MATHEMATICS OF COMPUTATION
Volume 74, Number 249, Pages 321–340
S 0025-5718(04)01657-6
Article electronically published on May 17, 2004

DIRECTIONS FOR COMPUTING TRUNCATED MULTIVARIATE TAYLOR SERIES

RICHARD D. NEIDINGER

Abstract. Efficient recurrence relations for computing arbitrary-order Taylor coefficients for any univariate function can be directly applied to a function of n variables by fixing a direction in $\mathbb{R}^n$. After a sequence of directions, the multivariate Taylor coefficients or partial derivatives can be reconstructed or "interpolated". The sequence of univariate calculations is more efficient than multivariate methods, although previous work indicates a space cost for this savings and significant cost for the reconstruction. We completely eliminate this space cost and develop a much more efficient algorithm to perform the reconstruction. By appropriate choice of directions, the reconstruction reduces to a sequence of Lagrange polynomial interpolation problems in $\mathbb{R}^{n-1}$ for which a divided difference algorithm computes the coefficients of a Newton form. Another algorithm collects like terms from the Newton form and returns the desired multivariate coefficients.

1. Introduction

Given a real-valued function of n variables, we seek an efficient algorithm to compute all multivariate Taylor series coefficients through arbitrary order d, or equivalently, all partial derivatives through order d at some point. A tenant of automatic differentiation, AD, is that truncated series manipulation algorithms are preferable to symbolic differentiation or numerical approximation of derivatives, having the accuracy of symbolic work but using "numerical" floating-point computations. Univariate truncated series algorithms have been exploited successfully for many years [10]. Extensions to multivariate AD algorithms have been developed, for example [1] and [12]. Unfortunately, the huge number of coefficients and the multi-dimensional nature of the algorithms lead to serious concerns about computational time, storage space and efficient data access. An interpolation scheme was proposed in [2] that would reduce computation and simplify data access by computing univariate Taylor coefficients along fixed directions in n-space and, then, reconstructing the n-dimensional Taylor coefficients. The idea was inspired by a construction of mixed partials from directional series in [13]. The detailed interpolation scheme in [5] achieves time savings but incurs extra space cost in comparison to direct multivariate series algorithms. The interpolation scheme presented herein improves the time savings while eliminating any extra space cost, since the largest array ever constructed is the final array of multivariate Taylor coefficients.

Received by the editor May 28, 2002 and, in revised form, June 10, 2003.
2000 *Mathematics Subject Classification.* Primary 65D25, 65D05, 41A05, 41A63, 65Y20.
Key words and phrases. Automatic differentiation, multivariate, polynomial interpolation, higher-order derivatives, divided difference.

TABLE 1. Supplementary recurrence relations for univariate AD.

Standard function	Consecutive recurrence relations
$u(t) = \tan(b(t))$	$\tilde{u}^{(k)} = \sum_{j=1}^{k} v^{(k-j)} \tilde{b}^{(j)}$ and $\tilde{v}^{(k)} = 2 \sum_{j=1}^{k} u^{(k-j)} \tilde{u}^{(j)}$
$u(t) = \arcsin(b(t))$	$\tilde{u}^{(k)} = \frac{1}{v^{(0)}} \left(\tilde{b}^{(k)} - \sum_{j=1}^{k-1} v^{(k-j)} \tilde{u}^{(j)} \right)$ and $\tilde{v}^{(k)} = - \sum_{j=1}^{k} b^{(k-j)} \tilde{u}^{(j)}$
$u(t) = \arctan(b(t))$	$\tilde{u}^{(k)} = \frac{1}{v^{(0)}} \left(\tilde{b}^{(k)} - \sum_{j=1}^{k-1} v^{(k-j)} \tilde{u}^{(j)} \right)$ and $\tilde{v}^{(k)} = 2 \sum_{j=1}^{k} b^{(k-j)} \tilde{b}^{(j)}$

2. Univariate algorithms

If u is an analytic function of t, we denote the Taylor series about zero as

$$u(t) = u^{(0)} + u^{(1)} t + u^{(2)} t^2 + \cdots + u^{(d)} t^d + \cdots ,$$

where the notation $u^{(k)}$ denotes the kth derivative of u at zero divided by $k!$, rather than simply the derivative.

This paper assumes an efficient method for computing univariate Taylor polynomial coefficients up through degree d for any elementary function. In standard forward automatic differentiation, this can be accomplished by $O\left(d^2\right)$ flops (counting floating-point multiplications or divisions) for each standard function or operation in the function expression (except for addition and scalar multiplication which are trivial in comparison). The list $\{u^{(0)}, u^{(1)}, \ldots, u^{(d)}\}$ of coefficients is computed by beginning with the coefficient list $\{0, 1, 0, \ldots, 0\}$ for the identity t and propagating through the expression. In particular, if $u(t) = u_l(t) * u_r(t)$ and the coefficient lists for u_l and u_r have already been computed, then each $u^{(k)} = \sum_{j=0}^{k} u_l^{(j)} u_r^{(k-j)}$. So, a multiplication in u requires $\binom{d+2}{d} \approx \frac{1}{2} d^2$ flops. The beauty of this approach is that all standard functions and operations can be reformulated as one or two multiplications of the argument(s), the result, or their derivatives. Thus, coefficients can be computed from the general principle: If $u'(t) = a(t) * b'(t)$, then $\tilde{u}^{(k)} = \sum_{j=1}^{k} a^{(k-j)} \tilde{b}^{(j)}$ where $\tilde{u}^{(k)} = k\, u^{(k)}$ and $\tilde{b}^{(j)} = j\, b^{(j)}$. The simplest example is $u(t) = \exp(b(t))$, where $u'(t) = u(t) * b'(t)$ yields the recurrence $\tilde{u}^{(k)} = \sum_{j=1}^{k} u^{(k-j)} \tilde{b}^{(j)}$. Some functions, such as sin and cos, must be computed in pairs and require twice as many, $\approx d^2$, flops.

Specific recurrence relations for multiplication, division, squaring, square root, constant power, ln, exp, sin, and cos are listed in [4, Tables 10.1 and 10.2]. Table 1 supplements these with recurrence relations for tan, arcsin, and arctan, derived as follows. If $u(t) = \tan(b(t))$ and $v(t) = \sec^2(b(t))$, then $u'(t) = v(t) * b'(t)$ and $v'(t) = 2\, u(t) * u'(t)$. These yield the tan recurrences in Table 1, by the general principle from the preceding paragraph. If $u(t) = \arcsin(b(t))$ and $v(t) = \cos(u(t))$, then $b'(t) = v(t) * u'(t)$ and $v'(t) = -b(t) * u'(t)$, and the recurrence relations for arcsin follow. Similarly, if $u(t) = \arctan(b(t))$ and $v(t) = \sec^2(u(t))$, then $b'(t) = v(t) * u'(t)$ and $v'(t) = 2\, b(t) * b'(t)$. This set of recurrence relations will suffice for most elementary function expressions. Such techniques can also be applied to functions described by program code or differential equations.

3. Multivariate notation

Multivariate Taylor series will use multi-indices as in this example.

$$\begin{aligned} f(x,y,z) &= a_{(0,0,0)}1 + a_{(1,0,0)}x + a_{(0,1,0)}y + a_{(0,0,1)}z \\ &\quad + a_{(2,0,0)}x^2 + a_{(1,1,0)}x\,y + a_{(1,0,1)}x\,z \\ &\quad + a_{(0,2,0)}y^2 + a_{(0,1,1)}y\,z + a_{(0,0,2)}z^2 + \cdots . \end{aligned}$$

Define the multi-index set

$$\Gamma^n = \{(\psi_1, \psi_2, \ldots, \psi_n) : \text{each } \psi_i \in \{0, 1, 2, \ldots\}\}.$$

The Taylor series for $f : \mathbb{R}^n \to \mathbb{R}$ is then $f(\vec{x}) = \sum_{\psi\in\Gamma^n} a_\psi \vec{x}^{\,\psi}$ where $\vec{x}^{\,\psi} = x_1^{\psi_1} x_2^{\psi_2} \cdots x_n^{\psi_n}$. We assume multivariate analyticity so that

$$a_\psi = \frac{1}{\psi_1!\psi_2!\cdots\psi_n!}\left[\left(\frac{\partial}{\partial x_1}\right)^{\psi_1}\left(\frac{\partial}{\partial x_2}\right)^{\psi_2}\cdots\left(\frac{\partial}{\partial x_n}\right)^{\psi_n} f\right](0,0,\ldots,0),$$

but we will not use derivatives to compute the coefficients. To the contrary, partial derivative values can be obtained efficiently from the coefficients, the output of the algorithms to follow. Any function can be shifted to achieve a series about (or partial derivatives at) any point and in any set of directions. Suppose $g : \mathbb{R}^m \to \mathbb{R}$, $\vec{x}_0 \in \mathbb{R}^m$, and you are interested in directions $\vec{s}_1, \vec{s}_2, \ldots, \vec{s}_n \in \mathbb{R}^m$ for some $n \le m$. Let S be the $m \times n$ matrix with columns $\vec{s}_i$ and

$$f(\vec{x}) = g(\vec{x}_0 + S\,\vec{x}).$$

Then the coefficients (or partials) for f as introduced above will be the desired coefficients (or partials) for g about (or at) $\vec{x}_0$ in the desired directions.

