MATHEMATICS OF COMPUTATION
Volume 83, Number 289, September 2014, Pages 2403–2427
S 0025-5718(2014)02798-1
Article electronically published on January 6, 2014

FINITE DIFFERENCE WEIGHTS, SPECTRAL DIFFERENTIATION, AND SUPERCONVERGENCE

BURHAN SADIQ AND DIVAKAR VISWANATH

ABSTRACT. Let $z_1, z_2, \ldots, z_N$ be a sequence of distinct grid points. A finite difference formula approximates the m-th derivative $f^{(m)}(0)$ as $\sum w_k f(z_k)$, with w_k being the weight at z_k. We derive an algorithm for finding the weights w_k which uses fewer arithmetic operations and less memory than the algorithm in current use (Fornberg, *Mathematics of Computation*, vol. 51 (1988), pp. 699-706). The algorithm we derive uses fewer arithmetic operations by a factor of $(5m+5)/4$ in the limit of large N. The optimized C++ implementation we describe is a hundred to five hundred times faster than MATLAB. The method of Fornberg is faster by a factor of five in MATLAB, however, and thus remains the most attractive option for MATLAB users.

The algorithm generalizes easily to the calculation of spectral differentiation matrices, or equivalently, finite difference weights at several different points with a fixed grid. Unlike the algorithm in current use for the calculation of spectral differentiation matrices, the algorithm we derive suffers from no numerical instability.

The order of accuracy of the finite difference formula for $f^{(m)}(0)$ with grid points hz_k, $1 \le k \le N$, is typically $\mathcal{O}\left(h^{N-m}\right)$. However, the most commonly used finite difference formulas have an order of accuracy that is higher than normal. For instance, the centered difference approximation $(f(h)-2f(0)+f(-h))/h^2$ to $f''(0)$ has an order of accuracy equal to 2 not 1. Even unsymmetric finite difference formulas can exhibit such superconvergence or boosted order of accuracy, as shown by the explicit algebraic condition that we derive. If the grid points are real, we prove a basic result stating that the order of accuracy can never be boosted by more than 1.

1. Introduction

Since the beginning of the subject, finite difference methods have been widely used for the numerical solution of partial differential equations. Finite difference methods are easier to implement than finite element or spectral methods. For handling irregular domain geometry, finite difference methods are better than spectral methods but not as flexible as finite element discretizations.

The basic problem in designing finite difference discretizations is to approximate $f^{(m)}(0)$, the m-th derivative of the function $f(z)$ at $z = 0$, using function values at the grid points $hz_1, hz_2, \ldots, hz_N$. The grid points can be taken as $z_1, \ldots, z_N$ by setting the mesh parameter $h = 1$. We make the mesh parameter h explicit where necessary but suppress it otherwise. The finite difference formula can be given as

Received by the editor July 20, 2012 and, in revised form, December 22, 2012 and January 17, 2013.

2010 *Mathematics Subject Classification.* Primary 65D05, 65D25.

The authors were supported by NSF grants DMS-0715510, DMS-1115277, and SCREMS-1026317.

either

$$f^m(0) \approx w_{1,m} f(z_1) + \cdots + w_{N,m} f(z_N) \tag{1.1}$$

or

$$f^{(m)}(0) \approx \frac{w_{1,m} f(hz_1) + \cdots + w_{N,m} f(hz_N)}{h^m}. \tag{1.2}$$

If we require (1.2) to have an error that is $\mathcal{O}\left(h^{N-m}\right)$ for smooth f, the choice of the weights $w_{k,m}$, $1 \le k \le N$, is unique (see Section 5). The grid points are always assumed to be distinct.

Some finite difference formulas such as the centered difference approximations to $f'(0)$ and $f''(0)$—$(f(h) - f(-h))/2h$ and $(f(h) - 2f(0) + f(-h))/h^2$, respectively, occur very commonly and are part of the bread and butter of scientific computation. The most common finite difference formulas presuppose an evenly spaced grid. However, evenly spaced grids are often inadequate. In applications, it is frequently necessary to make the grid finer within boundary layers or internal layers where the underlying phenomenon is characterized by rapid changes. In addition, evenly spaced grids do not lend themselves to adaptive mesh refinement. For these reasons, it is often necessary to use grids that are not evenly spaced.

Fornberg [5–7] devised an algorithm for determining the weights $w_{i,m}$ given the grid points z_k. The starting point of his derivation was the Lagrange interpolant, which we now turn to.

There exists a unique polynomial $\pi(z)$ of degree $N-1$ which satisfies the interpolation conditions $\pi(z_k) = f_k$ for $k = 1 \ldots N$; see [4]. The Lagrange form of this interpolating polynomial is given by

$$\pi(z) = \sum_{k=1}^{N} w_k \pi_k(z) f_k \text{ where } \pi_k(z) = \prod_{j \neq k} (z - z_j) \text{ and } w_k = 1/\pi_k(z_k), \tag{1.3}$$

where w_k is Lagrange weight at z_k. The finite difference weight $w_{k,m}$ is equal to the coefficient of z^m in $w_k \pi_k(z)$ times $m!$ (see Section 8). The computation of the Lagrange weights w_k takes $2N^2$ arithmetic operations roughly half of which are multiplications and half are additions or subtractions.

In effect, Fornberg's algorithm [5] is to multiply the binomials $(z - z_j)$ using a recursion to determine the coefficient of z^m in the Lagrange cardinal function $w_k \pi_k(z)$. The algorithm is not presented in this way in [5]. Instead it is organized to yield the finite difference weights for partial lists of grid points $z_1, \ldots, z_k$ with k increasing from 1 to N. The exact operation count for Fornberg's algorithm is given in Section 3. The method of partial products derived in Section 3 always uses fewer arithmetic operations. The operation count is lower by a factor of $(5m+5)/4$, where m is the order of the derivative, in the limit of large N. Carefully timed trials in Section 7 show that it is faster in practice.

The polynomial $\pi(z)$ displayed in (1.3) is the Lagrange form of the interpolant as already stated. Another form of the polynomial interpolant is the Newton series. Most of the textbooks in numerical analysis recommend the Newton form over the Lagrange form believing the Newton form to be more accurate or more efficient. The beliefs are mistaken. In an expository paper, Berrut and Trefethen [1] have cogently summarized and clarified arguments showing that the Lagrange interpolant, if implemented correctly, is more efficient and has much better numerical stability. The Lagrange interpolant is also useful for root-finding; see [3] .

In their discussion, Berrut and Trefethen [1] show that it is advantageous to think of the weights w_k and the polynomials $\pi_k(z)$ which occur in (1.3) separately. The key idea in the method of partial products, which is derived in Section 3, is to think of the Lagrange weights $w_k = \prod_{j\neq k}(z_k - z_j)^{-1}$ separately from $\pi_k(z) = \prod_{j\neq k}(z - z_j)$, which is a product of the binomials $z - z_j$.

The method of partial products gets its name because it is based on the following partial products:

$$l_k(z) = \prod_{j=1}^{k}(z - z_j) \text{ and } r_k(z) = \prod_{j=k}^{N}(z - z_j).$$

By convention, $l_0 \equiv r_{N+1} \equiv 1$. For $k = 1, \ldots, N$, coefficients of these partial products are computed up to the z^M term using recursions. Since $\pi_k = l_{k-1}r_{k+1}$, the finite difference weights $w_{k,m}$ for $m = 0, \ldots, M$, are obtained by convolving the coefficients of l_{k-1} and r_{k+1} followed by a multiplication by $m!$ and the Lagrange weight w_k.

At first, forming partial products may seem roundabout. A more direct and more efficient approach would be to form the product

$$p(z) = \prod_{j=1}^{N}(z - z_j) \tag{1.4}$$

up to the $(M+1)$-st power, where M is the order of the derivative, and then calculate the coefficient of z^M in $\pi_k(z)$ using $\pi_k = (z - z_k)^{-1}p(z)$. However, there are subtle issues of numerical stability at play here. The direct method outlined suffers from a numerical instability whose harmful effect on accuracy increases exponentially in M. The method of partial products, on the other hand, seems to have very good numerical stability (see Section 6). A discussion of the subtle issues of numerical stability at play is given in Section 2.

When spectral differentiation matrices are computed, the grid $z_1, \ldots, z_N$ is fixed. However, the finite difference weights are required at each of the grid points and not just at $z = 0$ as we assumed in the discussion following (1.3). The algorithm in current use for this problem [13,14] suffers from a numerical instability that worsens exponentially in M, where M is the order of the derivative, as shown in Section 6. This instability appears identical to that of the method based on the product (1.4) outlined in the previous paragraph. The key observation for computing spectral differentiation matrices using the method of partial products is that the Lagrange weights w_k do not change at all if the grid $z_1, \ldots, z_N$ is shifted to $z_1 - \zeta, \ldots, z_N - \zeta$. Supported by this observation, the method of partial products can compute spectral differentiation matrices in cost comparable to the method of Welfert [14] but with no numerical instability. The computation of spectral differentiation matrices using partial products is described in Section 4.

If the number of grid points is N and the order of the derivative is m, the finite difference weights are unique if the difference formula is required to be $\mathcal{O}\left(h^{N-m}\right)$ (see Section 5). For certain grids, these unique weights imply an error of $\mathcal{O}\left(h^{N-m+1}\right)$, which is of higher order than what is typical for N grid points and the m-th derivative. We term this as superconvergence or boosted order of accuracy.

