 23 & 29 & 41 \\ 3 & 6 & 9 & 12 & 15 & 19 & 22 & 23 & 29 & 40 & 44 & 65 \end{bmatrix}$$

and the last row is the new entries from M_{44} and M_{40} with 65 at the end. Such two-row matrices are efficiently used, since computing as in (6) would only require

$$h[65] = u[M_{65}(2,:)] * h[\text{reverse}(M_{65}(1,:))]^T.$$

We now establish the formula for linear addresses corresponding to multi-indices that are down one in a coordinate from $\text{multi}(i) = \mathbf{i}$. Let $\mathbf{i}(k)$ denote the kth coordinate of $\mathbf{i}$. For n variables, we seek linear indices $d_1, d_2, \ldots, d_n$ where $\text{multi}(d_k)$ is the same as $\mathbf{i}$ except that the kth coordinate is $\mathbf{i}(k) - 1$, assuming $\mathbf{i}(k) \neq 0$. If $\mathbf{i}(k) = 0$, d_k is ignored except as needed to compute the other d_j. Sum coordinates of $\mathbf{i}$ starting at k and define $s_k = \mathbf{i}(k) + \mathbf{i}(k+1) + \cdots + \mathbf{i}(n) - 1$. Let $d_0 = i$, then we claim that

$$d_k = d_{k-1} - \binom{n-k+s_k}{n-k}. \tag{7}$$

The last two values reduce to $d_{n-1} = d_{n-2} - (\mathbf{i}(n-1) + \mathbf{i}(n))$ and $d_n = d_{n-1} - 1$. In general, the binomial coefficient is the cardinality of the set S of multi-indices in $n-k$ coordinates from order zero up through order s_k, as can be seen by an s_k stars and $n-k$ bars argument. Assuming $\mathbf{i}(k) \neq 0$, we justify (7) by taking each element of S and tacking some k coordinates on the front to form all multi-indices between $\text{multi}(d_k)$ and the later $\text{multi}(d_{k-1})$ in our ordering. Recall the ordering is by order first, then reverse lexical within order. Split S into two parts with the upper part starting at the last $n-k$ coordinates of $\text{multi}(d_k)$, and the lower part of preceding multi-indices. In front of each multi-index in the upper part, put the first $k-1$ coordinates of $\text{multi}(d_k)$ and whatever kth coordinate is needed to keep order $|\mathbf{i}| - 1$. This kth coordinate is nonincreasing as S progresses, resulting in all mutli-indices between $\text{multi}(d_k)$ and a multi-index with the same first $k-1$ coordinates but then all zeros except for s_k in the last coordinate. If $k > 1$, the next multi-index changes the first $k-1$ coordinates to those of $\text{multi}(d_{k-1})$ and follows with s_k+1 and then all zeros. From here we may let the last $n-k$ coordinates increase through the lower part of the ordering of S, keeping order $|\mathbf{i}|-1$ using the kth coordinate. The next multi-index is $\text{multi}(d_{k-1})$, which has the same last $n-k$ coordinates as $\text{multi}(d_k)$, so we used every element of S exactly once to get from one to the other. If $k = 1$, the next mult-index after all zeros then s_1 is $s_1 + 1 = |\mathbf{i}|$ then all zeros, so the lower part of S is used by keeping the higher order $|\mathbf{i}|$ with the first coordinate until the process ends at $\mathbf{i} = \text{multi}(d_0)$.

Let's summarize using this notation. To form M_i, the union is over $\{M_{d_k} : \mathbf{i}(k) \neq 0\}$, organized in rows according to the privileged coordinate. If p is the privileged coordinate, M_i can build on M_{d_p}, literally adding one more row if this smallest nonzero $\mathbf{i}(p) > 1$ or flattening M_{d_p} into one row and adding a second if $\mathbf{i}(p) = 1$. If $\mathbf{i}(k) = 0$, then d_k is ignored, $\text{multi}(d_k)$ and M_{d_k} exist but do not have useful interpretations. Still, d_k successfully leads to useful values via (7), even though the argument of the previous paragraph does not work directly.