The *order* of a multi-index is $|\psi| = \psi_1 + \psi_2 + \cdots + \psi_n$ and we define

$$\Gamma_d^n = \{\psi \in \Gamma^n : |\psi| \le d\}.$$

For $f : \mathbb{R}^n \to \mathbb{R}$, the desired set of coefficients $\{a_\psi : \psi \in \Gamma_d^n\}$ will be called a *pyramid*. It contains $\binom{n+d}{d}$ values. Occasionally, we use $n = 10$ and $d = 10$ as a specific reference point to help comprehend the sizes at issue. The $n, d = 10$ pyramid has 184,756 coefficients.

We impose a specific ordering on the multi-indices, so that multi-indices can be used unambiguously as a loop counter in algorithms and as row and column indices in matrices. This ordering will increase with the *order* of a multi-index and have reverse lexical ordering within a fixed order. Algorithm 2 will increment a multi-index through this ordering. The series for $f(x, y, z)$ displayed above shows the ordering for multi-indices with $n = 3$ up to order $d = 2$.

There are multi-dimensional generalizations of the univariate recurrence relations. Recall that multiplying two univariate coefficient lists requires

$$u^{(k)} = \sum_{j=0}^{k} a^{(j)} b^{(k-j)}$$

for each $k \le d$, for a total of $\binom{2+d}{d}$ flops. To multiply two pyramids requires $c_\psi = \sum_{\varphi\le\psi} a_\varphi\, b_{\psi-\varphi}$ where the sum is over $\varphi \le \psi$ coordinatewise for each $|\psi| \le d$, for a total of $\binom{2n+d}{d}$ flops [5]. For $n, d = 10$, this is 30,045,015 flops for every function or operation in the expression. Moreover, much of the effort in implementing such a scheme goes into finding the array addresses of pyramid entries a_φ and $b_{\psi-\varphi}$ [1],

[11], [12], [15], [18]. The interpolation idea will avoid these complexities by doing automatic differentiation only on univariate coefficient lists.

4. The interpolation idea

Rather than use the multivariate function $f : \mathbb{R}^n \to \mathbb{R}$ to compute the pyramid of coefficients, the idea of interpolation ([2]) is to use univariate directional functions of the form $u_\varphi(t) = f(t\,\vec{r}_\varphi)$, compute the univariate coefficient lists and then "interpolate" the multivariate pyramid. With appropriate choice of directions $\{\vec{r}_\varphi \in \mathbb{R}^n : \varphi \in \Gamma^n,\ |\varphi| = d\}$, the univariate coefficient lists $\{\{u_\varphi^{(0)}, u_\varphi^{(1)}, \ldots, u_\varphi^{(d)}\} : \varphi \in \Gamma^n,\ |\varphi| = d\}$ can be used to construct the entire pyramid $\{a_\psi : \psi \in \Gamma^n,\ |\psi| \le d\}$. Our focus will be on the choice of directions that make this possible and make the construction an efficient algorithm.

Just this rough idea shows serious time savings and space cost [5]. Recall that each operation in the multivariate f requires $\binom{2n+d}{d}$ flops, whereas the univariate operation requires $\binom{2+d}{d}$ flops for each of the $\binom{n+d-1}{d}$ multi-indices of order d. The total $\binom{2+d}{d}\binom{n+d-1}{d} \le \binom{2n+d}{d}$ for $d > 4$ (always assuming $n > 1$); in the $n, d = 10$ case, this inequality is $6,096,948 < 30,045,015$ (flops for each operation in f) for a cost ratio of about $1/5$. Analysis of this ratio in [5] shows that it decreases exponentially with d. This savings does not even take into account the savings in memory management which may make it more efficient, even for $2 \le d \le 4$ where the ratio ranges from $3/2$ to $15/16$. There is an apparent space cost for this time savings, since each of the $\binom{n+d-1}{d}$ univariate coefficient lists has length $d+1$. Saving the identical constant term only once, this requires $1 + d\binom{n+d-1}{d}$ reals. Compared to the pyramid, $1 + d\binom{n+d-1}{d} \ge \binom{n+d}{d}$ for all n and d. In the $n, d = 10$ case, $923,781 > 184,756$. We will completely eliminate this space cost.

The basic relationship between a_ψ and u_φ is found in

$$u_\varphi(t) = f(t\,\vec{r}_\varphi) = \sum_{\psi \in \Gamma^n} a_\psi (t\,\vec{r}_\varphi)^\psi = \sum_{\psi \in \Gamma^n} a_\psi\ (\vec{r}_\varphi)^\psi\ t^{|\psi|}.$$

Since $u_\varphi(t) = \sum_{k=0}^\infty u_\varphi^{(k)} t^k$, we conclude that

$$\sum_{|\psi|=k} a_\psi\ (\vec{r}_\varphi)^\psi = u_\varphi^{(k)}. \tag{4.1}$$

The construction in [5] chooses the directions $\vec{r}_\varphi = \varphi$ (although, we will choose $\vec{r}_\varphi = (1, w_{\varphi_2}, w_{\varphi_3}, \ldots w_{\varphi_n})$ as described later). In the simple $n = 2$, $d = 4$ case, Figure 1 shows the desired multivariate coefficients and the directions from [5] labelled with corresponding univariate coefficients. Considering order 3 in Figure 1, the four multivariate coefficients marked with a triangle are related to the five univariate coefficients $u_\varphi^{(3)}$ by equation (4.1), which can be interpreted as a 5×4 matrix G_3. The method in [5] may be viewed as finding a 4×5 left-inverse of G_3, thereby mapping the univariate to multivariate coefficients. This process was called interpolation for conceptual reasons, rather than literal polynomial fitting. For orders $1 \le k \le 4$ in Figure 1, this method uses 20 univariate coefficients to compute 14 multivariate coefficients, plus one trivial case $a_{(0,0)} = u_\varphi^{(0)}$. (Actually, many of the 20 terms are not really necessary in this simple case, but, in general, such vanishing terms cannot easily be anticipated, so [5] only makes an exception for the trivial order 0.)

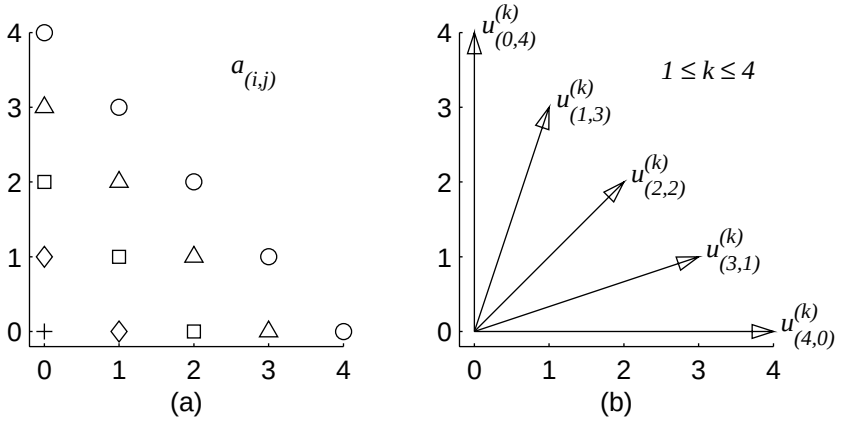

FIGURE 1. $n = 2$, $d = 4$; (a) pyramid of 15 multivariate coefficients reconstructed from (b) 21 univariate coefficients using directions from [5].

For general n and d, the nonsquare matrix $G_k = \left[\varphi^\psi : |\varphi| = d, |\psi| = k\right]$ has row index φ and column index ψ. The left-inverse gives $a_\psi = \sum_{|\varphi|=d} c_{\psi,\varphi}\, u^{(k)}_\varphi$ for each $|\psi| = k$. Specifically, the "main result" of [5] is a formula for $c_{\psi,\varphi}$. The space cost comes from the nonsquare requirement of $|\varphi| = d$ inputs to get $|\psi| = k$ outputs. Although many of the $c_{\psi,\varphi}$ may be zero (not known a priori), there are more nonvanishing terms than multivariate results. For example, with $n, d = 3$ and $k = 2$, the formula uses 9 out of 10 values in $\{u^{(2)}_\varphi : |\varphi| = 3\}$ to compute the 6 values in $\{a_\psi : |\psi| = 2\}$. An attempt to make this into a square matrix problem by truncating G_k and $\left(u^{(k)}_\varphi\right)$ at $\binom{n+k-1}{k}$ rows will find some of the truncated G_k matrices to be singular. While G_d is invertible, it is not clear if any ordering of the vectors $\vec{r}_\varphi = \varphi$ will create a sequence of invertible matrices. Our choice of directions will create a sequence of square invertible matrices and a more efficient algorithm than the formula for $c_{\psi,\varphi}$ in [5].

Another problem with the choice $\vec{r}_\varphi = \varphi$ is that it is not nested; i.e., if the number of variables n remains the same but a higher order d is required, $\{\vec{r}_\varphi : |\varphi| = d\} \not\subseteq \{\vec{r}_\varphi : |\varphi| = d+1\}$. For example, changing Figure 1 to order $d = 5$ would create all new directions except for the axes.

5. Using interpolation in $n-1$ dimensions

There is a one-to-one correspondence between $\{\psi \in \Gamma^n : |\psi| = k\}$ and $\Gamma^{n-1}_k = \{\beta \in \Gamma^{n-1} : |\beta| \le k\}$. The map from ψ to β simply drops the first entry. The inverse map is given by $T_k(\beta) = (k - |\beta|, \beta_1, \beta_2, \ldots, \beta_{n-1})$. The remainder of the paper will use multi-indices in Γ^{n-1} and denote them by Greek letters, other than ψ and φ.

