JOURNAL OF COMPUTATIONAL PHYSICS 98, 174–177 (1992)

NOTE

A Fast Algorithm for Spectral Differentiation

1. INTRODUCTION

Suppose we know the value of a function at several points and we want to approximate its derivative at those points. One way to do this is to find the polynomial that passes through all of the data points, differentiate it analytically, and evaluate this derivative at the grid points. Or, we could generalize this idea by replacing polynomials by linear combinations of some other set of functions. In either case, if $u = (u(x_0), u(x_1), ..., u(x_n))$ is the vector of function values, and $u' = (u'(x_0), u'(x_1), ..., u'(x_n))$ is the vector of approximate derivatives obtained by this idea, then there is a matrix D such that

$$u' = Du. \tag{1}$$

Such a matrix is called a derivative matrix [1].

Approximating differentiation by matrix multiplication is a very common computation, and a somewhat expensive one. A matrix–vector multiplication takes $O(n^2)$ operations. This would take up most of the cpu time if it were used in a numerical scheme to solve a PDE, since most of the other computations in a PDE solution take only $O(n)$ operations. At least, this is true using an explicit time-stepping scheme. With an implicit scheme, there are systems of equations to be solved and it is not clear what would use up the most CPU time. But with either type of scheme, any reduction in the number of operations would be very welcome.

The matrices used for spectral differentiation have various regularities in them. It is reasonable to hope that they can be exploited to reduce the number of operations needed to do the multiplication. For some matrices this is already known to be true, for example the matrices based on trigonometric interpolation at equally spaced points, [1–3], or polynomial interpolation at the extrema of a Chebyshev polynomial [2, 3]. They allow a matrix–vector multiplication in $O(n \log n)$ operations by using fast Fourier or cosine transforms and $O(n)$ operations on the transformed vector. In this paper we present a simple algorithm for doing the matrix multiplication in just over half the number of operations required by the usual algorithm. It requires that the matrix have a kind of symmetry that almost all derivative matrices have.

The required regularity is that either

$$d_{i,j} = d_{n-i,n-j} \quad \text{or} \quad d_{i,j} = -d_{n-i,n-j} \tag{2}$$

for $0 \leqslant i, j \leqslant n$. The first corresponds to the property that if the vector containing function values is reversed, i.e., $(u_0, u_1, ..., u_n)$ is replaced by $(u_n, u_{n-1}, ..., u_0)$ then the derivative of the function is also reversed. The second regularity means that if the vector is reversed, the derivative is reversed and also multiplied by -1. If the gridpoints are arranged symmetrically about the center of the domain of approximation, and the basis functions have enough symmetry with respect to the center of the domain, then the matrix will have one of these properties. For example, the Chebyshev first derivative matrix has the second property, and the Chebyshev second derivative matrix has the first property.

The entries of the Chebyshev first derivative matrix can be found in [2 or 3]. The gridpoints used by the Chebyshev derivative matrix are

$$x_i = \cos\left(\frac{\pi i}{n}\right), \qquad i = 0 \text{ to } N. \tag{3}$$

If we let $c_0 = c_n = 2$ and $c_i = 1$, $i = 1$ to $n-1$, then the entries of the matrix are

$$\begin{aligned} d_{jk} &= \frac{c_j}{c_k}\frac{(-1)^{j+k}}{(x_j - x_k)}, && j \neq k, \\ d_{jj} &= \frac{-x_j}{2(1-x_j^2)}, && j \neq 0, n, \\ d_{0,0} &= -d_{n,n} = \frac{2n^2+1}{6}. \end{aligned} \tag{4}$$

We can see that $c_{n-j} = c_j$. Since $x_{n-j} = -x_j$ we have $x_i - x_j = -(x_{n-i} - x_{n-j})$. Finally, since $(-1)^{n-i+n-j} = (-1)^{i+j}$, it is obvious that the off-diagonal entries have the second property. Since is also clear that the diagonal terms have the second property, the claim is proved.