Finite difference formulas that exhibit superconvergence or boosted order of accuracy have been quite popular. The centered difference formulas for $f'(0)$ and

$f''(0)$ are only two examples. Yet there is little clarity about what causes such boosted order of accuracy. There is an incorrect folklore belief that boosted order of accuracy has something to do with the symmetry of the grid.

In Section 5, we clarify the situation completely and give explicit conditions for boosted order of accuracy. The finite difference approximation (1.2) to $f^{(m)}(0)$ has an order of accuracy boosted by 1 if and only if

$$S_{N-m} = 0,$$

where S_k is the elementary symmetric function

$$\sum_{1 \le i_1 < \cdots < i_k \le N} z_{i_1} \ldots z_{i_k}.$$

If the grid points are real, we prove that the order of accuracy cannot be boosted by more than 1.

For the special case $m = 2$, the finite difference approximation to the second derivative at $z = 0$ using three grid points has an order of accuracy equal to 2, which is a boost of 1, if and only if the grid points satisfy $z_1 + z_2 + z_3 = 0$. Evidently, this condition is satisfied by the grid points $-1, 0, 1$ used by the centered difference formula. An unsymmetric choice of grid points such as $-3, 1, 2$ also boosts the order of accuracy by 1. However, no choice of z_1, z_2, and z_3 on the real line can boost the order of accuracy by more than 1.

With four grid points and $m = 2$, the condition for a boost in the order of accuracy is

$$z_1 z_2 + z_1 z_3 + z_1 z_4 + z_2 z_3 + z_2 z_4 + z_3 z_4 = 0.$$

No choice of z_1, z_2, z_3, and z_3 on the real line can boost the order of accuracy of the finite difference approximation to $f''(0)$ by more than 1. The maximum possible order of accuracy is 3. In this case, a symmetric choice of grid points such as -2, -1, 1, 2 does not boost the order of accuracy. Unsymmetric grid points that boost the order of accuracy to 3 can be found easily. For example, the order of accuracy is 3 for the grid points $-2/3$, 0, 1, 2.

These results about superconvergence or boosted order of accuracy of finite difference formulas are quite basic. It is natural to suspect that they may have been discovered a long time ago. However, the results are neither stated nor proved in any source that we know of.

If the grid points z_i are allowed to be complex, the order of accuracy can be boosted further but not by more than m. The order of accuracy is boosted by k with $1 \le k \le m$ if and only if

$$S_{N-m} = S_{N-m+1} = \cdots = S_{N-m+k-1} = 0.$$

For complex grid points, the maximum boost in the order of accuracy is obtained when the grid points are arranged symmetrically on a circle centered at 0, with 0 being the point at which the derivative is to be approximated.[1] An algorithm to detect the order of accuracy and compute the error constant of the finite difference formula (1.2) is given in Section 5.

[1]We thank Professor Jeffrey Rauch for this observation.

2. From roots to coefficients

Given $\alpha_1, \ldots, \alpha_N$, the problem is to determine the coefficients of a polynomial of degree N whose roots are $\alpha_1, \ldots, \alpha_N$. The polynomial is evidently given by $\prod_{k=1}^{N}(z - \alpha_k)$. If the product $\prod_{k=1}^{n}(z - \alpha_k)$ is given by $c_0 + c_1 z + \cdots + z^n$, then the coefficients $c'_0, c'_1, \ldots$ of the product $\prod_{k=1}^{n+1}(z - \alpha_k)$ are formed using

$$c'_0 = -c_0\alpha_{n+1} \quad \text{and} \quad c'_m = -c_m\alpha_{n+1} + c_{m-1} \quad \text{for} \quad m = 1, 2, \ldots. \tag{2.1}$$

All algorithms to compute finite difference weights come down to multiplying binomials of the form $(z - z_k)$ and extracting coefficients from the product. The numerical stability of multiplying binomials, or equivalently of going from roots of a polynomial to its coefficients, has aspects that are not obvious at first sight. A dramatic example is the product $\left(z - \omega^0\right)\left(z - \omega^1\right)\ldots\left(z - \omega^{N-1}\right)$ where $\omega = \exp(2\pi i/N)$. Mathematically the answer is $z^N - 1$. Numerically the error is as high as 10^{15} for $N = 128$ in double precision arithmetic [2]. For numerical stability, the binomials must be ordered using the bit reversed ordering or the Leja ordering or some other scheme as shown by Calvetti and Reichel [2]. The roots must be ordered in such a way that the coefficients of intermediate products are not too large. If N is large, the first several ω^i are close to 1 leading to partial products which resemble $(z - 1)^n$ and have coefficients that are of the order of the binomial coefficients. In contrast, the complete product is simply $z^N - 1$.

This matter of ordering the roots carefully is equivalent to choosing a good order of grid points when determining finite difference weights. Good ordering of grid points may improve accuracy but is not as important as it is in the general problem of determining coefficients from roots. When determining finite difference weights for a derivative of order M, we need coefficients of terms $1, z, \ldots, z^M$ but no higher. The most dramatic numerical instabilities in determining coefficients occur near the middle of the polynomial, but M, which is the order of differentiation, will not be large in the determination of finite difference weights. Nevertheless, the instability which results if grid points are not ordered properly is noticeable even for $M = 2$ and can be quite harmful if $M = 4$ or $M = 8$ if we use Chebyshev points with $N = 128$, for instance.

The recurrence (2.1) may be used to multiply binomials only if the roots have been ordered. It might seem better to use a method that does not impose an ordering on the roots. We derive such a method below, but find it to be numerically unstable. The cause of numerical instability is once again relevant to the computation of finite difference weights.

Let

$$\begin{aligned} \mathcal{P}_r &= \sum_{k=1}^{N} \alpha_k^{-r}, \\ \mathcal{E}_r &= \sum_{1 \le i_1 < \cdots < i_r \le N} \left(\alpha_{i_1} \ldots \alpha_{i_r}\right)^{-1}. \end{aligned}$$

By the Newton identities

$$\begin{aligned} \mathcal{E}_1 &= \mathcal{P}_1, \\ 2\mathcal{E}_2 &= \mathcal{E}_1\mathcal{P}_1 - \mathcal{P}_2, \\ 3\mathcal{E}_3 &= \mathcal{E}_2\mathcal{P}_1 - \mathcal{E}_1\mathcal{P}_2 + \mathcal{P}_3, \end{aligned}$$

and so on. By convention, $\mathcal{E}_0 = 1$. The algorithm begins by computing the power sums $\mathcal{P}_1, \ldots, \mathcal{P}_M$ directly and uses the Newton identities to compute the elementary symmetric functions $\mathcal{E}_r$, $0 \leq r \leq M$. The coefficients are obtained using

$$c_r = (-1)^{N+r-1} \mathcal{E}_r \prod_{k=1}^{N} \alpha_k.$$

This algorithm does not really presuppose an ordering of the α_k and the computation of the power sums $\mathcal{P}_r$ is backward stable, and especially so if compensated summation is used [9]. If this method is used to compute the product $\prod_{k=1}^{N} \left(z - \omega^{k-1}\right)$, where ω is as before, it finds the coefficients of the product with excellent accuracy. But in general this method is inferior to the repeated use of (2.1) after choosing a good ordering of the roots α_k.

Why is a method which appears so natural numerically unstable? The culprit is the use of Newton identities. The Newton identities evidently have the structure of triangular back substitution, as the elementary symmetric functions $\mathcal{E}_1, \ldots, \mathcal{E}_k$ are used to compute $\mathcal{E}_{k+1}$. Thus, in effect, the Newton identities are inverting a triangular matrix. Unfortunately, triangular matrices typically have condition numbers that increase exponentially in the dimension of the matrix [12]. For example, a triangular matrix with independent standard normal entries has a condition number that increases as 2^n, where n is the dimension of the matrix [12]. The triangular matrix that is implicit in the Newton identities will not follow any of the random distributions considered in [12]. However, the conclusion that triangular matrices are exponentially ill-conditioned still applies.

In the computation of finite difference weights, it is often tempting to divide by a polynomial. For example, the polynomial $\pi_k(z)$ that appears in the Lagrange interpolant $\pi(z)$ in (1.3) may be obtained by forming the product $\prod_{j=1}^{N}(z - z_j)$ up to whatever power is desired and then dividing by $(z - z_k)$ for each k. This division operation involves back substitution and is a source of numerical instability for the reason given in the previous paragraph.

Suppose $a_0 + a_1 z + \cdots = (z - \alpha)^{-1} (b_0 + b_1 z + \cdots)$. To find the a_j in terms of the b_j the following equations may be used (assuming $\alpha \neq 0$):

$$\begin{aligned} a_0 &= -b_0/\alpha, \\ a_j &= (a_{j-1} - b_j)/\alpha \quad \text{for } j = 1, 2, \ldots. \end{aligned}$$

The equation for calculating a_j evidently uses a_{j-1}. Thus the equations are implicitly using back substitution to invert a bi-diagonal triangular matrix. In contrast, (2.1) does not use any one of $c'_0, \ldots, c'_{m-1}$ to compute c'_m and is therefore numerically stable if the grid points are ordered following the prescriptions of Calvetti and Reichel [2].

In this section, we have discussed numerical instabilities that arise when binomials are multiplied and when a polynomial is divided by a binomial. The first of these instabilities is overcome by ordering the grid points carefully. The method of partial products for computing finite difference weights, to which we now turn, avoids back substitution entirely and is therefore not plagued by the second instability.