Let $(w_i : i = 0, 1, 2, \ldots)$ be any sequence of distinct real numbers. For simplicity, we assume $w_0 = 0$. Define $\vec{w}_\alpha = (w_{\alpha_1}, w_{\alpha_2}, \ldots, w_{\alpha_{n-1}}) \in \mathbb{R}^{n-1}$. For $\alpha \in \Gamma^{n-1}$, we redefine $\vec{r}_\alpha = (1, w_{\alpha_1}, w_{\alpha_2}, \ldots, w_{\alpha_{n-1}}) \in \mathbb{R}^n$ and univariate functions $u_\alpha(t) = f(t\,\vec{r}_\alpha)$. In the simple $n = 2$, $d = 4$ case, Figure 2 shows these directions for arbitrary w_i and for a specific choice. In this case, the four 3rd-order multivariate coefficients

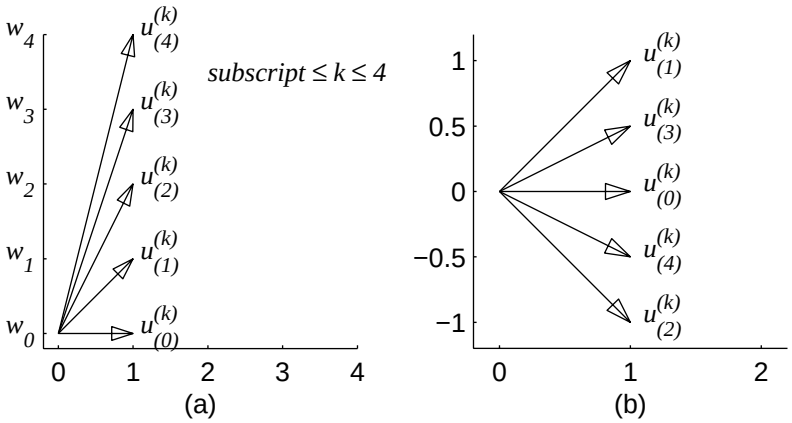

FIGURE 2. $n = 2$, $d = 4$; (a) New directions using 15 univariate coefficients for 1-dimensional interpolation; (b) Physical directions from alternating dyadic rational w_i.

marked with a triangle in Figure 1(a) can be recovered from the four $(0 \le i \le 3)$ 3rd-order univariate coefficients $u^{(3)}_{(i)}$ in Figure 2. Here a literal interpolation of these four univariate values at the nodes w_0, w_1, w_2, w_3 yields a polynomial in one variable y. If this polynomial is in standard form, the coefficients of powers of y are the desired multivariate coefficients $a_{(3,0)}, a_{(2,1)}, a_{(1,2)}, a_{(0,3)}$. Of course, we can first find the polynomial in Newton form by the classic divided-difference algorithm. A higher-dimensional example, for $n = 3$ and $d = 5$, can be visualized using the nodes in Figure 3(b) of Section 6. These nodes are terminal points in the $x = 1$ plane for direction arrows that originate at (0,0,0). The actual interpolation takes place entirely in two dimensions and results in the 3-dimensional pyramid of multivariate Taylor coefficients.

For general n and d, we use directions $\{\vec{r}_\alpha : \alpha \in \Gamma^{n-1}, |\alpha| \le d\}$, a set that is clearly nested as d increases. The basic relationship in equation (4.1) becomes

$$\sum_{\beta \in \Gamma^{n-1}_k} a_{T_k(\beta)} (\vec{w}_\alpha)^\beta = u^{(k)}_\alpha. \tag{5.1}$$

We use an equal number of univariate values $\{u^{(k)}_\alpha : \alpha \in \Gamma^{n-1}_k, k = 0, 1, \ldots, d\}$ to compute the pyramid $\{a_{T_k(\beta)} : \beta \in \Gamma^{n-1}_k, k = 0, 1, \ldots, d\} = \{a_\psi : \psi \in \Gamma^n_d\}$. Define the square matrix

$$M_k = \left[(\vec{w}_\alpha)^\beta : \alpha, \beta \in \Gamma^{n-1}_k \right]$$

with row index α and column index β. For each k, (5.1) becomes $M_k \left(a_{T_k(\beta)} \right) = \left(u^{(k)}_\alpha \right)$ with the one-column matrices indexed by $\beta, \alpha \in \Gamma^{n-1}_k$, where $\left(a_{T_k(\beta)} \right) = (a_\psi : \psi \in \Gamma^n, |\psi| = k)$. We will show that each M_k is invertible, so that one version of the interpolation algorithm is simply $\left(a_{T_k(\beta)} \right) = M_k^{-1} \left(u^{(k)}_\alpha \right)$. Since M_k does not depend on f or d, one could "once and for all" invert M_k and store the M_k^{-1} matrix entries, for a fixed number of variables n. However, these huge matrices would destroy our space savings. We will develop efficient algorithms, DivDiff and CollectTerms, to perform M_k^{-1} times a column without actually inverting or storing

a matrix. For now, simply define

$$\text{CollectTerms(DivDiff}(\left(u_\alpha^{(k)}\right)))=M_k^{-1}\left(u_\alpha^{(k)}\right).$$

With this understanding, the following algorithm produces the time savings of interpolation while using space in such a way that the largest array ever created is the final pyramid of Taylor coefficients. For each k, we use an array V_k to hold $\left(u_\alpha^{(k)}:\alpha\in\Gamma_k^{n-1}\right)$ of length $\binom{n+k-1}{k}$. If all of these arrays are statically allocated, the space exactly corresponds to the size of the pyramid. As soon as V_k is complete (all directions using $|\alpha|\le k$ have been differentiated), we can use and overwrite these values with the multivariate Taylor coefficients of order k. In fact, the algorithms DivDiff and CollectTerms will make the changes in place. We will iterate through each α, overwriting the univariate array U on each step and only saving those values of higher order. Suppose we reach the last α of order k; all values of order $<k$ are complete and no longer needed and only those univariate values for $|\alpha|\le k$ have been computed. Thus, the necessary storage at this stage is for values $\{V_j(\alpha):k\le j\le d,|\alpha|\le k\}$, a rectangle of size $(d-k+1)\binom{n+k-1}{k}$ which must be smaller than the pyramid. In the algorithm, dynamic allocation is shown to maintain this rectangle of storage. In the $n,d=10$ case, the largest needed rectangle occurs when $k=9$ and requires $10/19$ of the pyramid space. If n is much larger than d, then the size of V_d dominates the total of all others, so the dynamic strategy does not save much proportionally.

Algorithm 1. Multivariate Taylor coefficients

INPUT: degree d and function $f:\mathbb{R}^n\to\mathbb{R}$ (or more generally, g, $\vec{x}_0$, S).
GLOBAL: sequence (w_i) for node coordinates.
OUTPUT: array $(a_\psi:\psi\in\Gamma_d^n)$ of Taylor coefficients.
Initialize each V_j, for $j=0$ to d, to be an empty 1-dimensional array;
For $k=0$ to d do
  For each $\alpha\in\Gamma^{n-1}$ with $|\alpha|=k$ do
    Use univariate automatic differentiation to order d at $t=0$ on:
      $u_\alpha(t)=f(t\,(1,w_{\alpha_1},w_{\alpha_2},\ldots,w_{\alpha_{n-1}}))$ or
        $=g(\vec{x}_0+t\,S(1,w_{\alpha_1},w_{\alpha_2},\ldots,w_{\alpha_{n-1}}))$
      which returns $U=\left(u_\alpha^{(0)},u_\alpha^{(1)},\ldots,u_\alpha^{(k)},\ldots u_\alpha^{(d)}\right)$;
    For $j=k$ to d do
      $V_j=\text{Append}(V_j,\ u_\alpha^{(j)})$;
    End j;
  End For α;
  $V_k=\text{CollectTerms(DivDiff}(\ V_k\ ))$;
  Output V_k; Deallocate V_k; // or save to Return below
End For k;
Return $\{V_k:k=0,\ldots,d\}$ // or Output above.

The α loop in Algorithm 1 iterates through a bundle of directions corresponding to all order k multi-indices in $n-1$ coordinates. This can take advantage of cached function evaluations in the automatic differentiation, before the interpolation process that follows the α loop. It is trivial to iterate through an integer index corresponding to α. However, one does need the actual multi-index α to specify the direction. If the loop only keeps track of the current multi-index value, the

following algorithm shows how to increment to the next α value. This will take some time, though the loops might be empty. If time is more precious than space, something like this could be done once and stored in a reference array of all multi-indices in Γ_d^{n-1}. The algorithm handles any number of coordinates m, so that it can be applied now for $m = n-1$ and more generally in Algorithm 5.

Algorithm 2. Increment
INPUT: positive integer m and $\alpha \in \Gamma^m$.
OUTPUT: the next multi-index after α in our ordering of Γ^m.
$i = m-1$;
While ($i > 0$ and $\alpha_i = 0$) do $i = i-1$;
If $i > 0$ then $\alpha_i = \alpha_i - 1$;
$\alpha_{i+1} = \alpha_m + 1$;
For $j = i+2$ to m do $\alpha_j = 0$;
Return α.