5. Conclusion

To accurately compute all high-order partial derivatives or multivariate Taylor series coefficients, forward multivariate AD succeeds where the interpolation methods have limitation, especially beyond order ten. Much of this error, for both interpolation direction schemes, comes from roundoff in the matrix that describe the interpolation method, as confirmed using *Mathematica.* This error corresponds to the condition number of the matrix, indicating that the difficulty is not just implementation but that more accurate high-order interpolation performance would require a new direction scheme with lower condition numbers. The problem is not quite so dire if the goal is to accurately compute a multivariate Taylor polynomial value, since many of the high-order coefficients don't matter (so maybe shouldn't be computed) for fast convergence near the center. However, when further from the center, the error from interpolation methods can result in a significant polynomial value error that is larger than the truncation error for that point. In most trials, the nested interpolation methods did give a more accurate polynomial value than the GUW interpolation, sometimes rivaling the accuracy of forward multivariate AD which was always reliable.

Theoretically, the interpolation methods are more efficient but, surprisingly, this was not observed in our implementations in MATLAB. Usually the Forward Corner program was significantly faster using similar or smaller space for global reference arrays. Unlike accuracy, this efficiency comparison is closely tied to specific implementations in MATLAB and may not generalize. Even within MATLAB, more interpolation efficiency may be achieved using sparsity and vectorization. In compiled languages, timings might be radically different, as seen in the efficiency of AD Tools that use interpolation. Our Comparison Test Codes are available at http://www.neidinger.net/publicat.html for those wishing to try such improvements in MATLAB or comparisons in other computing environments. The point of our efficiency comparisons is that direct forward multivariate AD can be a reasonable alternative that should be considered, especially in light of the accuracy. To share efficient implementation of the Forward Corner algorithm, we described details concerning the global array of matrices of sub-box indices.

While forward multivariate AD method is certainly the most accurate method, we did see limitation on accuracy relative to each individual derivative value in our first example. Some examples do have inherent limitation, even symbolically as shown in [7], where univariate series values from numerical evaluation of huge symbolic results were less accurate than from numerical series propagation as in AD. Like most numerical methods, we should study and be aware that some examples are sensitive.

5.1 *Acknowledgements*

We thank Andrea Walther for performing and sharing the computation error for the normalized tennis serve function using ADOL-C, and thank Andreas Griewank for the related timing.

References

[1] B. Altman, *Higher-order automatic differentiation of multivariate functions in MATLAB*, Undergraduate Honors Thesis, Davidson College, 2010. Available at http://davidson.lyrasistechnology.org/islandora/object/davidson:63066.

[2] M. Berz, *Algorithms for higher derivatives in many variables with applications to beam physics*, in

Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G.F. Corliss, eds., SIAM, Philadelphia, PA, 1991, pp. 147–156.

[3] R.N. Greenwell, N.P. Ritchey, and M.L. Lial, *Calculus for the Life Sciences*, 1st ed., Addison-Wesley, Boston, 2003.

[4] A. Griewank, J. Utke, and A. Walther, *Evaluating higher derivative tensors by forward propagation of univariate Taylor series*, Math. Comp. 69 (2000), pp. 1117–1130.

[5] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed., SIAM, Philadelphia, PA, 2008.

[6] A. Griewank, L. Lehmann, H. Leovey, and M. Zilberman, *Automatic evaluations of cross-derivatives*, Math. Comp. 83 (2014), pp. 251–274.

[7] R.D. Neidinger, *Series as a computational differentiation tool*, Mathematica Educ. Res. 9 (2000), pp. 5-14.

[8] R.D. Neidinger, *Directions for computing truncated multivariate Taylor series*, Math. Comp. 74 (2005), pp. 321–340.

[9] R.D. Neidinger, *Introduction to automatic differentiation and MATLAB object-oriented programming*, SIAM Rev. 52 (2010), pp. 545–563.

[10] R.D. Neidinger, *Efficient recurrence relations for univariate and multivariate Taylor series coefficients*, Discrete Contin. Dyn. Syst. (2013) Dynamical systems, differential equations and applications. 9th AIMS Conference. Suppl., pp. 587–596. Available at http://aimsciences.org/journals/displayPaperPro.jsp?paperID=9241. MR 3462403.

[11] T. Ogita, S.M. Rump, and S. Oishi, *Accurate sum and dot product*, SIAM J. Sci. Comput. 26 (2005), pp. 1955–1988.

[12] I. Tsukanov and M. Hall, *Data structure and algorithms for fast automatic differentiation*, Internat. J. Numer. Methods Engrg. 56 (2003), pp. 1949-1972.