3. Finite difference weights using partial products

Let the grid points be $z_1, \ldots, z_N$ and let $f_1, \ldots, f_N$ be the function values at the grid points. Define

$$\pi_k(z) = \prod_{j \neq k}^{N} (z - z_j). \tag{3.1}$$

Then the Lagrange interpolant shown in (1.3) is $\pi(z) = \sum_{k=1}^{N} w_k \pi_k(z) f_k$. The Lagrange weight w_k equals $1/\pi_k(z_k)$. Our objective is to derive formulas for $d^m\pi(z)/dz^m$ at $z = 0$ for $m = 1, \ldots, M$. The $m = 0$ case is regular Lagrange interpolation. The weights w_k will be assumed to be known. The formulas for $d^m\pi(z)/dz^m$ at $z = 0$ will be linear combinations of f_k with weights. We assume $1 \leq M \leq N - 1$.

If the coefficient of z^m in $\pi_k(z)$ is denoted by $c_{k,m}$, we have

$$\left.\frac{d^m\pi(z)}{dz^m}\right|_{z=0} = m! \sum_{k=1}^{N} c_{k,m} w_k f_k.$$

The finite difference weights are then given by

$$w_{k,m} = m! w_k c_{k,m}. \tag{3.2}$$

Once the $c_{k,m}$ are known, the weights $w_{k,m}$ are computed using (3.2) for $k = 1, \ldots, N$ and $m = 1, \ldots, M$.

In the Introduction, we mentioned the basic plan of our method for computing finite difference weights. The basic plan is to separate the computation of the Lagrange weights w_k from that of the coefficients $c_{k,m}$ of $\pi_k(z)$, which is defined as a product of binomials. This plan is already realized in (3.2) where w_k and $c_{k,m}$ occur separately in the formula for $w_{k,m}$. To complete a description of the algorithm, it suffices to show how the $c_{k,m}$ are computed, which we do presently.

Let $l_k(z) = \prod_{j=1}^{k}(z - z_j)$ and $r_k(z) = \prod_{j=k}^{N}(z - z_j)$. Denote the coefficients of $1, z, \ldots, z^M$ in $l_k(z)$ and $r_k(z)$ by $L_{k,0}, \ldots, L_{k,M}$ and $R_{k,0}, \ldots, R_{k,M}$, respectively. The coefficients $L_{k,m}$ are computed in the order $k = 1, 2, \ldots, N$. The coefficients $R_{k,m}$ are computed in the reverse order, which is $k = N, N-1, \ldots, 1$. It is evident that $\pi_k(z)$, which is defined by (1.3) or (3.1), is equal to $l_{k-1}(z) r_{k+1}(z)$. Therefore the coefficient $c_{k,m}$ of z^m in $\pi_k(z)$ can be obtained using

$$c_{k,m} = \sum_{s=0}^{m} L_{k-1,m-s} R_{k+1,s}.$$

The finite difference weight $w_{k,m}$ is obtained as $m! w_k c_{k,m}$, where w_k is the Lagrange weight at z_k.

Lemma 1. *The number of arithmetic operations used by the method of partial products (Algorithm 1) to compute the finite difference weights $w_{k,m}$, $0 \leq m \leq M$, $1 \leq k \leq N$, is fewer than $2N^2 + NM^2 + 8NM - 4M^2 - N + 2M + 2$.*

Proof. The computation of Lagrange weights (function on line 1 and function call on line 18 of Algorithm 1) uses $2N^2 - 2N$ arithmetic operations.

The computation of $L_{k,m}$, the coefficients of $l_k(z)$, uses the `MULTBINOM` function on line 7 called from line 21. For easier reading, Algorithm 1 treats each $l_k(z)$ as

Algorithm 1 The method of partial products for finding finite difference weights

1: **function** LAGRANGEWEIGHTS($z_1, \ldots, z_N, w_1, \ldots, w_N$)
2: **for** $i = 1, 2, \ldots, N$ **do**
3: $w_i = \prod_j (z_i - z_j)$ over $j = 1, \ldots, N$ but $j \neq i$.
4: $w_i = 1/w_i$
5: **end for**
6: **end function**
7: **function** MULTBINOM($a_0, \ldots, a_M, b_0, \ldots, b_M, \zeta$)
8: $b_0 = -\zeta\, a_0$
9: $b_m = -\zeta\, a_m + a_{m-1}$ for $m = 1, \ldots, M$
10: **end function**
11: **function** CONVOLVE($a_0, \ldots, a_M, b_0, \ldots, b_M, c_0, \ldots, c_M$)
12: $c_m = a_m b_0 + a_{m-1} b_1 + \cdots + a_0 b_m$ for $m = 0, \ldots, M$
13: **end function**
14: **function** FDWEIGHTS($z_1, \ldots, z_N, M, w_{1,M}, \ldots, w_{N,M}$)
15: Temporaries: $w_1, \ldots, w_N$
16: Temporaries: $L_{k,m}$ and and $R_{k,m}$ for $0 \leq k \leq N+1$, $0 \leq m \leq M$
17: Finite difference weights: $w_{k,m}$ for $1 \leq k \leq N$ and $0 \leq m \leq M$
18: LAGRANGEWEIGHTS($z_1, \ldots, z_N, w_1, \ldots, w_N$)
19: $L_{0,m} = 1$ for $m = 0$ and $L_{k,m} = 0$ for $m = 1, \ldots, M$
20: **for** $k = 1, \ldots, N-1$ **do**
21: MULTBINOM($L_{k-1,0}, \ldots, L_{k-1,M}, L_{k,0}, \ldots, L_{k,M}, z_k$)
22: **end for**
23: $R_{N+1,m} = 1$ for $m = 0$ and $R_{N+1,m} = 0$ for $m = 1, \ldots, M$.
24: **for** $k = N, N-1, \ldots, 2$ **do**
25: MULTBINOM($R_{k+1,0}, \ldots, R_{k+1,M}, R_{k,0}, \ldots, R_{k,M}, z_k$)
26: **end for**
27: **for** $k = 1, \ldots, N$ **do**
28: Temporaries: $c_{k,m}$
29: CONVOLVE($L_{k-1,0}, \ldots, L_{k-1,M}, R_{k+1,0}, \ldots, R_{k+1,M}, c_{k,0}, \ldots, c_{k,M}$)
30: $w_{k,m} = m! w_k c_{k,m}$ for $m = 0, \ldots, M$ or simply $w_{k,M} = M! w_k c_{k,M}$
31: **end for**
32: **end function**

being truncated at the M-th power. Since the degree of $l_k(z)$ is k, an implementation can truncate earlier for $k < M$. Multiplying a polynomial by a binomial of the form $(z-\alpha)$ up to the k-th power costs fewer than $2k$ operations. The total cost for computing l_k, $1 \leq k < N$, is fewer than $\sum_{n=1}^{M} 2n + (N-M)(2M) = 2NM - M^2 + M$ operations. The cost of computing r_k, $1 < k \leq N$, is the same.

The convolution function on line 11 is called from line 29 to form the coefficients of the product $l_{k-1} r_{k+1}$ for $1 \leq k \leq N$. The cost is fewer than $N\left(\sum_{m=0}^{M} 2m + 1\right) = NM^2 + 2NM + N$ operations.

In this last bound, we have taken the operation count of a single convolution to be $M^2 + 2M + 1$. However, the convolutions that correspond to $l_1 r_3$ and $l_{N-1} r_{N+1}$ are really multiplications by binomials, each with a count of $2M + 1$. Thus we may subtract $2M^2$ from the grand total.

Assuming the weights $w_{k,m}$ are computed for $0 \leq m \leq M$ and $1 \leq k \leq N$, the rescaling on line 30 costs $2NM$ floating point operations. □

Lemma 2. *Fornberg's method for computing $w_{k,m}$, $0 \le m \le M$, $1 \le k \le N$, uses*

$$\left(\frac{5M+5}{2}\right)N^2 + \left(\frac{7M+3}{2}\right)N - 5M^3/6 - 3M^2 - 13M/6 - 4$$

arithmetic operations.

Proof. The calculation here is with reference to the pseudo-code on page 700 of [5]. The arithmetic operations with $c3$ and $c2$ on the left-hand side add up to $N(N+1)$. The number of arithmetic operations that correspond to $\delta^m_{n-1,\nu}$ with $m > 0$ is $\sum_{n=1}^{N} 5\min(n, M)n$ and the count with $m = 0$ is $3N(N+1)/2$. The number of arithmetic operations that correspond to $\delta^m_{n,n}$ is $6(NM - M^2/2 + M/2)$ assuming $m > 0$ and is $4N$ for $m = 0$. The lemma gives the grand total with N replaced by $N-1$ since the pseudo-code in [5] is given for $N+1$ grid points. □

Lemma 3. *The method of partial products uses fewer arithmetic operations than Fornberg's method except when $N = 2$ and $M = 1$.*

Proof. If $N \ge 6$, a tedious but elementary calculation, which we omit, shows that the method of partial products uses fewer arithmetic operations. The other cases can be checked manually. □

In the limit of large N, the method of partial products uses fewer arithmetic operations by a factor of $4/(5M+5)$. In Section 7, we shall see that the method of partial products runs faster than Fornberg's method by a similar factor.

If the method of partial products is required to return the finite difference weights for all derivatives of order m, with $0 \le m \le M$, it uses slightly more space than Fornberg's method. While Fornberg's method uses space equal to $N(M+1)$ numbers, the method of partial products requires storage for about $2M$ additional doubles to store the r_k and the Lagrange weights. In Section 7, we report timing data from two optimized implementations of the method of partial products. The optimized C++ implementation, which returns the finite difference weights only for derivatives of order M, uses less space than Fornberg's method. The other implementation returns finite difference weights for all orders up to M. It uses $\mathcal{O}(NM)$ scratch space and not the best possible scratch space of $2M$ numbers. Scratch space is nearly always cached and any attempt to reduce it will bring no benefit. The operation count for either implementation of the method of partial products is almost certainly less than the upper bound given in Lemma 1.