We now return to why M_k is invertible and how to perform the inverse. By choosing 1 as the first coordinate of each $\vec{r}_\alpha$, the problem is reduced to a Lagrange polynomial interpolation problem in $\mathbb{R}^{n-1}$. In this context, Lagrange does not refer to a specific form. Instead, it refers to using $\prod_k^{n-1}$, the space of polynomials of degree $\leq k$ in $n-1$ variables, to interpolate any values on dim $\left(\prod_k^{n-1}\right) = \binom{n-1+k}{k}$ nodes. Arbitrary $q \in \prod_k^{n-1}$ has the *standard form* $q(\vec{x}) = \sum_{|\beta|\leq k} c_\beta \vec{x}^\beta$, where $\beta \in \Gamma^{n-1}$. Let $\{\vec{w}_\alpha : \alpha \in \Gamma_k^{n-1}\}$, defined above, be the interpolation nodes and let $v_\alpha \in \mathbb{R}$ denote the corresponding value at each point. The interpolation condition is $q(\vec{w}_\alpha) = \sum_{|\beta|\leq k} c_\beta (\vec{w}_\alpha)^\beta = v_\alpha$ for all $\alpha \in \Gamma_k^{n-1}$ or, in matrix form, $M_k\,(c_\beta) = (v_\alpha)$. Equation (5.1) is a special case of the interpolation condition and our primary task $\left(a_{T_k(\beta)}\right) = M_k^{-1}\left(u_\alpha^{(k)}\right)$ is an application of the general $(c_\beta) = M_k^{-1}\,(v_\alpha)$. For arbitrary values (v_α), interpolating polynomial q exists (and is unique) if and only if M_k is invertible.

It is easiest to show that q exists by switching to another basis for $\prod_k^{n-1}$. For each $\beta \in \Gamma^{n-1}$, define the Newton fundamental polynomial

$$p_\beta\,(x_1, x_2, \ldots, x_{n-1}) = \prod_{m=1}^{n-1} \prod_{j=0}^{\beta_m - 1} (x_m - w_j)\,. \tag{5.2}$$

For example, $p_{(1,0,3)}\,(x,y,z) = (x-w_0)\,(z-w_0)\,(z-w_1)\,(z-w_2)$. The key property of this basis is that

$$\text{if } \alpha_m < \beta_m \text{ for some } m, \text{ then } p_\beta\,(\vec{w}_\alpha) = 0. \tag{5.3}$$

Because (w_i) is a sequence of distinct numbers,

$$p_\beta\,(\vec{w}_\beta) = \prod_{m=1}^{n-1} \prod_{j=0}^{\beta_m - 1} (w_{\beta_m} - w_j) \neq 0;$$

though we do not normalize to make this quantity equal 1 as in [17] and [14].

Arbitrary $q \in \prod_k^{n-1}$ has the *Newton form* $q(\vec{x}) = \sum_{|\beta|\leq k} \hat{c}_\beta\, p_\beta\,(\vec{x})$. If we define matrix $N_k = \left[p_\beta(\vec{w}_\alpha) : \alpha, \beta \in \Gamma_k^{n-1}\right]$, the interpolation condition becomes

$N_k\left(\hat{c}_\beta\right) = (v_\alpha)$. The interpolating polynomial q exists (and is unique), for arbitrary values (v_α), if and only if N_k is invertible. Partition N_k into $\begin{bmatrix} A & B \\ C & D \end{bmatrix}$ by dividing the row index between $|\alpha| < k$ and $|\alpha| = k$ and, likewise, dividing the column index between $|\beta| < k$ and $|\beta| = k$. Then, $A = N_{k-1}$ and, by induction, we may assume N_{k-1} is invertible. The upper-right B contains all zeros by equation (5.3). Also by (5.3), D is a diagonal matrix, since if $|\gamma| = k$ and $|\beta| = k$ but $\gamma \neq \beta$, then $\gamma_m < \beta_m$ for some m. Entries along the diagonal, $D_{\gamma\gamma} = p_\gamma\left(\vec{w}_\gamma\right)$, are nonzero. Thus, N_k is invertible and, in fact, lower triangular.

The matrix partition leads to an algorithm for N_k^{-1}. Since

$$\begin{aligned} N_k &= \begin{bmatrix} N_{k-1} & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} I & 0 \\ C & I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & D \end{bmatrix}, \\ N_k^{-1} &= \begin{bmatrix} I & 0 \\ 0 & D^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -C & I \end{bmatrix} \begin{bmatrix} N_{k-1}^{-1} & 0 \\ 0 & I \end{bmatrix}. \end{aligned}$$

For iteration, assume we have computed $\left(\hat{c}_\alpha\right) = N_{k-1}^{-1}\left(v_\alpha\right)$ for $|\alpha| < k$. For $|\gamma| = k$, compute $\left(\hat{c}_\gamma\right) = D^{-1}\left(\left(v_\gamma\right) - C\left(\hat{c}_\alpha\right)\right)$, which for one γ is the formula

$$\hat{c}_\gamma = \frac{1}{p_\gamma\left(\vec{w}_\gamma\right)} \left(v_\gamma - \sum_{|\beta| \leq k-1} \hat{c}_\beta\, p_\beta(\vec{w}_\gamma) \right). \tag{5.4}$$

A simple Newton coefficient algorithm iterates this through each k and γ. This algorithm is computationally equivalent to Algorithm 4.4 in [14], called "Finite differences". The summation in equation (5.4) is displayed over all β such that $|\beta| \leq |\gamma| - 1$ but only those $\beta \leq \gamma$ coordinatewise are needed, since otherwise equation (5.3) would make $p_\beta(\vec{w}_\gamma) = 0$. Since the values $p_\beta(\vec{w}_\gamma)$ depend only on the nodes, they could be pre-computed and stored, though this would take much more space than the pyramid of values being computed. We will develop a more efficient method to compute N_k^{-1} that is truly a divided difference algorithm as in one variable.

Returning to our goal, we need $\left(c_\beta\right) = M_k^{-1}\left(v_\alpha\right)$, the coefficients of the interpolating polynomial in standard form $q(\vec{x}) = \sum_{|\beta| \leq k} c_\beta \vec{x}^\beta$. We have $\left(\hat{c}_\beta\right) = N_k^{-1}\left(v_\alpha\right)$, the coefficients of the same polynomial in Newton form $q(\vec{x}) = \sum_{|\beta| \leq k} \hat{c}_\beta\, p_\beta\left(\vec{x}\right)$. Clearly, the Newton form can be expanded and collected to get each c_α as a linear combination of $\hat{c}_\beta$'s, defining a matrix R_k such that $\left(c_\alpha\right) = R_k\left(\hat{c}_\beta\right)$. Since $\vec{x}^\alpha$ is a term in the expansion of p_β only if $\beta \geq \alpha$ coordinatewise, matrix R_k is upper-right triangular. Matrix N_k^{-1} is lower-left triangular, as seen inductively from the above factorization. Our algorithm will have these two distinct steps represented by the matrix factorization $M_k^{-1} = R_k N_k^{-1}$. We will design algorithms DivDiff and CollectTerms such that $\text{DivDiff}(\left(v_\alpha : \alpha \in \Gamma_k^{n-1}\right)) = N_k^{-1}\left(v_\alpha\right)$ and $\text{CollectTerms}(\left(\hat{c}_\beta : \beta \in \Gamma_k^{n-1}\right)) = R_k\left(\hat{c}_\beta\right)$. Thus,

$$\text{CollectTerms}(\ \text{DivDiff}(\left(u_\alpha^{(k)}\right))) = M_k^{-1}\left(u_\alpha^{(k)}\right).$$

6. Choice of nodes

The nodes for interpolation in $\mathbb{R}^{n-1}$ may be chosen more generally, with some care. The Lagrange polynomial interpolation problem is not poised if the points lie

on some algebraic hypersurface. For example, in $\mathbb{R}^2$, the $\dim\left(\prod_2^2\right) = \binom{2+2}{2} = 6$ but if the 6 nodes lie on a conic section, then either the polynomial does not exist or it is not unique, depending on the values at the nodes. Given the appropriate number of points, the algorithm in [17] (summarized in [14]) will either construct a set of Newton fundamental polynomials for the nodes or determine that the problem is not poised. The construction will arrange the nodes by the labels $\{\vec{x}_\alpha \in \mathbb{R}^{n-1} : \alpha \in \Gamma_d^{n-1}\}$ and construct $p_\beta \in \prod_{|\beta|}^{n-1}$ such that $p_\beta(\vec{x}_\alpha) = \delta_{\alpha\beta}$ for $|\alpha| \le |\beta|$. Of course, these polynomials will be more complicated than the specific p_β defined above. Still, in theory, the steps outlined above could be carried out. The N_k^{-1} step is accomplished by equation (5.4) using the more complicated p_β's. The R_k step would correspond to a block-upper-right triangular matrix with entries determined by the algorithm that constructed the p_β's. This would succeed (if poised) in inverting the generalization of M_k, specifically $\left[(\vec{x}_\alpha)^\beta : \alpha, \beta \in \Gamma_k^{n-1}\right]$. Such matrices are always nested, so they are invertible if and only if the Schur complement is invertible. Our node structure is precisely the one that will make the Schur complement into a diagonal matrix. In the sections that follow, algorithms DivDiff and CollectTerms will exploit our $\vec{w}_\alpha$'s and the corresponding Newton fundamental polynomials defined in equation (5.2).