0021-9991/92 $3.00

More generally, any odd-derivative matrix based on polynomial interpolation at a set of points which is symmetric about the center of the domain will have the second property. And any such even-derivative matrix will have the first property. So the matrix that uses data at the zeroes of a Legendre polynomial will have this property. The entries of the Legendre first and second derivative matrices can be found in [3].

2. DERIVATION OF THE ALGORITHM

The fast differentiation algorithm is based on an odd–even decomposition of the function vector. Suppose we start with the second property, $d_{i,j} = -d_{n-i,n-j}$. Let u be the function vector, and e and o be its even and odd parts. For the moment, assume that the number of gridpoints, $n+1$, is even. The case of $n+1$ odd is slightly more complicated and will be treated later. So

$$u_i = e_i + o_i, \qquad i \in \{0, 1, ..., n\}, \tag{5}$$

where

$$e_i = \tfrac{1}{2}(u_i + u_{n-i}) \tag{6a}$$

$$o_i = \tfrac{1}{2}(u_i - u_{n-i}), \tag{6b}$$

and now

$$\begin{aligned} e_i &= e_{n\ \ i}, \\ o_i &= -o_{n-i}. \end{aligned} \tag{7}$$

By the linearity of the derivative operator, we have

$$u' = Du = D(e+o) = De + Do = e' + o', \tag{8}$$

so we can differentiate e and o separately and then add them together later; e' is given by

$$\begin{aligned} e'_i &= \sum_{j=0}^{n} d_{i,j} e_j = \sum_{j=0}^{(n-1)/2} d_{i,j} e_j + d_{i,n-j} e_{n-j} \\ &= \sum_{j=0}^{(n-1)/2} (d_{i,j} + d_{i,n-j})\, e_j, \end{aligned} \tag{9}$$

since e is even. But e' is odd, that is, $e'_{n-j} = -e'_j$:

$$\begin{aligned} e'_{n-i} &= \sum_{j=0}^{(n-1)/2} d_{n-i,j} e_j + d_{n-i,n-j} e_{n-j} \\ &= \sum_{j=0}^{(n-1)/2} -d_{i,n-j} e_j - d_{i,j} e_j \\ &= -\sum_{j=0}^{(n-1)/2} (d_{i,j} + d_{i,n-j})\, e_j = -e'_i. \end{aligned} \tag{10}$$

So we only have to compute the left half of e'. This takes only $n^2/4$ multiplications and $(n^2/4) - (n/2)$ additions.

Similarly, with o we have

$$\begin{aligned} o'_i &= \sum_{j=0}^{n} d_{i,j} o_j = \sum_{j=0}^{(n-1)/2} d_{i,j} o_j + d_{i,n-j} o_{n-j} \\ &= \sum_{j=0}^{(n-1)/2} (d_{i,j} - d_{i,n-j})\, o_j. \end{aligned} \tag{11}$$

But o' is even, i.e., $o'_{n-j} = o'_j$:

$$\begin{aligned} o'_{n-i} &= \sum_{j=0}^{n} d_{n-i,j} o_j = \sum_{j=0}^{(n-1)/2} d_{n-i,j} o_i + d_{n-i,n-j} o_{n-j} \\ &= \sum_{j=0}^{(n-1)/2} -d_{i,n-j} o_j - d_{ij}(-o_j) \\ &= \sum_{j=0}^{(n-1)/2} (d_{ij} - d_{i,n-j})\, o_j = o'_i. \end{aligned} \tag{12}$$

So we only have to compute the left half of o' as well. This takes the same number of operations as for the computation of e'.

Finally, to reconstruct u' from e' and o' we have

$$u'_i = e'_i + o'_i \tag{13a}$$

and

$$u'_{n-i} = e'_{n-i} + o'_{n-i} = -e'_i + o'_i \tag{13b}$$

for $0 \leqslant i < n/2$.