4. Spectral differentiation matrices

The method of partial products, which is presented as Algorithm 1, computes the finite difference weights for the M-th derivative at $z = 0$ in two stages. The first stage is the computation of Lagrange weights w_k which correspond to the grid points $z_1, \ldots, z_N$. The second stage is the computation of the partial products $l_k(z)$ and $r_k(z)$ as well as the finite difference weights $w_{k,M}$. The operation count given in Lemma 1 can be divided into $2N^2$ for the first stage and $NM^2 + 8NM$ for the second state.

If finite difference weights are desired at the point $z = \zeta$, the same algorithm can be used but the grid points must be shifted to $z_1 - \zeta, \ldots, z_N - \zeta$. The Lagrange weights do not change and do not need to be recomputed. However, the partial products change and the second stage of the method must be executed with the

shifted grid points. If finite difference weights are computed at p points, the total cost is $2N^2 + NM^2p + 8NMp$ operations with some low order terms omitted.

The finite difference weights can be computed at each of the grid points $z_1, \ldots, z_N$ and arranged in an $N \times N$ spectral differentiation matrix. The cost of computing a spectral differentiation matrix using the method of partial products is $N^2\left(2 + 8NM + NM^2\right)$. Welfert [14] has derived an algorithm whose total cost, according to Weideman and Reddy [13], is $N^2(4 + 7NM)$.

Welfert's method has a lower cost than the method of partial products. However, it has a numerical instability that grows exponentially with M (see Section 6). If that type of numerical instability is deemed to be acceptable, which we do not, a direct use of (1.4) leads to a method which has an even lower operation count of $2N^2 + 6N^2M$ and which is particularly easy to implement. Welfert [14] pointed out that the errors become "very important" for $M > 6$. In Section 6, we find that Welfert's method may have slightly higher errors for $M = 2$ and that its errors increase somewhat erratically with N for $M = 4$. The errors are quite bad for $M = 8$ and $M = 16$. The method of partial products appears numerically stable for computing finite difference weights for all M (it must be noted however that ill-conditioning is intrinsic to the computation of high derivatives).

5. Superconvergence or boosted order of accuracy

Let $z_1, \ldots, z_N$ be distinct grid points. Let

$$f^{(m)}(0) \approx \frac{w_{1,m} f(hz_1) + \cdots + w_{N,m} f(hz_N)}{h^m} \tag{5.1}$$

be an approximation to the m-th derivative at 0. We begin by looking at the order of accuracy of this approximation. Here (1.2) is shown again as (5.1) for convenience. The order of the derivative m is assumed to satisfy $m \leq N - 1$. The case $m = 0$ corresponds to interpolation. The allowed values of m are from the set $\{1, 2, \ldots, N-1\}$.

Lemma 4. *The finite difference formula* (5.1) *has an error of* $\mathcal{O}\left(h^{N-m}\right)$ *if and only if*

$$\sum_{k=1}^{N} w_{k,m} x_k^m = m! \quad \text{and} \quad \sum_{k=1}^{N} w_{k,m} x_k^n = 0$$

for $n \in \{0, 1, \ldots, N-1\} - \{m\}$. *The function* f *is assumed to be* N *times continuously differentiable.*

Proof. Assume that the weights $w_{k,m}$ satisfy the conditions given in the lemma. The function $f(z)$ can be expanded using Taylor series as $f(0) + f'(0)z + \cdots + f^{(N-1)}(0)z^{N-1}/(N-1)! + z^N g(z)$, where $g(z)$ is a continuous function. In particular, $g(z)$ is continuous at $z = 0$. If the Taylor expansion is substituted into the right-hand side of (5.1) and the conditions satisfied by the weights are used, we get the following expression:

$$f^{(m)}(0) + h^{N-m}\left(w_{1,m} z_1^N g(hz_1) + \cdots + w_{N,m} z_N^N g(hz_N)\right).$$

The coefficient of h^{N-m} is bounded in the limit $h \to 0$, and therefore the error is $\mathcal{O}\left(h^{N-m}\right)$.

The necessity of the conditions on the weights $w_{k,m}$ is deduced by applying the finite difference formula (5.1) to $f = 1, z, \ldots, z^{N-1}$. □

The conditions on the weights in Lemma 5 correspond to the matrix system

$$\begin{pmatrix} 1 & 1 & \cdots & 1 \\ z_1 & z_2 & \cdots & z_N \\ & & \cdots & \\ z_1^{N-1} & z_2^{N-1} & \cdots & z_N^{N-1} \end{pmatrix} \begin{pmatrix} w_{1,m} \\ w_{2,m} \\ \vdots \\ w_{N,m} \end{pmatrix} = m! e_m, \tag{5.2}$$

where e_m is the unit vector with its m-th entry equal to 1. The matrix here is the transpose of the well-known Gram or Vandermonde matrix.

Newton and Lagrange interpolations are techniques for solving Vandermonde systems. Newton interpolation is equivalent to an LU decomposition of the Gram or Vandermonde matrix [4]. Partly because the matrix in (5.2) is the transpose of the Gram or Vandermonde matrix, the interpolation techniques are not directly applicable.

The Gram or Vandermonde determinant equals $\prod_{1 \le i < j \le N} (z_j - z_i)$ and is therefore nonsingular [4]. Thus we have the following theorem.

Theorem 1. *There exists a unique choice of weights $w_{k,m}$, $k = 1, \ldots, N$, such that the finite difference formula* (5.1) *has error $\mathcal{O}\left(h^{N-m}\right)$.*

This theorem is trivial and generally known. However, its clear formulation is essential for developments that will follow. Our main interest is in boosted order of accuracy.

Lemma 5. *The finite difference formula* (1.2) *has boosted order of accuracy with an error of $\mathcal{O}\left(h^{N-m+b}\right)$, where b is a positive integer, if and only if the weights $w_{k,m}$ satisfy*

$$w_{1,m} z_1^{N-1+\beta} + \cdots + w_{N,m} z_N^{N-1+\beta} = 0$$

for $\beta = 1, \ldots, b$ in addition to the conditions of Lemma 4.

Proof. Similar to the proof of Lemma 4. □

To derive conditions for boosted order of accuracy that do not involve the weights, we introduce the following notation. By

$$\det\left(z_1, z_2 \ldots z_N; n_1, n_2, \ldots, n_N\right) \tag{5.3}$$

we denote the determinant of the $N \times N$ matrix whose (i,j)-th entry is $z_j^{n_i}$. The transpose of the Vandermonde or Gram determinant of the grid points, which occurs in (5.2), is $\det(z_1, \ldots, z_N; 0, \ldots N-1)$ in this notation.

Theorem 2. *Let $w_{k,m}$, $k = 1, \ldots, N$, be the unique solution of* (5.2) *so that the finite difference formula* (5.1) *has an order of accuracy that is at least $N - m$. The order of accuracy is boosted by b, where b is a positive integer, if and only if*

$$\det\left(z_1, \ldots, z_N; [0, 1, \ldots, N-1, N-1+\beta] - m\right) = 0$$

for $\beta = 1, \ldots, b$. Here $[0, 1, \ldots, N-1, N-1+\beta] - m$ denotes the sequence $0, 1, \ldots, N-1, N-1+\beta$ with m deleted.

Proof. First, assume the weights $w_{k,m}$ and the grid points z_k to be real. The condition of Lemma 5 requires that the row vector $W_m = [w_{1,m}, \ldots, w_{N,m}]$ be orthogonal to

$$\left[z_1^{N-1+\beta}, \ldots, z_N^{N-1+\beta}\right]. \tag{5.4}$$

By (5.2), W_m is orthogonal to every row of the Gram matrix except the m-th row. Since the Gram matrix is nonsingular, the rows of that matrix are a linearly independent basis. Consequently, the $N-1$-dimensional space of vectors orthogonal to W_m is spanned by the rows of the Gram matrix with the m-th row excepted. The vector (5.4) is orthogonal to W_m if and only if it lies in the span of the vectors

$$[z_1^n, \ldots, z_N^n] \quad n \in \{0, 1 \ldots N-1\} - \{m\}\,. \tag{5.5}$$

Thus the condition of Lemma 5 holds if and only if the determinant of the $N \times N$ matrix whose first $(N-1)$ rows are the vectors (5.5) and whose last row is (5.4) vanishes as stated in the theorem.

If the weights and the grid points are complex, the same argument can be repeated after replacing the weights by their complex conjugates in the definition of W_m. □

Theorem 2 gives determinantal conditions for boosted order of accuracy. We will cast those conditions into a more tractable algebraic form. The following theorem gives the template for the algebraic form into which the conditions of Theorem 2 will be cast.

Theorem 3. *If $n_1, n_2, \ldots, n_N$ are distinct positive integers, the determinant* (5.3) *can be factorized as*

$$\prod_{1 \le i < j \le N} (z_j - z_i)\; S(z_1, \ldots z_N)\,,$$

where $S(z_1, \ldots z_N)$ is a symmetric polynomial that is unchanged when $z_1, \ldots, z_N$ are permuted. All the coefficients of S are integers.