Our node structure is a "wedge" of lattice points on an arbitrary irregular grid in $\mathbb{R}^{n-1}$. Specifically, for each axis $m \in \{1, 2, \ldots, n-1\}$, pick any sequence of distinct reals $w_0^{[m]}, w_1^{[m]}, w_2^{[m]}, \ldots$, interpreted as arbitrarily spaced grid marks labelled in arbitrary (not necessarily monotone) order. We refer to the set

$$\left\{\vec{w}_\alpha = (w_{\alpha_1}^{[1]}, w_{\alpha_2}^{[2]}, \ldots, w_{\alpha_{n-1}}^{[n-1]}) : \alpha_1 + \alpha_2 + \cdots + \alpha_{n-1} \le d\right\}$$

as a *wedge* of nodes. Both (a) and (b) in Figure 3 are example wedges for $n-1 = 2$, $d = 5$. In (a), the multi-indices form a wedge using $w_i = i$ on both axes. In (b), the w_i sequence is 0, 1, $-1, 1/2, -1/2, 1/4$ on both axes. Lacking a computational purpose for different grid marks on different axes, we drop the axis superscript and use the same sequence. Nevertheless, all our algorithms are easily adjusted to keep track of different sequences if so desired. By translation, we may assume $w_0 = 0$,

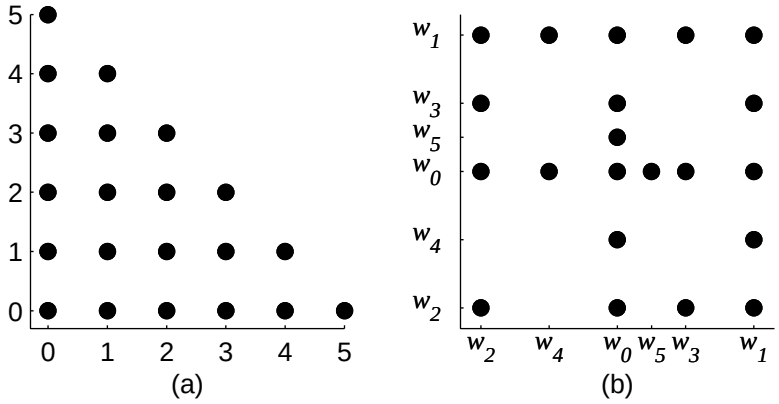

FIGURE 3. $n-1 = 2$, $d = 5$; (a) multi-indices α corresponding to (b) interpolation nodes for alternating dyadic rational w_i.

although this assumption is used only in Algorithm 5. There is a computational purpose in allowing irregular grid divisions. Since $M_k = \left[(\vec{w}_\alpha)^\beta : \alpha, \beta \in \Gamma_k^{n-1}\right]$ is a type of Vandermonde matrix, it may have a large condition number. The most regular grid uses sequence $w_i = i$, so that the wedge is the multi-index set Γ_d^{n-1}. For this regular corner of nodes, with $n-1=5$, the condition number of M_5 is 3×10^5 and the important values $p_\beta(\vec{w}_\alpha)$ are products of factorials. If we use an alternating dyadic rational sequence $0, 1, -1, 1/2, -1/2, 1/4, -1/4, 3/4, -3/4, 1/8, \ldots$ for the grid marks, with $n-1=5$, the condition number of M_5 is 3×10^3 and the important values $p_\beta(\vec{w}_\alpha)$ are products of dyadic rationals between -2 and 2. Since the condition number is a serious concern for numerical accuracy, we choose the irregular dyadic rational wedge for numerical experiments.

$$\begin{aligned} w_0 &= 0 \quad \text{and} \quad w_1 = 1. \\ \text{If } i \text{ is odd, } w_i &= \frac{i \bmod 2^p}{2^p} \text{ where } 2^p < i < 2^{p+1}. \\ \text{If } i \text{ is even, } w_i &= -w_{i-1}. \end{aligned}$$

In our algorithms, we use an arbitrary sequence (w_i) for clarity, leaving the door open for further work to lower the condition number.

The specific structure of a wedge of nodes is mentioned in a few old texts ([16], [8], and [7]) but is hard to find otherwise. The well-known geometric condition of nodes on hyperplanes (theorem of Chung and Yao in [9, section 6.10]) fails to include a wedge on an irregular grid. A regular grid (monotone, equally spaced marks) does create a triangular case that is handled in several papers (see survey [3]). Still, only the old texts use the simple Newton polynomials as in equation (5.2). These texts present the bivariate extension of the univariate divided-difference algorithm and observe that "extension to more dimensions offers no difficulty" [7]. While a rectangular grid is considered first, they do cover the case that we call a wedge.

7. A divided difference algorithm

Let $\{v_\alpha \in \mathbb{R} : \alpha \in \Gamma_d^{n-1}\}$ be the values at a corresponding wedge of nodes $\{\vec{w}_\alpha = (w_{\alpha_1}, w_{\alpha_2}, \ldots, w_{\alpha_{n-1}}) \in \mathbb{R}^{n-1} : \alpha \in \Gamma_d^{n-1}\}$. For $\alpha \le \beta$ coordinatewise in Γ_d^{n-1}, define $v[\alpha, \alpha] = v_\alpha$ and, if $\alpha \ne \beta$, the *divided difference*

$$v[\alpha, \beta] = \frac{v[\alpha + e_m, \beta] - v[\alpha, \beta - e_m]}{w_{\beta_m} - w_{\alpha_m}} \tag{7.1}$$

where m is the largest integer such that $\alpha_m < \beta_m$ and $e_m = (0, \ldots, 0, 1, 0, \ldots, 0)$ with a 1 in the mth position. The bivariate case $v[(\alpha_1, \alpha_2), (\beta_1, \beta_2)]$ is written as $v[x_{\alpha_1}, \ldots, x_{\beta_1}; y_{\alpha_2}, \ldots, y_{\beta_2}]$ in the more typical divided difference notation, found in the old texts such as [7], though we assume for simplicity that each $x_i = y_i = w_i$. For example, the definition of $v[(0,3),(8,6)]$ would use $e_2 = (0,1)$ to perform what would be $\left(v[x_0, \ldots, x_8; y_4, \ldots, y_6] - v[x_0, \ldots, x_8; y_3, \ldots, y_5]\right)/(y_6 - y_3)$ in the old notation. By using the largest integer m such that $\alpha_m < \beta_m$, we follow these old texts which assume only one y node whenever using a divided difference in x. Actually, the same value results for any integer m such that $\alpha_m < \beta_m$ but we do not need this proposition.

By nested application of the univariate principle, as in [7],

$$q(\vec{x}) = \sum_{|\beta| \le k} v[\vec{0}, \beta]\, p_\beta(\vec{x})$$

is the unique polynomial in $\prod_k^{n-1}$ that interpolates v_α at each $\vec{w}_\alpha$ for $\alpha \in \Gamma_k^{n-1}$. In our previous notation, $\left(v[\vec{0},\beta]\right) = (\hat{c}_\beta) = N_k^{-1}(v_\alpha)$, so a divided difference algorithm will accomplish the N_k^{-1} step.

We need to organize a tableau of values but, just as in the univariate case [9], this does not actually require any additional memory. For $n-1=2$ and $k=5$, the tableau could be envisioned as values corresponding to the multi-indices β as shown in Figure 3(a). After initializing this example tableau of values, the jth pass would proceed from multi-indices of order 5 down to order j, replacing the value at β by its divided difference with a value one step to the left or down. Which way to difference depends on β and j, as one "uses up" the components of β: for example at $(2,1)$, the first pass $j=1$ will difference in coordinate $m=1$ and leaves $\alpha_m = 1$ steps to be done in that coordinate; pass $j=2$ will difference in coordinate $m=1$ and leaves $\alpha_m = 0$; pass $j=3$ will difference in coordinate $m=2$ and leaves $\alpha_m = 0$. We now define this process in general and show how it is related to the $v[\vec{0},\beta]$ notation.

Define $v_0[\beta] = v[\beta,\beta] = v_\beta$. For each $\beta \neq \vec{0}$ and integer j with $1 \leq j \leq |\beta|$, define

$$v_j[\beta] = v[(0,\ldots,0,\alpha_m,\beta_{m+1},\ldots,\beta_{n-1}),\,\beta\,] \quad \text{where } \alpha_m = \left(\sum_{i=1}^{m}\beta_i\right) - j \geq 0 \text{ and } \sum_{i=1}^{m-1}\beta_i < j \tag{7.2}$$

and m is defined by the inequalities. Then, $v_{|\beta|}[\beta] = v[\vec{0},\beta]$. We now show that

$$v_j[\beta] = \frac{v_{j-1}[\beta] - v_{j-1}[\beta - e_m]}{w_{\beta_m} - w_{\alpha_m}} \tag{7.3}$$

using m for β and j. To prove this, define α so that $v_j[\beta] = v[\alpha,\beta]$ as in equation (7.2). Consider the case $\sum_{i=1}^{m-1}\beta_i < j-1$, so that the same coordinate m will be used in $v_j[\beta]$, $v_{j-1}[\beta]$, and $v_{j-1}[\beta - e_m]$. Then, $v_{j-1}[\beta]$ differs from equation (7.2) simply by replacing j with $j-1$. This adds one to the mth coordinate of α so that $v_{j-1}[\beta] = v[\alpha + e_m, \beta]$. If, in addition, we use $\beta - e_m$ in equation (7.2), both mth coordinates are lowered by 1, resulting in $v_{j-1}[\beta - e_m] = v[\alpha, \beta - e_m]$. Thus, equation (7.3) is a special case of equation (7.1). In the other case $\sum_{i=1}^{m-1}\beta_i = j-1$, so $v_{j-1}[\beta] = v[(0,\ldots,0,\beta_m,\beta_{m+1},\ldots,\beta_{n-1}),\,\beta]$ since $j-1$ is subtracted from a partial sum in an earlier coordinate. Likewise, $v_{j-1}[\beta - e_m] = v[(0,\ldots,0,\beta_m - 1,\beta_{m+1},\ldots,\beta_{n-1}),\,\beta - e_m]$. Since $\beta_m = (\sum_{i=1}^{m}\beta_i) - (\sum_{i=1}^{m-1}\beta_i) = (\sum_{i=1}^{m}\beta_i) - j + 1 = \alpha_m + 1$, we have $v_{j-1}[\beta] = v[\alpha + e_m, \beta]$ and $v_{j-1}[\beta - e_m] = v[\alpha, \beta - e_m]$ as before.