The odd–even decomposition takes n additions, the odd–even reconstruction takes n additions and the two matrix–vector multiplications take $2((n^2/4)\text{ mults} + (n^2/4) - (n/2)$ adds) for a total of

$$\tfrac{1}{2}n^2 \text{ multiplies} + \tfrac{1}{2}n^2 + n \text{ additions}, \tag{14a}$$

compared to

$$n^2 \text{ multiplies} + (n^2 - n) \text{ additions} \tag{14b}$$

for the normal algorithm.

Many people who have read this paper have commented on the similarity between this algorithm and FFT. Both algorithms use the idea of breaking the problem into two pieces, solving them separately, and gluing them back together cheaply. However, the FFT is able to use the idea on the smaller subproblems, while the even–odd decomposition has to stop at one step. If it could be repeated past the first step, it would be much more interesting than it is now. Unfortunately, I do not know how to do this.

Now we turn to the case where $n+1$ is odd. We have been assuming that the sum $\sum_{j=0}^{n} d_{ij}e_j$ can be broken up into two pieces of equal size. If $n+1$ is odd then this cannot be done. If the sum $\sum_j d_{ij}e_j + d_{i,n-j}e_{n-j}$ goes from 0 to $n/2-1$ then we miss the $n/2$ term. If it goes from 0 to $n/2$ then we count the $n/2$ term twice. This is not hard to fix. We can let the sum go from 0 to $n/2$ and then divide the i, $n/2$-terms (the rightmost column) of the matrix $d_{ij}+d_{i,n-j}$ by 2. In addition, $e_{n/2}$ is nonzero so we must include it in our sum, but $e'_{n/2}$ is zero so there is no need to compute it. So the matrix $d_{ij}+d_{i,n-j}$ has $n/2$ rows and $1+n/2$ columns. For the odd terms, $o_{n/2}=0$, so we will not include it in our sums, and the issue of counting it twice does not occur; $o'_{n/2}$ is nonzero though, and we do need to compute it. So the matrix $d_{ij}-d_{i,n-j}$ has $1+n/2$ rows and $n/2$ columns. The odd–even decomposition and reconstruction steps are unchanged.

The case for $d_{ij}=d_{n-i,n-j}$ is exactly the same as the case $d_{ij}=-d_{n-i,n-j}$, at least for $n+1$ even, but with one change: e' is even and o' is odd, rather than the reverse as before, so the reconstruction is

$$u'_i = e'_i + o'_i \tag{15a}$$

and

$$u'_{n-i} = e'_{n-i} + o'_{N-i} = e'_i - o'_i \tag{15b}$$

for $0 \leqslant i \leqslant n/2$. The derivation of the algorithm is the same as the first case, except for some sign changes.

3. MAPPINGS

Frequently, you do not want to use an unmodified spectral differentiation matrix, because either the domain of approximation is wrong, or the distribution of gridpoints is undesirable, or both. In such cases, a mapping from the new domain to the old one is used. Usually, the even–odd decomposition can still be used.

Suppose $g(x)$ is the mapping, then we want to compute the derivative of $u(g(x))$. By the chain rule, this is

$$u'(g(x))\, g'(x). \tag{16}$$

The discrete version of this is

$$D_m u = G' D u, \tag{17}$$

where D_m is the new derivative matrix associated with the mapping g, G' has $g'(x_i)$ on its main diagonal, and D is the unmapped derivative matrix as before. If g is antisymmetric (g must be monotone; it cannot be symmetric). Then the matrix $G'D$ will still have symmetry,

$$\begin{aligned}(G'D)_{n-i,n-y} &= g'(x_{n-i})\, D_{n-i,n-j} = -g'(x_{n-i})\, D_{i,j} \\ &= -g'(x_i)\, D_{i,j} = -(G'D)_{ij},\end{aligned} \tag{18}$$

since the derivative of an antisymmetric function is symmetric. So an antisymmetric mapping can be implemented with no increase in the number of computations. But even if the mapping is not antisymmetric, multiplication of a vector by G' only takes n operations, so the differentiation would still be almost twice as fast as the normal algorithm.