Proof. We will work over $\mathbb{Q}$, the field of rational numbers. We can think of the determinant (5.3) as a polynomial in z_N with coefficients in the field $\mathbb{Q}(z_1, \ldots, z_{N-1})$. Since the determinant (5.3) vanishes, if z_N is equal to any one of $z_1, \ldots, z_{N-1}$, we have that the determinant can be factorized as

$$(z_N - z_1)(z_N - z_2) \ldots (z_N - z_{N-1})\, f,$$

where f is an element of the field $\mathbb{Q}(z_1, \ldots, z_{N-1})$. By Gauss's lemma, f should in fact be an element of $\mathbb{Z}[z_1, \ldots, z_{N-1}]$, the ring of polynomials in $z_1, \ldots, z_{N-1}$ with integer coefficients (for Gauss's lemma, see Section 2.16 of [10] and in particular the corollary at the end of that section). Now f can be considered as a polynomial in z_{N-1} and factorized similarly, and so on, until we get a factorization of the form shown in the theorem.

To prove that S is symmetric, consider a transposition that switches z_p and z_q. The determinant (5.3) changes sign by a familiar property of determinants. The product of all pairwise differences $z_j - z_i$ also changes sign as may be easily verified or as may be deduced by noting that the product is the Gram or Vandermonde determinant. Therefore S is unchanged by transpositions and is a symmetric function. □

Remark 1. For the determinants that arise as conditions for boosted order of accuracy in Theorem 2, we describe a method to compute the symmetric polynomial S explicitly. The symmetric polynomials that arise in Theorem 3 are the Schur functions of symmetric function theory. The connection to symmetric functions is possibly of use for generalizations to higher dimensions.

To begin with, let us consider the Gram determinant

$$\det(z_1, \ldots, z_N, z_{N+1}; 0, \ldots, N-1, N). \tag{5.6}$$

This determinant is equal to

$$\prod_{1 \le i < j \le N} (z_j - z_i) \times \prod_{k=1}^{N} (z_{N+1} - z_k). \tag{5.7}$$

See [4, p. 25]. By expanding (5.6) using the entries of the last column (each of these entries is a power of z_{N+1}), we deduce that the coefficient of z_{N+1}^m in the expansion of (5.6) is equal to

$$(-1)^{N+m} \det(z_1, \ldots, z_N; [0, \ldots N-1, N] - m). \tag{5.8}$$

This determinant is the minor that corresponds to the entry z_{N+1}^m in the expansion of (5.6). By inspecting (5.7), we deduce that the coefficient of z_{N+1}^m in that expression is equal to

$$\prod_{1 \le i < j \le N} (z_j - z_i) \times (-1)^{N-m} S_{N-m}, \tag{5.9}$$

where

$$S_p = \sum_{1 \le i_1 < \cdots < i_p \le N} z_{i_1} \ldots z_{i_p}.$$

Thus S_p denotes the sum of all possible terms obtained by multiplying p of the grid points $z_1, \ldots, z_N$. For future use, we introduce the notation S_p^+ for the sum of all possible terms obtained by multiplying p of the numbers $z_1, \ldots, z_N, z_{N+1}$.

Theorem 4. *The finite difference formula* (5.1) *with distinct grid points* z_k *and weights* $w_{k,m}$ *that satisfy* (5.2) *has an order of accuracy that is boosted by* 1 *if and only if* $S_{N-m} = 0$.

Proof. The condition for a boost of 1 is obtained by setting $\beta = 1$ in Theorem 2. By equating (5.8) with (5.9), we get

$$\det(z_1, \ldots, z_N; [0, \ldots N-1, N] - m) = \prod_{1 \le i < j \le N} (z_j - z_i) \times S_{N-m}. \tag{5.10}$$

Since the grid points are distinct, the determinant is zero if and only if $S_{N-m} = 0$. □

The corollary that follows covers all of the popular cases that have boosted order of accuracy.

Corollary 1. *If the grid points* $z_1, \ldots, z_N$ *are symmetric about* 0 *(in other words* z *is a grid point if and only if* $-z$ *is a grid point) and* $N - m$ *is odd, the order of accuracy is boosted by* 1.

Although we have restricted m to be in the set $\{1, 2, \ldots, N-1\}$, Theorems 2 and 4 hold for the case $m = 0$ as well. The case $m = 0$ of (5.1) corresponds to interpolation. According to Theorem 4, the interpolation has boosted order of accuracy if and only if $S_N = 0$ or one of the grid points is zero. Of course, the interpolant at zero is exact if zero is one of the grid points. We do not consider the case $m = 0$ any further.

To derive an algebraic condition for the order of accuracy to be boosted by 2, we apply the identity (5.10) with grid points $z_1, \ldots, z_N, z_{N+1}$ and rewrite it as follows:

$$\det(z_1, \ldots, z_N, z_{N+1}; [0, \ldots, N-1, N, N+1] - m) = \prod_{1 \le i < j \le N} (z_j - z_i) \times S^+_{N-m+1} \times \prod_{k=1}^{N} (z_{N+1} - z_k).$$

We equate the coefficients of z_{N+1}^N to deduce that

(5.11)
$$\det(z_1, \ldots, z_N; [0, \ldots N-1, N+1] - m) = \prod_{1 \le i < j \le N} (z_j - z_i) \times (S_1 S_{N-m} - S_{N-m+1}).$$

To obtain this identity, we assumed $m \ge 1$ and used $S^+_{N-m+1} = S_{N-m+1} + z_{N+1} S_{N-m}$.

Lemma 6. *The order of accuracy of the finite difference formula* (5.1) *is boosted by* 2 *if and only if* $S_{N-m} = 0$ *and* $S_{N-m+1} = 0$.

Proof. We already have the condition $S_{N-m} = 0$ for the order of accuracy to be boosted by 1. By Theorem 2, the order of accuracy is boosted by 2 if and only if the determinant of (5.11) is zero as well. Since $S_{N-m} = 0$, by (5.11), it is equivalent to $S_{N-m+1} = 0$. □

Theorem 5. *The order of accuracy of the finite difference formula* (5.1) *for the* m*-th derivative can never be boosted by more than* 1 *as long as the grid points are real. Here* $m \ge 1$.

Proof. By the preceding lemma, the grid points $z_1, \ldots, z_N$ must satisfy $S_{N-m} = 0$ and $S_{N-m+1} = 0$ for the order of accuracy to be boosted by more than 1. We show that cannot happen by closely following the proof of Newton's inequality (see Theorem 144 on page 104 of [8]) .

Let $f(z)$ denote $(z - z_1) \ldots (z - z_N)$, which is a polynomial of degree N with N distinct real roots. Define $g(x, y) = y^N f(x/y)$ so that $g(x, y)$ is a homogenous polynomial in x and y of degree N. Differentiate $g(x, y)$ $(m-1)$ times with respect to x and $N - m - 1$ with respect to y to get the polynomial

$$N! \quad \times \quad \frac{S_{N-m-1}}{2\binom{N}{m+1}} x^2 + \frac{S_{N-m}}{\binom{N}{m}} xy + \frac{S_{N-m+1}}{2\binom{N}{m-1}} y^2.$$

Repeated application of Rolle's theorem shows that this quadratic polynomial must have distinct roots (see [8]). Therefore $S_{N-m} = 0$ and $S_{N-m+1} = 0$ cannot hold simultaneously. □

If the grid points are complex, it may be possible to boost the order of accuracy by more than 1. One may obtain formulas for the sequence of determinants with $\beta = 1, \ldots, b$ in Theorem 2. We have already covered the case with $\beta = 1$ in (5.10) and the case with $\beta = 2$ in (5.11). To illustrate the general procedure, we show how to get a formula for the determinant of Theorem 2 with $\beta = 3$. We write down

the identity (5.11) using the grid points $z_1, \ldots, z_N, z_{N+1}$ and replace N by $N+1$.

$$\det(z_1, \ldots, z_N, z_{N+1}; [0, \ldots N-1, N, N+2] - m)$$
$$= \prod_{1 \le i < j \le N} (z_j - z_i) \times \left(S_1^+ S_{N-m+1}^+ - S_{N-m+2}^+ \right) \times \prod_{k=1}^{N} (z_{N+1} - z_k).$$

We use $S_1^+ = S_1 + z_{N+1}$, $S_{N-m+1}^+ = S_{N-m+1} + z_{N+1} S_{N-m}$, and $S_{N-m+2}^+ = S_{N-m+2} + z_{N+1} S_{N-m+1}$, and equate coefficients of z_{N+1}^N to get

$$\det(z_1, \ldots, z_N; [0, \ldots N-1, N+2] - m)$$
$$= \prod_{1 \le i < j \le N} (z_j - z_i) \times \left(S_{N-m+2} - S_{N-m+1} S_1 + S_{N-m} S_1^2 - S_{N-m} S_2 \right).$$

This is the determinant with $\beta = 3$ in Theorem 2. It gets cumbersome to go on like this. However, we notice that the condition for the determinants with $\beta = 1, 2, 3$ to be zero is $S_{N-m} = S_{N-m+1} = S_{N-m+2} = 0$. Here a simple pattern is evident.

To prove this pattern, we assume that the determinant of Theorem 2 with $\beta = r$ is of the form given by Theorem 3 with

$$S = S_{N-m+r-1} + \text{more terms},$$

where each term other than the first has a factor that is one of $S_{N-m}, ..., S_{N-m+r-2}$. We pass to the case $\beta = r+1$ using the grid points $z_1, \ldots, z_N, z_{N+1}$ as illustrated above. Then it is easy to see that the form of S for $\beta = r+1$ is

$$S = S_{N-m+r} + \text{more terms},$$

where each term other than the first has a factor that is one of $S_{N-m}, ..., S_{N-m+r-1}$. If the determinants with $\beta = 1, \ldots, r$ in Theorem 2 are zero, the additional condition that must be satisfied by the grid points for the determinant with $\beta = r+1$ to be zero is $S_{N-m+r} = 0$.