We can now summarize an algorithm to compute $\hat{c}_\beta = v[\vec{0},\beta] = v_{|\beta|}[\beta]$. This algorithm overwrites $v_{j-1}[\beta]$ by $v_j[\beta]$ so that the j subscript is not needed. By processing β in decreasing order, $v_{\beta - e_m}$ still contains the value at the $j-1$ level. To end with $v_{|\beta|}[\beta]$, v_β is updated at level j if and only if $|\beta| \geq j$. The reciprocals of the divided difference denominators may be precomputed and stored in a small reference array, $dw_{i,j} = 1/(w_i - w_j)$ for all $0 \leq j < i \leq d$ (not the above j). For clarity, we show a subtraction and division in the algorithms but implementation may replace these by one multiplication by $dw_{i,j}$. Since this is the only use of division in our algorithms, we count divisions and multiplications as the same in our floating-point operation counts in the conclusion.

Algorithm 3. DivDiff

INPUT: integer k and array $\left(v_\beta : \beta \in \Gamma_k^{n-1}\right)$ of real values.
GLOBAL: sequence (w_i) for node coordinates.
OUTPUT: array $(\hat{c}_\beta)$ of Newton coefficients (to interpolate each v_α at $\vec{w}_\alpha$).
For $j = 1$ to k do
  For *order* $= k$ down to j do
    For each $\beta \in \Gamma^{n-1}$ with $|\beta| =$ *order* do
      $m =$ first m such that $\alpha_m = (\sum_{i=1}^{m} \beta_i) - j \geq 0$;
      $v_\beta = \dfrac{v_\beta - v_{\beta - e_m}}{w_{\beta_m} - w_{\alpha_m}}$;
    End For β;
  End For *order*;
End For j;
Return $\left(v_\beta : \beta \in \Gamma_k^{n-1}\right)$.

There are several alternative ways to implement such an algorithm. Let us consider how the array access in $v_{\beta - e_m}$ might be performed. In practice, $\left(v_\beta : \beta \in \Gamma_k^{n-1}\right)$ is stored as $\left(v[i] : i = 0, \ldots, \binom{n-1+k}{k} - 1\right)$, a linear array with nonnegative integer index. As the index i progresses, multi-index β can be incremented by Algorithm 2 (or found in a global reference array multi$[i] = \beta$). The inverse relation is given by a sum of binomial coefficients [15]. In particular, we need the numbers Pascal$[c, k] = \binom{c+k}{c}$, the cardinality of Γ_k^c, for $k = -1, 0, 1, \ldots, d$ and $c = 1, \ldots, n-1$. Let Pascal$[c, -1] = 0$. Using Pascal$[c, 0] = 1$ and Pascal$[0, k] = 1$, the rest of the values can be stored in a small, relative to v, reference array generated exactly as in Pascal's triangle. The multi-index $\beta = (order, 0, \ldots, 0) \in \Gamma^{n-1}$ has integer index $i =$ Pascal$[n-1, order-1]$; and $\beta - e_1$ has integer index $ie1 =$ Pascal$[n-1, order-2]$. In general, the nonnegative integer index is given by the function

$$\text{index}(\beta) \quad = \quad \sum_{c=1}^{n-1} \text{Pascal}[n - c,\, s_c] \qquad \text{where } s_c = \beta_c + \beta_{c+1} + \cdots + \beta_{n-1} - 1. \tag{7.4}$$

If $p_c = \beta_1 + \beta_2 + \cdots + \beta_c$, the usual partial sum, then $s_c = |\beta| - 1 - p_{c-1}$. Using these partial sums based on β, we seek the integer index *idown* of $\beta - e_m$. Compared to β, the order of $\beta - e_m$ is one lower but the partial sum corresponding to p_{c-1} is one lower only when $c > m$. Thus, *idown*$= \sum_{c=1}^{m} \text{Pascal}[n - c, |\beta| - 2 - p_{c-1}] + \sum_{c=m+1}^{n-1} \text{Pascal}[n - c, |\beta| - 1 - p_{c-1}]$.

Algorithm 4. β loop in DivDiff Algorithm 3.

FIXED: level j and *order*.
$\beta = (order, 0, \ldots, 0)$;
$ie1 =$ Pascal$[n - 1, order - 2]$;
For $i =$ Pascal$[n - 1, order - 1]$ to Pascal$[n - 1, order] - 1$ do
  $m = 1$;
  $psum = \beta_1$;
  $idown = ie1$;
  While $psum < j$ do
    $m = m + 1$;
    $idown = idown +$ Pascal$[n - m, order - 2 - psum]$;
    $psum = psum + \beta_m$;

End While;
$b = \beta_m$;
$a = psum - j$;
For $c = m + 1$ to $n - 1$ do
    $idown = idown + \text{Pascal}[n - c, order - 1 - psum]$;
    $psum = psum + \beta_c$;
End For c;
$v[i] = (v[i] - v[idown])/(w[b] - w[a])$;
$\beta = \text{Increment}(\beta)$;
End For i.

Algorithm 4 avoids using any reference arrays that are as large as v, but a different, more time efficient, algorithm can avoid this computation of *idown*, a, and b. Three reference arrays $a[i], b[i], \text{idown}[i]$ can be used to directly find these values from the earlier values $a[\hat{\imath}], b[\hat{\imath}], \text{idown}[\hat{\imath}]$, without even knowing the multi-index β or the m explicitly. This avoids using Algorithm 2 with a space cost equivalent to only three columns in a listing of multi-indices. The idea is contained in the recursive relationship in Γ_{order}^{n-1}:

$$\{\beta : |\beta| = order\} = \bigcup_{c=1}^{n-1} \{\alpha + e_c : |\alpha| = order - 1 \text{ and } \alpha_1 = \cdots = \alpha_{c-1} = 0\}.$$

For each c, an integer index $\hat{\imath}$ can loop through values representing α, while the integer index i represents $\beta = \alpha + e_c$. To define idown$[i]$, we seek $\beta - e_m$ with m chosen for level j. Since higher orders are set before lower orders, idown$[\hat{\imath}]$ contains the index of $\alpha - e_{m'}$ with m' chosen for level $j - 1$. Because $\beta = \alpha + e_c$, we see that $m' = m$, so $\beta - e_m$ and $\alpha - e_{m'}$ also differ by e_c. Thus, idown$[i]$ and idown$[\hat{\imath}]$ differ by a fixed constant Δ_c. For the first $\hat{\imath}$ for a given c, the first $\alpha = (j-1) * e_c$, so $\beta - e_m = \alpha$ and we set idown$[i] = \hat{\imath}$ and compute $\Delta_c = \hat{\imath} - \text{idown}[\hat{\imath}]$. For all subsequent $\hat{\imath}$, set idown$[i]$ = idown$[\hat{\imath}] + \Delta_c$. Other details of implementing this algorithm will be left for a more technical discussion.

8. Collecting terms from the Newton form

The divided difference algorithm returns coefficients $(\hat{c}_\beta)$ of Newton fundamental polynomials (p_β), but the desired multivariate Taylor coefficients of some order k are given by standard coefficients (c_α) such that $\sum_{|\alpha| \le k} c_\alpha \, \vec{x}^{\,\alpha} = \sum_{|\beta| \le k} \hat{c}_\beta \, p_\beta(\vec{x})$. Define $r_{\alpha\beta}$ to be the coefficient of $\vec{x}^{\,\alpha}$ in p_β, so that $p_\beta(\vec{x}) = \sum_{|\alpha| \le k} r_{\alpha\beta} \, \vec{x}^{\,\alpha}$. Thus, $c_\alpha = \sum_{|\beta| \le k} r_{\alpha\beta} \, \hat{c}_\beta$, or in matrix form $(c_\alpha) = R_k \, (\hat{c}_\beta)$. By definition, $p_\beta \, (x_1, x_2, \ldots, x_{n-1}) = \prod_{m=1}^{n-1} \prod_{j=0}^{\beta_m - 1} (x_m - w_j)$. The leading term is $\vec{x}^{\,\beta}$, so $r_{\beta\beta} = 1$. Other nonzero terms $\vec{x}^{\,\alpha}$ of p_β can occur only if $\alpha \le \beta$ coordinatewise. (In particular, $r_{\alpha\beta} = 0$ if $|\alpha| \ge |\beta|$ and $\alpha \ne \beta$, so R_k is upper-right triangular as mentioned before.) Thus,

$$c_\alpha = \hat{c}_\alpha + \sum_{\beta \ge \alpha, \beta \ne \alpha}^{|\beta| \le k} r_{\alpha\beta} \, \hat{c}_\beta. \tag{8.1}$$

The $r_{\alpha\beta}$ values could be "hard-wired" in these algorithms since they depend only on the sequence (w_i) for node coordinates, not on any function. However, the set of possible nonzero values, $\left\{r_{\alpha\beta} : \alpha, \beta \in \Gamma_d^{n-1}, \beta \ge \alpha \text{ coordinatewise}\right\}$, has cardinality

TABLE 2. Expanded coefficients of $(x-w_0)(x-w_1)\cdots(x-w_{j-1})$ in column j, for alternating dyadic rational w_i.