The case of the second derivative is somewhat more complicated:

$$[u(g(x))]'' = u''(g(x))\, g'^2(x) + u'(g(x))\, g''(x). \tag{19}$$

The discrete approximation to this is

$$D_m^2 u = (G'^2 D^2 + G'' D)\, u, \tag{20}$$

where D_m^2 is the new second derivative matrix associated with the mapping g and D^2 is the unmapped second derivative matrix. If g is an antisymmetric function, then this matrix still has the desired symmetry, g'^2 is a symmetric function, and g'' is antisymmetric, so

$$\begin{aligned}&(G'^2D^2 + G''D)_{n-i,n-j} \\ &\quad = g'^2(x_{n-i})\, D^2_{n-i,n-j} + g''(x_{n-i})\, D_{n-i,n-j} \\ &\quad = g'^2(x_i)\, D^2_{ij} + g''(x_i)\, D_{ij} \\ &\quad = (G'^2D^2 + G''D)_{ij}.\end{aligned} \tag{21}$$

This works if g is antisymmetric, but if g is not antisymmetric, then there does not seem to be any way to make the even–odd decomposition work. At least this is true if you only want the second derivative. If you want to compute both $D_m u$ and $D_m^2 u$, then you can compute Du and D^2u using the even–odd decomposition, and then compute $D_m u$ and $D_m^2 u$ from them using only $O(n)$ operations. The situation with the third derivative is very similar to the second derivative.

4. RESULTS

We have used this algorithm to multiply the Chebyshev derivative matrix by a vector. We measured the execution time for this and compared it to the execution time of the

TABLE I

Execution Time, Sun Sparcstation

n	Normal matrix multiply	Even–odd multiply	Cosine transform
16	1.5 ms	1.3 ms	2.1 ms
32	3.3 ms	1.5 ms	3.0 ms
64	10.6 ms	5.4 ms	4.5 ms
128	37.5 ms	19.9 ms	11.8 ms
256	150.7 ms	75.1 ms	19.7 ms

TABLE II

Execution Time, Cray Y-MP

n	Normal matrix multiply	Even–odd multiply
16	4.4 μs	5.5 μs
32	8.4 μs	9.8 μs
64	23.5 μs	18.5 μs
128	60.5 μs	42.7 μs
256	175.7 μs	112.3 μs

normal matrix multiplication algorithm. The programs were run on a Cray Y-MP, and on a Sun Sparcstation. In addition, on the Sparcstation, the algorithm was compared to an $O(n \log n)$ algorithm that uses cosine transforms. No attempt was made to optimize any of this code. The results are in Tables I and II.

On the Sparcstation, the even–odd multiply is roughly twice as fast as the normal matrix multiply for all values of n. In addition, the even–odd multiply is faster than the cosine transform code for all n less than about 65.

On the Cray Y-MP, the even–odd multiply is slower than the normal multiply for small values of n. I do not know why this is so. One possible explanation is that since the even–odd algorithm, as coded, involved twice as many vector operations on shorter vectors, the vector startup overhead was greater. For the larger values of n, the ratio of execution times approached the expected value of 2.

REFERENCES

1. E. Tadmor, *SIAM Rev.* **29**, 525 (1987).
2. C. Canuto, A. Quarteroni, M. Y. Hussaini, and T. Zang, *Spectral Methods in Fluid Mechanics* (Springer-Verlag, New York, 1988).
3. D. Gottlieb, M. Y. Hussaini, and S. A. Orszag, in *Spectral Methods for Partial Differential Equations*, edited by R. G. Voigt, D. Gottlieb, and M. Y. Hussaini (SIAM, Philadelphia, 1984), p. 1.

Received October 7, 1990; accepted November 9, 1990

ALEX SOLOMONOFF *
Division of Applied Mathematics
Brown University
Providence, Rhode Island 02912

* Supported by NASA Grant NAG-1-703 and Air Force Grant AFOSR 90-0093.