Theorem 6. *The order of accuracy of the finite difference formula* (5.1) *for the m-th derivative is boosted by b if and only if $S_{N-m} = S_{N-m+1} = \cdots = S_{N-m+b-1} = 0$. Even with complex grid points, the order of accuracy can never be boosted by more than m.*

Proof. The first part of the theorem was proved by the calculations that preceded its statement. To prove the second part, suppose that the order of accuracy is boosted by $m+1$. Then we must have $S_N = 0$ which means at least one of the grid points is zero. Since no other grid point can be zero, we must have $S_{N-1} \neq 0$, which is a contradiction. □

Remark 2. Algorithm 2 uses the results of this section to determine the order of accuracy and the leading error term in the case of real grid points. We suspect that the error is exactly equal to $C \frac{f^{(r+m)}(\zeta)}{(r+m)!} h^r$ for some point ζ in an interval that includes 0 and all of the grid points.

Algorithm 2 Order of Accuracy and Error Constant

Input: Grid points $z_1, \ldots, z_N$ all of which are real.
Input: Order of derivative m with $1 \leq m \leq N-1$.
Input: Weights $w_{1,m}, w_{2,m}, \ldots, w_{N,m}$ in the finite difference formula for $f^{(m)}(0)$.
Comment: $w_{k,m}$ are computed using Algorithm 2.
Input: Tolerance τ
$S_{N-m} = \sum_{1 \leq i_1 < \cdots < i_{N-m} \leq N} z_{i_1} \ldots z_{i_{N-m}}$.
$T_{N-m} = \sum_{1 \leq i_1 < \cdots < i_{N-m} \leq N} \left| z_{i_1} \ldots z_{i_{N-m}} \right|$.
if $\left| S_{N-m} \right| < \tau T_{N-m}$ **then**
    $r = N - m + 1$.
else
    $r = N - m$.
end if
$C = \sum_{k=1}^{k=N} w_{k,m} z_k^{r+m}$.
Leading error term of (1.2): $C \frac{f^{(r+m)}(0)}{(r+m)!} h^r$.

Remark 3. Our approach to the proof of Theorem 6 used explicit manipulation of determinants. There is another approach based on the remainder formula for polynomial interpolation. Let $p(z)$ be the unique polynomial interpolant to $f(z)$ at the grid points $z_1, z_2, \ldots, z_N$. Then

$$f(z) - p(z) = f[z_1, \ldots, z_N, z](z - z_1) \ldots (z - z_N),$$

where $f[\ldots]$ is a divided difference; for this formula, see [4]. The error in the finite difference approximation to $f^{(m)}(z)$ is given by

$$\frac{d^m}{dz^m} f[z_1, \ldots, z_N, z](z - z_1) \ldots (z - z_N).$$

By using the product rule, we may expand the error as

$$f[z_1, \ldots, z_N, z] \frac{d^m}{dz^m}(z - z_1) \ldots (z - z_N) + \binom{m}{1} \frac{d}{dz} f[z_1, \ldots, z_N, z] \frac{d^{m-1}}{dz^{m-1}}(z - z_1) \ldots (z - z_N) + \cdots .$$

It is readily seen that $S_{N-m} = 0$ corresponds to the vanishing of the first term at $z = 0$, $S_{N-m+1} = 0$ to that of the second term, and so on. Basic facts about divided differences and their derivatives may be used to complete the proof.

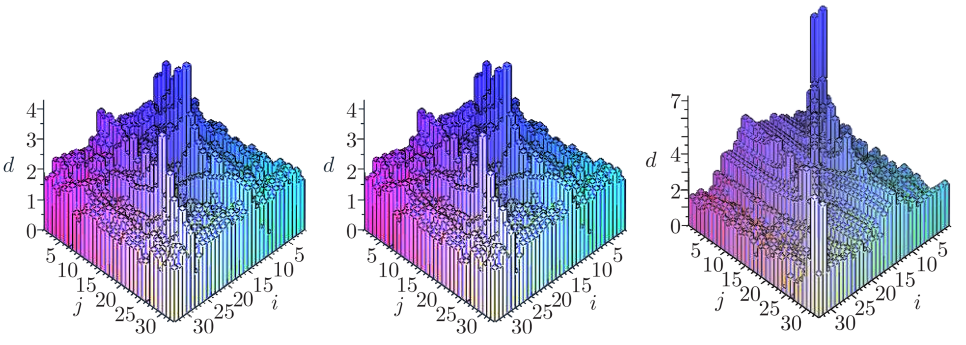
Remark 4. Theorem 6 asserts that the maximum boost in the order of accuracy is m even if complex grid points are allowed. This maximum boost is realized when $z_k = z_1 \exp(2\pi(k-1)i/N)$ for $1 \leq k \leq N$, z_1 being a nonzero complex number. We thank Jeffrey Rauch for this remark.

6. Illustration of numerical stability

For simple choices of grid points, such as $z_k = 0, \pm 1, \pm 2, \pm 3, \pm 4$, all algorithms find the finite difference weights with errors that are very close to machine precision. To compare the different methods, we must turn to more complicated examples.

The Chebyshev points are defined by

$$z_k = \cos\left((k-1)\pi/(N-1)\right) = \sin\left(\pi(N-2k+1)/(N-1)\right)$$



(a) Fornberg (b) Partial products (c) Welfert

FIGURE 6.1. Errors in the entries of the 32×32 Chebyshev differentiation matrix of order $M = 8$ for three different methods. The vertical axis labeled d shows the number of digits of precision lost due to rounding errors.

for $k = 1, \ldots, N$. We will look at the relative errors in the spectral differentiation matrix of order M for $M = 2, 4, 8, 16$ and $N = 32, 64, 128, 256, 512$.

The Chebyshev points are distributed over the interval $[-1, 1]$. The logarithmic capacity of an interval is one quarter of its length, which in this case is $1/2$. Therefore the Lagrange weights w_k will be approximately of the order $1/2^N$. To prevent the possibility of underflow for large N, the Chebyshev points are scaled to $2z_k$ and the resulting finite difference weights for the M-th derivative are multiplied by 2^{-M}.

For reasons described in Section 2, the Chebyshev points are reordered. The reordering we use is bit reversal. With N being a power of 2 in our examples, the binary representation of k (here k is assumed to run from 0 to $N - 1$) can be reversed to map it to a new position. The permutation induced by bit reversal is its own inverse, which simplifies implementation. The reordering of the grid points has the additional effect of making underflows less likely [11]. For a discussion of various orderings of Chebyshev points, see [2]. For convenience, the figures and plots are given with the usual ordering of Chebyshev points.

Before turning to Figure 6.1 , which compares the numerical errors in different methods, we make an important point. Even though the number of digits of precision lost in the 32×32 differentiation matrix of order $M = 8$ may be just 3, the errors in an eighth derivative evaluated using that matrix will be much higher. Some entries of the $N \times N$ Chebyshev differentiation matrix of order M are $\mathcal{O}\left(N^{2M}\right)$. Very large entries occur in the differentiation matrix and in exact arithmetic an accurate derivative will be produced after delicate cancellations during matrix-vector multiplication. In finite precision arithmetic, the largeness of the entries implies that even small rounding errors in the entries of machine epsilon are sufficient to cause explosive errors in numerically computed derivatives.

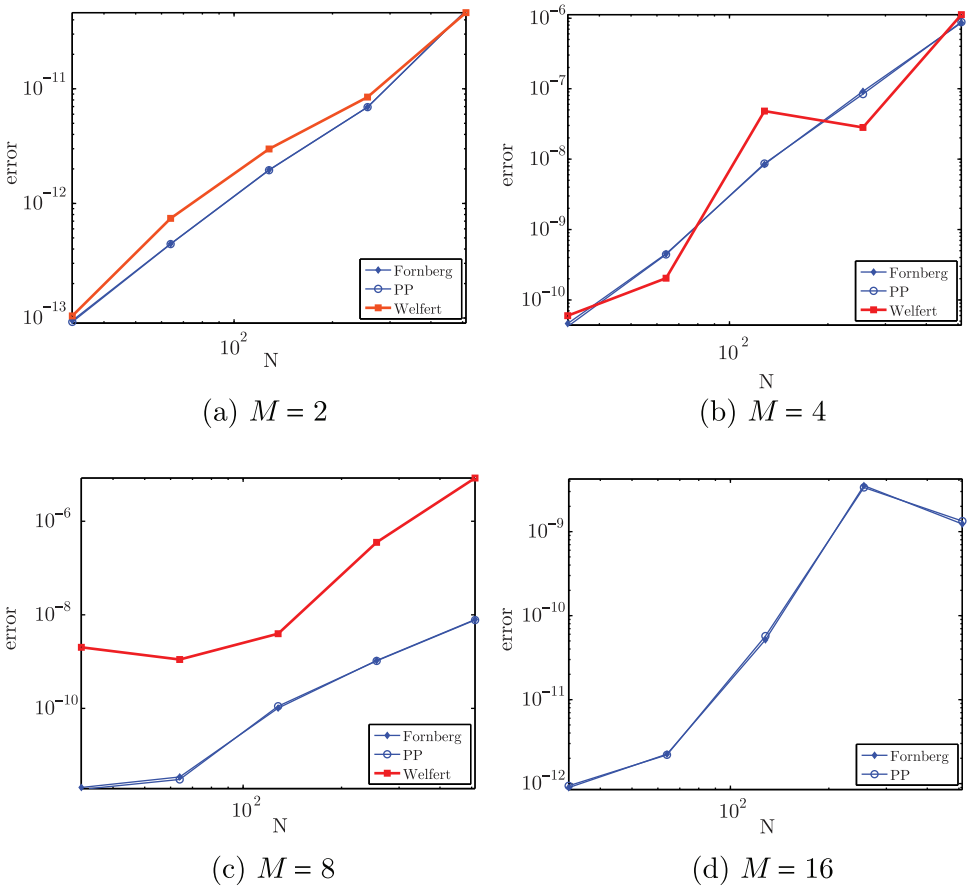


FIGURE 6.2. Variation of the maximum relative error over the N^2 entries of the $N \times N$ Chebyshev differentiation matrix. The order of differentiation is M. While the method of partial products is as accurate as Fornberg's method in spite of using many fewer operations, Welfert's method has an instability which grows exponentially with M. For $M = 16$, the errors for Welfert's method are about 10^6 and therefore omitted from the plot.