$b_{i,j}$	0	1	2	3	4	5	6	7	8
0	1	0	0	0	0	0	0	0	0
1		1	-1	-1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{-1}{16}$	$\frac{-1}{64}$	$\frac{3}{256}$
2			1	0	-1	0	$\frac{1}{4}$	0	$\frac{-1}{64}$
3				1	$\frac{-1}{2}$	$\frac{-5}{4}$	$\frac{5}{16}$	$\frac{21}{64}$	$\frac{-63}{256}$
4					1	0	$\frac{-5}{4}$	0	$\frac{21}{64}$
5						1	$\frac{-1}{4}$	$\frac{-21}{16}$	$\frac{63}{64}$
6							1	0	$\frac{-21}{16}$
7								1	$\frac{-3}{4}$
8									1

$\binom{2(n-1)+d}{d}$. This is prohibitively large. In the $n, d = 10$ case, there are 13,123,110 such values while the entire pyramid of Taylor coefficients contains 184,756 values, and the size of the largest input vector to DivDiff or CollectTerms is half that, at 92,378. The set of distinct nonzero values is smaller and may be tractable. We choose to compute the values from the univariate values for each coordinate.

Define $q_j(x) = (x-w_0)(x-w_1)\cdots(x-w_{j-1})$ and define $b_{i,j}$ to be the coefficient of x^i in $q_j(x)$. Since

$$p_\beta(\vec{x}) = \prod_{m=1}^{n-1} q_{\beta_m}(x_m),$$

we see that

$$r_{\alpha\beta} = \prod_{m=1}^{n-1} b_{\alpha_m,\beta_m}.$$

The array $B = [b_{i,j} : i, j = 0, 1, \ldots, d]$ has comparatively trivial size. The entries follow from a recurrence relation $b_{i,j} = b_{i-1,j-1} - w_{j-1}\, b_{i,j-1}$, found by distributing $(x - w_{j-1})$, to get $q_j(x) = x\, q_{j-1}(x) - w_{j-1}\, q_{j-1}(x)$. Table 2 shows $b_{i,j}$ for alternating dyadic rational (w_i). This small table suffices for any number of variables n, up to order $d = 8$. For any (w_i), $b_{i,j} = 0$ for $i > j$ and these values are never called. (This unused portion of the array may be used to hold the $dw_{i,j} = 1/(w_i - w_j)$ values as mentioned just before Algorithm 3.) All $b_{j,j} = 1$, including $b_{0,0} = 1$. For $j > 0$, the constant term of q_j is $b_{0,j} = \prod_{i=0}^{j-1}(-w_i) = 0$, under our assumption that $w_0 = 0$, producing zeros in the first row of Table 2. Our algorithms will not assume the other zeros and patterns in Table 2, since they are more specific to the alternating dyadic rationals and we want to leave flexibility in the choice of nodes.

Because of the $w_0 = 0$ assumption, if $\alpha_m = 0$ and $\beta_m \neq 0$ for some m, then $r_{\alpha\beta} = 0$. Thus, the summation in formula (8.1) may be restricted to the index set

$$\{\beta \in \Gamma_k^{n-1} : \beta \geq \alpha \text{ coordinatewise}, \beta \neq \alpha, \text{ and } \alpha_m = 0 \implies \beta_m = 0\}.$$

In other words, β can be made by adding to the nonzero components of α. Specifically, suppose α has m nonzero components and $NZ = \{i : \alpha_i \neq 0\}$. Define $E_\alpha : \Gamma_k^m \to \Gamma_k^{n-1}$ to embed any m components into coordinates where α is nonzero.

For example, if $\alpha = (0,1,0,0,3)$ and $\mu = (6,0)$, then $NZ = \{2,5\}$, $E_\alpha(\mu) = (0,6,0,0,0)$ and $\alpha + E_\alpha(\mu) = (0,7,0,0,3)$.

Algorithm 5. CollectTerms

INPUT: integer k and array $\left(\hat{c}_\beta : \beta \in \Gamma_k^{n-1}\right)$ (of Newton coefficients).
GLOBAL: reference array $[b_{i,j}]$ (combinations of (w_i) node coordinates).
OUTPUT: array (c_α) of standard coefficients in expansion of $\sum \hat{c}_\beta\, p_\beta$.
For *order* = 1 to $k-1$ do
  For each $\alpha \in \Gamma^{n-1}$ with $|\alpha| =$ *order* do
    $NZ = \{i \in \{1,\ldots,n-1\} : \alpha_i \neq 0\}$;
    $m =$ cardinality(NZ);
    For each $\mu \in \Gamma_{k-order}^m$, $\mu \neq 0$ do
      $\beta = \alpha + E_\alpha(\mu)$;
      $\hat{c}_\alpha = \hat{c}_\alpha + \hat{c}_\beta \prod_{c \in NZ} b_{\alpha_c,\beta_c}$;
    End For μ;
  End For α;
End For *order*;
Return $\left(\hat{c}_\alpha : \alpha \in \Gamma_k^{n-1}\right)$.

Since every $|\beta| > |\alpha|$, $\hat{c}_\beta$ values remain unchanged while accumulating the sum for c_α in the $\hat{c}_\alpha$ location. For $\alpha = \vec{0}$ or $|\alpha| = k$, $\hat{c}_\alpha$ is never changed. In the μ loop, $\beta = \alpha + E_\alpha(\mu)$ (actually, its integer index) and the subsequent product can be computed in one loop indexed by $c \in NZ$. The number of multiplications could be reduced, since the embedding and the product do not need to include any c where $\mu_c = 0$, since $b_{\alpha_c,\alpha_c} = 1$. To be efficient for alternating dyadic rationals (w_i), all μ's could be skipped that lead to a zero in Table 2. A different looping order might make it possible to re-use products, since the same product is computed many times over, as different α's can have the same nonzero components in different coordinates. We will not assume any of these efficiencies in the concluding analysis.

Computing the integer index of β is again the most significant task in implementing this conceptual algorithm. One may accomplish this by the index function in formula (7.4). Alternatively, apply the indexing scheme of Berz [1]. By his method, if integer indices i of α and j of $\gamma = E_\alpha(\mu)$ are known, then one addition and indirect references return the integer index of $\beta = \alpha + \gamma$. The index i is trivially incremented along with α. Incrementing j is easy if there is no restriction to nonzero coordinates, so $\gamma = \mu \in \Gamma_{k-order}^{n-1}$. Otherwise, some computation, such as (7.4), appears necessary to convert the integer index of $\mu \in \Gamma^m$ to the index of $\gamma = E_\alpha(\mu) \in \Gamma^{n-1}$. One could always go through all $\gamma \in \Gamma_{k-order}^{n-1}$, instead of μ, although many would correspond to some $\alpha_c = 0$ and $\gamma_c \neq 0$ so that $b_{\alpha_c,\gamma_c} = 0$, meaning unnecessary loops and multiplications.

A different approach to the CollectTerms algorithm is to proceed by columns rather than rows of the R_k matrix. For each $\beta \in \Gamma_k^{n-1}$ in increasing order, the value $\hat{c}_\beta$ is used to contribute to the accumulations in each c_α with α in $\{\alpha \in \Gamma^{n-1} : \alpha \leq \beta$ coordinatewise, $\alpha \neq \beta$, and $\beta_m \neq 0 \implies \alpha_m \neq 0\}$. This set can be described as an $(n-1)$-dimensional rectangular grid with lower corner $\chi(\beta) \in \Gamma^{n-1}$ defined by $\chi(\beta)_c = \min\{1, \beta_c\}$. For each β, increment α coordinatewise from $\chi(\beta)$ up to but not including β. (Sometimes $\beta = \chi(\beta)$, and the α loop is empty.) In this way, the integer index of β is free but the α loop must also compute a corresponding integer index. In the index formula (7.4), partial sums may be retained from the previous α to be used for the next α, since many coordinates remain unchanged.

9. Comparisons and conclusions

Theoretical measures of time and space efficiency will show that our interpolation method has advantages in both measures, when compared to current methods that tackle the massive task of computing multivariable derivatives or Taylor coefficients to high order. Prior to interpolation, automatic differentiation techniques were applied directly to data structures for multi-dimensional Taylor coefficients or derivatives, a method we abbreviate as *multi-AD*. The *GUW-method* of interpolation in [5] showed that directional univariate AD accomplishes time savings over multi-AD with a cost of more space and significant time cost in reconstructing multivariate from univariate. We now show that these hurdles have been overcome in our method which consists of univariate AD computation in Algorithm 1, the DivDiff Algorithm 3, and the CollectTerms Algorithm 5.

By our choice of directions, we do not need more space and, in fact, use less memory than multi-AD. The pyramid of multivariable Taylor coefficients has $\binom{n+d}{d}$ entries and static allocation of this much memory is enough to accomplish everything in our method. (Other temporary or reference arrays are of negligible size, such as $(d+1)^2$.) Actually, the output coefficients can be written to an external device as they are created by order. The largest single array ever actually used by our algorithms has $\binom{n+d-1}{d}$ entries, corresponding to multivariable coefficients of order d in $\mathbb{R}^n$ and to polynomial coefficients of order $\le d$ in $\mathbb{R}^{n-1}$. Algorithm 1 maintains $(d-k+1)$ arrays of length $\binom{n+k-1}{k}$, and $(d-k+1)\binom{n+k-1}{k} < \binom{n+d}{d}$ for every k. In contrast, multi-AD would typically require $3\binom{n+d}{d}$ entries; for example, a sin operation requires three pyramids, one each for sin, cos, and the argument. The GUW-method was based on storing $1 + \frac{d\,n}{d+n}\binom{n+d}{d}$ univariate AD values.

For reconstructing multivariate from univariate values, we must admit that a large part of the time cost is in handling integer indices corresponding to different multi-indices. However, this is a significant advantage of interpolation over multi-AD which must deal with this difficulty throughout calculation using the function expression. Implementation of index manipulation should strive to reduce additions, (in)equality tests, indirect array references, and widely scattered memory access. The actual time cost is greatly affected by implementation details that are not settled by our mathematical algorithms. It is more appropriate here to compare algorithms in terms of floating-point multiplications and divisions, assuming that index costs are cheaper done once (in DivDiff and CollectTerms or in GUW-method interpolation) than repeatedly in multi-AD.

There are five relevant operation counts that are actually functions of d and n. For each operation in the function expression f being differentiated, $\text{pyr}\,\# = \binom{2n+d}{d}$ is the number of floating-point multiplications required to operate on a pyramid of coefficients in multi-AD. For each f operation, the univariate automatic differentiation in Algorithm 1 will total $\text{uni}\,\# = \binom{2+d}{d}\binom{n+d-1}{d}$ floating-point multiplications for all the loops. (We use multiplication to represent all f operations, since they are computationally similar in automatic differentiation.) Let dd# denote the number of divisions (implemented as multiplications) in calls to DivDiff, Algorithm 3, for $k = 1$ to d. Then

$$\text{dd}\,\#(d,n) = \sum_{k=1}^{d}\sum_{j=1}^{k}\sum_{order=j}^{k}\binom{n-1+order-1}{order} = (n-1)\binom{n+d}{n+1}.$$

Let ct # be the number of multiplications in calls to CollectTerms, Algorithm 5, for $k = 1$ to d. Then

$$\text{ct}\,\#(d,n) = \sum_{k=1}^{d} \sum_{order=1}^{k-1} \sum_{m=1}^{n-1} \binom{n-1}{m} \binom{order-1}{order-m} \left[\binom{m+k-order}{k-order} - 1 \right] m.$$

This number could be lowered as discussed immediately after Algorithm 5, but this will suffice to show a reasonable expense. Finally, [5] bounded (and conjectured equality for) the number of necessary (nonvanishing coefficient) multiplications in the GUW-method interpolation formula by a quantity $p(d,n)$ which we rename

$$\text{guw}\,\#(d,n) = \sum_{m=1}^{d} \binom{n}{m} \binom{d}{m} \binom{m+d-1}{d}.$$

This is the number of nonvanishing $c_{\psi,\varphi}$ described in Section 4 and does not include significant work in creating the coefficients. For the $n, d = 10$ case, $\text{pyr}\,\# = 30,045,015$, $\text{uni}\,\# = 6,096,948$, $\text{dd}\,\# = 1,511,640$, $\text{ct}\,\# = 6,820,110$, and $\text{guw}\,\# = 368,750,757$.

The first conclusion is that interpolation is much cheaper in our method than in the GUW-method, since $\text{dd}\,\# + \text{ct}\,\# < \text{guw}\,\#$ for all d and n. In fact, calculations show the bound $\text{dd}\,\# + \text{ct}\,\# < 0.56 \cdot \text{pyr}\,\#$ for all d and n. In contrast, [5] shows that $\text{guw}\,\# \approx \left(2^{d-1}/\sqrt{\pi d}\right) \text{pyr}\,\#$ for large d. The smaller bound insures that the one-time interpolation cost will not overshadow the savings from univariate AD versus multivariate AD.

We now compare the total cost of our method versus multi-AD. Let ops_f be the number of operations (binary or unary, except addition and scalar multiplication) in the function expression f being differentiated. Actually, operations that must be done in pairs, such as sin and those in Table 1, should be counted twice. The total cost of our method is $\text{ops}_f \cdot \text{uni}\,\# + \text{dd}\,\# + \text{ct}\,\#$, while the cost of multi-AD is $\text{ops}_f \cdot \text{pyr}\,\#$. Using the bound from the preceding paragraph, the cost ratio

$$r(d,n) = \frac{\text{ops}_f \cdot \text{uni}\,\# + \text{dd}\,\# + \text{ct}\,\#}{\text{ops}_f \cdot \text{pyr}\,\#} < \frac{\text{uni}\,\#}{\text{pyr}\,\#} + \frac{0.56}{\text{ops}_f}.$$

The ratio $q\,(d,n) = \frac{\text{uni}\,\#}{\text{pyr}\,\#}$ is studied in [5]. As long as there are at least a few operations in f, conclusions based on $q(d,n)$ are valid. In particular, $q\,(d,n) < 1$ for $d \geq 5$ and q goes exponentially to zero as d increases. We conclude that, for $d \geq 5$, our method will be faster than multi-AD and much faster for higher d. For $d = 4$, $q(d,n)$ hovers around 1 and so the methods would have a similar number of multiplications; still our method should be preferable due to the savings in space cost and index costs as discussed in the second and third paragraphs of this section. For $d = 2$ and 3, there will be more multiplications in our method than in multi-AD but we can appeal to the observed global bounds of $q(d,n) < 1.5$ and $r(d,n) < 2$. Such a modest increase in multiplications may be worth the saving in space cost and index costs. The dependence on n is not as significant, though our method with $n = 2$ reduces to pure one-variable interpolation so the indexing could be trivialized in this case.

The numerical stability of our method is a serious concern that we leave for further study. The entire method is theoretically exact, since Taylor and interpolating polynomials are never used for approximation but only to describe exact values.

Still, there is a massive number of floating-point computations that will introduce roundoff error and the interpolation steps are mathematically equivalent to multiplying by a large, possibly ill-conditioned, inverse matrix. The choice of values (w_i) for direction nodes is unrestricted for dealing with this. We found alternating dyadic rationals to yield the best result among a few ad hoc alternatives. Our algorithms were tested, using double-precision floating-point arithmetic in Mathematica on a PC, on multivariate exponential and random polynomial functions where exact results were known. For $(d, n) = (5, 5)$, the maximum relative error in all coefficients was around 10^{-13}. Increasing n was not much worse, 10^{-11} maximum relative error for $(d, n) = (5, 10)$; although increasing d reduced accuracy to 10^{-8} maximum relative error for $(d, n) = (10, 5)$ and 10^{-5} maximum relative error for $(d, n) = (10, 10)$. Only the latter case took more than a few seconds, and it took hours on this simple machine. Clearly, this is a massive task where accuracy is a concern.

Future work should compare actual run time costs and accuracy between different methods implemented on the same platform. Of course, implementations should find many algorithmic details that can be improved. The general algorithms could be enhanced by taking advantage of more specifics, such as the choice of (w_i) values, sparsity in arrays, or symmetry properties of the function. The goal could be generalized to computing different subsets of the pyramid of derivatives, such as boxes. Certainly, the efficiency comparisons with other methods of computing truncated multivariate Taylor series show that this method is worth pursuing.

References

[1] Martin Berz, *Differential algebraic description of beam dynamics to very high orders*, Particle Accelerators **24** (1989), 109-124.

[2] Christian H. Bischof, George F. Corliss, and Andreas Griewank, *Structured second- and higher-order derivatives through univariate Taylor series*, Optimization Methods and Software **2** (1993), 211-232.

[3] Mariano Gasca and Thomas Sauer, *Polynomial interpolation in several variables*, Advances in Computational Math. **12** (2000), 377-410. MR2001d:41010

[4] Andreas Griewank, *Evaluating Derivatives*, SIAM, Philadelphia, 2000. MR2001b:65003

[5] Andreas Griewank, Jean Utke, and Andrea Walther, *Evaluating higher derivative tensors by forward propagation of univariate Taylor series*, Mathematics of Computation **69** (2000), 1117-1130. MR2000j:65033

[6] R.B. Guenther and E.L. Roetman, *Some observations on interpolation in higher dimensions*, Mathematics of Computation **24** (1970), 517-522. MR43:1384

[7] Eugene Isaacson and Herbert B. Keller, *Analysis of Numerical Methods*, Wiley, New York, 1966. MR34:924

[8] Kaiser S. Kunz, *Numerical Analysis*, McGraw-Hill, New York, 1957. MR19:460c

[9] David Kincaid and Ward Cheney, *Numerical Analysis*, Second Edition, Brooks/Cole, Pacific Grove, CA, 1996. MR97g:65003

[10] Ramon E. Moore, *Methods and Applications of Interval Analysis*, SIAM, Philadelphia, 1979. MR81b:65040

[11] Richard D. Neidinger, *An efficient method for the numerical evaluation of partial derivatives of arbitrary order*, ACM Trans. Math. Software 18 (1992), 159-173. MR93b:65040

[12] Richard D. Neidinger, *Computing Multivariable Taylor Series to Arbitrary Order*, APL Quote Quad **25** (1995), 134-144.

[13] Louis B. Rall, *Point and interval differentiation arithmetics*, in Automatic Differentiation of Algorithms, A. Griewank and G. F. Corliss, editors, SIAM, Philadelphia, 1991, 17-24. MR92k:65031

[14] Thomas Sauer, *Computational aspects of multivariate polynomial interpolation*, Advances in Computational Math. **3** (1995), 219-237. MR95k:65012

[15] A.N. Shevchenko and V.N. Rokityanskaya, *Automatic differentiation of functions of many variables*, Cybernetics and Systems Analysis **32** (1996), 709-724. MR98f:65004
[16] J.F. Steffensen, *Interpolation*, Second Edition, Chelsea Publishing, New York, 1950 (First Edition, 1927). MR12:164d
[17] Thomas Sauer and Yuan Xu, *On multivariate Lagrange interpolation*, Mathematics of Computation **64** (1995), 1147-1170. MR95j:41051
[18] I. Tsukanov and M. Hall, *Data Structure and Algorithms for Fast Automatic Differentiation*, Preliminary Draft (2001).

Department of Mathematics, Davidson College, Box 7002, Davidson, North Carolina 28035
E-mail address: rineidinger@davidson.edu