From Figure 6.1, we see that Welfert's method loses 7 digits for $N = 32$ and $M = 8$, while the other two methods lose only 3 digits. Fornberg's method and the method of partial products have very similar accuracy. There is a kind of flip symmetry in the errors shown in each of the plots of that figure.

Figure 6.2 gives a more extensive report of errors. All the errors were estimated using 50 digit arithmetic in MAPLE. The errors were then validated using 60 digit arithmetic. From the figure, we see that the algorithm based on partial products is as accurate as Fornberg's method in spite of using many fewer arithmetic operations. From the four plots of Figure 6.2, a surprise is that the errors are smaller for $M = 16$ than for $M = 4$ or $M = 8$. Why is that the case? We are not certain of the answer.

We used the Matlab implementation of Welfert's method which is a part of the Matlab differentiation suite [13]. From Figure 6.2 and its caption, it is evident that instability of Welfert's method grows exponentially with the order of the derivative.

The main finding of this section is that Algorithm 1, which is based on partial products, is as accurate as Fornberg's method even though it uses fewer arithmetic operations.

7. Timed trials

Lemma 1 of Section 3 shows that the method of partial products has a lower operation count than Fornberg's method by a factor of $(5M+5)/4$, where M is the order of the derivative, if the number of grid points N is large. In this section, we compare the two algorithms using timed trials.

Timing scientific programs requires knowledge of computer architecture and an ability to read assembly instructions. On modern platforms, the number of cycles used by a segment of a program depends very greatly on the environment in which the program is run. A typical instruction to fetch a word from memory may take 300 cycles if there are cache misses, if the words are typically in far memory (in cc-NUMA architectures), and dependencies in the instruction sequence eliminate instruction level parallelism. On the other hand, a typical memory fetch instruction may take only 4 cycles if the typical word is in L1 cache and instruction level parallelism may cut that effective latency further by a factor of 10.

When the program is written in a sophisticated language such as C++, the compilers introduce considerable uncertainty in the execution time. Simple changes can change the generated machine code drastically, and for reasons most programmers will not suspect, compilers may generate sub-optimal code. Without reading the assembly code, it is impossible to be certain just what the program is doing. In this section, we briefly sketch how to write a program carefully so as to bring out the efficiencies inherent in the algorithm as well as some of the issues that arise in timing programs.

Table 1 shows that the method of partial products is faster than Fornberg's method, as predicted by the operation counts given in Section 3, except when $N = 4$. Each of the in-cache numbers in the table was obtained by taking the average of either 10^6 or 10^5 successive computations of finite difference weights. The number reported in the table is the median of five such averages. The out-of-cache numbers were obtained by arranging either 10^6 or 10^5 instances of grid points in memory and serially applying the algorithms to each instance and storing the computed weights in a serial fashion in memory. The total cache is less than 13 MB. This method of getting out-of-cache numbers ensures that data is out of cache while letting optimizations, such as prefetching to cache, implemented by the memory controllers take effect as they would in a normal program. The hardware Time Stamp Counter was accessed using the RDTSC instruction to count cycles. This is the most accurate method for counting cycles on modern processors.

TABLE 1. Number of cycles (on a single core of a 2.66 GHZ Xeon 5650 machine) for computing finite difference weights: the method of partial products for computing the weights for the M-th derivative only (PP), the method of partial products for computing the weights for all derivative up to M (PP++), and Fornberg's method (F). The manner in which in-cache and out-of-cache numbers were obtained, as well as the way MATLAB was timed, is described in the text. In nearly all applications, we have $M \leq 4$ as in this table.

		(in cache)			(out of cache)			(MATLAB)			
N	M	PP	PP++	F	PP	PP++	F	PP++	PX++	F1	F2
4	1	368	450	497	405	513	367	1.0e+6	3.5e+5	2.0e+5	3.0e+5
4	2	375	550	642	412	640	471	1.2e+6	3.6e+5	2.2e+5	3.0e+5
16	1	1,692	2,073	6,069	1,843	2,322	3,468	3.9e+6	7.7e+5	2.6e+6	9.4e+5
16	2	1,832	2,478	9,021	1,983	2,818	4,545	4.7e+6	8.5e+5	2.8e+6	9.5e+5
16	4	2,077	3,283	14,694	2,272	3,872	6,629	6.3e+6	8.1e+5	2.8e+6	9.7e+5
64	1	12,836	14,482	91,828	13,844	15,779	40,492	1.5e+7	4.9e+6	3.9e+6	3.8e+6
64	2	13,401	16,118	137,597	14,259	17,809	55,484	1.8e+7	4.3e+6	4.2e+7	3.9e+6
64	4	14,748	20,283	228,937	15,695	22,821	86,783	2.5e+7	4.8e+6	4.2e+7	4.0e+6
256	1	169,647	163,587	1,449,625	163,932	172,315	577,095	6.1e+7	1.5e+8	6.2e+8	2.1e+7
256	2	171,882	183,373	2,173,977	166,310	180,694	906,801	7.4e+7	1.2e+8	6.6e+8	2.3e+7
256	4	177,174	200,120	3,622,952	172,257	200,786	1,414,884	1.0e+8	1.4e+8	6.7e+8	2.5e+7

The MATLAB timing information given in Table 1 was obtained using four scripts, two of them supplied by the referee and two of them written by us. It may be observed that MATLAB is 100 to 500 times slower than C++.

The PP++ column of Table 1 is from a Matlab implementation which closely follows Algorithm 1 . The PX++ column is from another Matlab script. This column does not correspond to the method of partial products. It uses a single for-loop to compute the products $\prod_{j \neq i}(z - z_i)$ up to the z^M term for $i = 1, \ldots, N$. The F1 column is from a Matlab script supplied by the referee that corresponds directly to the Fortran program listed by Fornberg [5]. The F2 column is a highly compact version of Fornberg's method supplied by the referee. Table 1 shows that that the compact implementation of Fornberg's method beats the method of partial products in MATLAB by a factor of five.

Returning to C++, there are a few oddities in Table 1. The method of partial products is faster except when $N = 4$, and in some instances, the programs are faster out of cache than in cache. To understand these oddities, we consider the C++ programs that were used in more detail.

The C++ program used to generate the numbers in Table 1 was written with considerable care. The method of partial products has a modular structure as evidenced by the calls to the `MULTBINOM` and `CONVOLVE` functions in Algorithm 1. These functions correspond to the inner-most loops and the modular structure of the program as a whole allows us to write efficient code. The actual code exploits efficiencies that the presentation in Algorithm 1 does not. The `MULTBINOM` and `CONVOLVE` functions shown give the impression that all partial products are treated as if they have terms up to z^M. The code exploited the fact that l_k for $1 \leq k < M$ and r_k for $N - M < k \leq N$ have fewer terms. The optimized C++ program has separate functions for computing Lagrange weights and finite difference weights (this separation being a crucial feature of the method of partial products) but the functions `MULTBINOM` and `CONVOLVE` are inlined. A desirable feature of the method of partial products as displayed by Algorithm 1 is that there are no conditional statements within loops. We went to considerable trouble to preserve that feature even while allowing for the partial products to have fewer than M terms. Conditional statements inside loops can cause missed branch predictions and the overhead of missed branch predictions for the kind of algorithms that are being timed here can be substantial.

Fornberg's method does not appear to offer similar opportunities for writing efficient code. We used a C++ program that closely follows the Fortran code presented by Fornberg himself [7]. The C++ code we used for Fornberg's method used `restrict` pointers to enable compiler optimizations.

The numbers for the method of partial products given in Table 1 are not the best possible when N is small. We now explain how to modify our code, which we will post on the internet, to make it faster. The modifications are trivial and the speed-up can be as much as a factor of 2.

The C++ function for computing Lagrange weights is listed below.

```
1  void lagrangeWeights(double *restrict z,
2                       int N, double *restrict w){
3    for(int  i=0; i < N; i++){
4      w[i] = 1.0;
5      for(int j=0; j < i; j++)
6        w[i] *= z[i]-z[j];
7      for(int j=i+1; j < N; j++)
8        w[i] *= z[i]-z[j];
9    }
10   for(int i=0; i < N; i++)
11     w[i] = 1.0/w[i];
12 }
```

The inner loops that begin on lines 5 and 7 can be combined into a single loop. However, the combined loop will have a conditional inside to skip `j==i` and that conditional can cause missed branch predictions in the inner-most loop, which is undesirable. The division operations are collected together in the loop that begins on line 10. This reduces the number of division operations and keeps the division operations separate from the multiplications and subtractions. Mixing up division operations with multiplications and subtractions is likely to lead to an instruction stream without as much parallelism. The loops in the listing of `lagrangeWeights()` are written to be easy for the compiler to unroll, which it does as we will see.

Although the `lagrangeWeights()` function has only 12 lines, the assembly generated by Intel's `icpc` compiler (version 12) is several hundred lines. If we are to get a sense of why the program performs as it does, we have no option but to look at the assembly and understand why 12 lines of C++ turn into several hundred lines of assembly. Below is a snippet of assembly code that corresponds to the division loop (lines 9 and 10) of the C++ listing.

```
1
2  ..B2.69:
3          movaps    .L_2il0floatpacket.10(%rip), %xmm0
4          movaps    .L_2il0floatpacket.10(%rip), %xmm1
5          movaps    .L_2il0floatpacket.10(%rip), %xmm2
6          movaps    .L_2il0floatpacket.10(%rip), %xmm3
7          divpd     (%rdx,%rdi,8), %xmm0
8          divpd     16(%rdx,%rdi,8), %xmm1
9          divpd     32(%rdx,%rdi,8), %xmm2
10         divpd     48(%rdx,%rdi,8), %xmm3
11         movaps    %xmm0, (%rdx,%rdi,8)
12         movaps    %xmm1, 16(%rdx,%rdi,8)
13         movaps    %xmm2, 32(%rdx,%rdi,8)
14         movaps    %xmm3, 48(%rdx,%rdi,8)
15         addq      8, %rdi
16         cmpq      %rsi, %rdi
17         jb        ..B2.69
```

Lines 3 through 6 are moving 1 into four XMM registers `%xmm0` through `%xmm3`. Each XMM register is 128 bits and holds two double precision numbers. The `DIVPD` instructions on lines 7 through 10 are evidence that the care we took in writing the C++ program has resulted in good code. In `DIVPD`, the suffix PD stands for packed double and indicates that a single instruction will carry out two divisions. The four `DIVPD` instructions are independent of each other. They can and most likely will be executed in parallel. The move instructions on lines 11 through 14 will move the result of the divisions to the appropriate locations in memory.

Thus the inner loop of the assembly listing has eight divisions and not just one as in the original program. A moment's thought will show the complications the compiler has to go through to generate such code. What if N were only 3 in which case the total number of division operations in `lagrangeWeights()` is also 3? The compiler has to have an answer to that question and the answer is inside complicated assembly code. The compiler has to generate special instructions to check the value of N and skip the assembly snippet we displayed entirely using a jump instruction if N is too small.

That is not the only hoop the compiler has to jump through to generate efficient code. There are many others. One of them has to do with memory alignment. The `DIVPD` instruction requires that all the memory addresses that occur on lines 7 through 14 must be 16 byte aligned. In general, the pointers that the `lagrangeWeights()` function is called with are not 16 byte aligned. The compiler has to generate instructions to check pointer alignment and produce suitably optimized code for each case that arises.

Since our function for computing Lagrange weights was coded carefully in C++, the compiler was able to unroll loops and generate packed double instructions. However, the compiler has to contend with the fact that N, the number of grid points, can take a number of different values. It assumes N to be large enough that loop unrolling results in faster code. If N is small, the unrolling introduces significant overhead. Needless loop unrolling is the reason Fornberg's method is faster for $N = 4$ in Table 1. However, there is an easy way around. We can help the compiler by giving the value of N explicitly as in the listing below.

```
 1  void lagrangeWeights4(double *restrict z,
 2                        double *restrict w){
 3    const int N=4;
 4    for(int  i=0; i < N; i++){
 5      w[i] = 1.0;
 6      for(int j=0; j < i; j++)
 7        w[i] *= z[i]-z[j];
 8      for(int j=i+1; j < N; j++)
 9        w[i] *= z[i]-z[j];
10    }
11    for(int i=0; i < N; i++)
12      w[i] = 1.0/w[i];
13  }
```

The number of grid points N is no longer an argument to the function, but is given as a constant equal to 4 on line 3. Since the compiler knows the value of N, it can

unroll the loops completely and generate straight line code. However, the excellent `icpc` compiler does not unroll the inner loops.

The function for computing finite difference weights can be optimized similarly by giving the values of N and M explicitly as `const int`s.

The optimized program for $N = 4$ and $M = 2$ used 182 cycles, which is less than half the cycles used if values of N and M are not given to the compiler as constants (375 from Table 1). For $N = 32$ and $M = 16$, the improvement is from 7005 cycles to 4246 cycles—still very considerable and an illustration of the great role played by programming skill. The operation count for the method of partial products with $N = 32$ and $M = 16$ is 13314. This corresponds to an impressive 3.14 double precision floating point operations per cycle. The peak bandwidth is 4 operations per cycle since 2 additions or subtractions and 2 multiplications can be completed in every cycle. The fully optimized version reaches nearly 80% of the peak bandwidth. Scientific programs that are not expertly coded would be lucky to reach 10% of the peak bandwidth. Going beyond 80% of peak bandwidth will require hand coding in assembly with a keen awareness of instruction alignment, instruction latencies, register ports, and the port numbers of various instructions. At any rate, even the LINPACK benchmark does not typically get beyond 85% of the peak bandwidth, although the top-most supercomputers appear to beat that figure with very careful tuning.

Table 1 shows that the out-of-cache performance can be better than the in-cache performance. The operation counts for the method of partial products and Fornberg's method are $\mathcal{O}(N^2 + NM^2)$ and $\mathcal{O}(N^2M)$, respectively, as shown in Lemmas 1 and 2. Both methods use $\mathcal{O}(NM)$ space. The number of memory locations accessed is fewer than the number of arithmetic operations with the disparity more pronounced for larger N and M. Therefore neither algorithm is memory limited and we may expect the programs to be as fast out of cache as they are in cache. The out of cache numbers are better in some instances, and dramatically better in the case of Fornberg's method, because there are numerous memory optimizations that take effect when memory is accessed in a regular pattern. When the same memory, spanning no more than a few dozen cache lines, is reused, these optimizations introduce overhead. Exactly what these optimizations are is proprietary information not shared by Intel. Therefore a more complete explanation cannot be given.

Acknowledgments

The authors thank Sergey Fomin, Jeffrey Rauch, Nick Trefethen, and Oleg Zikanov for useful discussions.

References

[1] Jean-Paul Berrut and Lloyd N. Trefethen, *Barycentric Lagrange interpolation*, SIAM Rev. **46** (2004), no. 3, 501–517 (electronic), DOI 10.1137/S0036144502417715. MR2115059 (2005k:65018)

[2] Daniela Calvetti and Lothar Reichel, *On the evaluation of polynomial coefficients*, Numer. Algorithms **33** (2003), no. 1-4, 153–161, DOI 10.1023/A:1025555803588. International Conference on Numerical Algorithms, Vol. I (Marrakesh, 2001). MR2005559 (2004j:65061)

[3] R.M. Corless and S.M. Watt, *Bernstein bases are optimal, but, sometimes, Lagrange bases are better*, In Proceedings of SYNASC, Timisoara, pages 141–153. MIRTON Press, 2004.

[4] Philip J. Davis, *Interpolation and approximation*, Dover Publications Inc., New York, 1975. Republication, with minor corrections, of the 1963 original, with a new preface and bibliography. MR0380189 (52 #1089)

[5] Bengt Fornberg, *Generation of finite difference formulas on arbitrarily spaced grids*, Math. Comp. **51** (1988), no. 184, 699–706, DOI 10.2307/2008770. MR935077 (89b:65055)
[6] Bengt Fornberg, *A practical guide to pseudospectral methods*, Cambridge Monographs on Applied and Computational Mathematics, vol. 1, Cambridge University Press, Cambridge, 1996. MR1386891 (97g:65001)
[7] Bengt Fornberg, *Calculation of weights in finite difference formulas*, SIAM Rev. **40** (1998), no. 3, 685–691 (electronic), DOI 10.1137/S0036144596322507. MR1642772
[8] G. H. Hardy, J. E. Littlewood, and G. Pólya, *Inequalities*, Cambridge Mathematical Library, Cambridge University Press, Cambridge, 1988. Reprint of the 1952 edition. MR944909 (89d:26016)
[9] Nicholas J. Higham, *Accuracy and stability of numerical algorithms*, 2nd ed., Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2002. MR1927606 (2003g:65064)
[10] Nathan Jacobson, *Basic algebra. I*, 2nd ed., W. H. Freeman and Company, New York, 1985. MR780184 (86d:00001)
[11] B. Sadiq and D. Viswanath, *Barycentric Hermite interpolation*, Arxiv preprint, 2011.
[12] D. Viswanath and L. N. Trefethen, *Condition numbers of random triangular matrices*, SIAM J. Matrix Anal. Appl. **19** (1998), no. 2, 564–581 (electronic), DOI 10.1137/S0895479896312869. MR1614019 (99b:65061)
[13] J. A. C. Weideman and S. C. Reddy, *A MATLAB differentiation matrix suite*, ACM Trans. Math. Software **26** (2000), no. 4, 465–519, DOI 10.1145/365723.365727. MR1939962 (2003g:65004)
[14] Bruno D. Welfert, *Generation of pseudospectral differentiation matrices. I*, SIAM J. Numer. Anal. **34** (1997), no. 4, 1640–1657, DOI 10.1137/S0036142993295545. MR1461800 (98e:65105)

Department of Mathematics, University of Michigan, Ann Arbor, Michigan 48109
E-mail address: bsadiq@umich.edu

Department of Mathematics, University of Michigan, Ann Arbor, Michigan 48109
E-mail address: divakar@umich.edu