AIAA 99–3309

Efficient Implementations of the Quadrature-Free Discontinuous Galerkin Method

David P. Lockard and Harold L. Atkins
NASA Langley Research Center
Hampton, VA

14th AIAA CFD Conference
June 28–July 1, 1999
Norfolk, VA

EFFICIENT IMPLEMENTATIONS OF THE QUADRATURE-FREE DISCONTINUOUS GALERKIN METHOD

David P. Lockard* and Harold L. Atkins†
NASA Langley Research Center
MS 128 Hampton, VA 23681

Abstract

The efficiency of the quadrature-free form of the discontinuous Galerkin method in two dimensions, and briefly in three dimensions, is examined. Most of the work for constant-coefficient, linear problems involves the volume and edge integrations, and the transformation of information from the volume to the edges. These operations can be viewed as matrix-vector multiplications. Many of the matrices are sparse as a result of symmetry, and blocking and specialized multiplication routines are used to account for the sparsity. By optimizing these operations, a 35% reduction in total CPU time is achieved. For nonlinear problems, the calculation of the flux becomes dominant because of the cost associated with polynomial products and inversion. This component of the work can be reduced by up to 75% when the products are approximated by truncating terms. Because the cost is high for nonlinear problems on general elements, it is suggested that simplified physics and the most efficient element types be used over most of the domain.

Introduction

The discontinuous Galerkin (DG) method is a highly compact formulation that provides a method of obtaining the high accuracy required for computational aeroacoustics on unstructured grids. The ability to use an unstructured grid greatly simplifies the largest obstacle in computing the flow around complex geometries: the generation of the grid. In reference 1, the discontinuous Galerkin method was formulated in a quadrature-free form to reduce the computational effort and storage requirements. Atkins *et al.*[2] reported on an implementation of the method for parallel computers. The present paper discusses further strategies to reduce the required CPU time to perform meaningful calculations. Variable ordering and loop unrolling are discussed first as they have broad implications across all of the computations. The rest of the paper addresses the calculations involved with progressively more complex physics. For constant-coefficient, linear problems, the core computations can be cast into the form of matrix-vector multiplications. Improvements in the efficiency can be obtained by taking advantage of the sparsity in the matrices. When standard polynomials are used as the basis functions, most of the sparsity is a result of symmetry in the computational element. By reorganizing the basis functions, blocked matrices are obtained. However, blocking cannot account for all of the sparsity in the matrices for quadrilateral elements. Furthermore, considerably fewer non-zero entries, on the order of N instead of N^2 for an $N \times N$ matrix, can be obtained by using orthogonal polynomials as the basis set. In order to take advantage of all of the sparsity, a computer code is used to generate C macro functions that are included into the proper routines at compile time. The macro functions completely unroll all of the loops involved in the matrix-vector multiplies and account for all of the sparsity. CPU comparisons are given that demonstrate the relative improvements from blocking, using a different basis set, and using the C macros. For equation sets that involve quadratic terms such as for incompressible flow, the calculation of the flux becomes dominant. In the quadrature-free approach, the flux integrations are performed by storing the integrations of all relevant products of the basis sets and simply multiplying by the appropriate coefficients during the calculation. For nonlinear terms, it is necessary to represent products in terms of a single function in the basis set, and usually to expand the number of precomputed integrals. The work increases dramatically when the basis set is large. Still more work is required when the flux requires divisions such as the full Euler equations. To obtain a flux term such as $\rho u^2 = (\rho u)(\rho u)/\rho$ from ρ and ρu requires products and a division. The division can either be approximated from the infinite Taylor series of the inverse of a polynomial, or a matrix must be inverted for each element.

Numerical Method

The discontinuous Galerkin method is applicable to systems of first-order equations of the form

$$\frac{\partial \mathbf{q}}{\partial t} + \frac{\partial \mathbf{f}_i}{\partial x_i} = \mathbf{s}. \qquad (1)$$

*Senior Member, AIAA. Research Scientist NASA LaRC

†Senior Member, AIAA. Senior Research Scientist NASA LaRC

A summation convention is used for all repeated indices. Here, i ranges from unity to the number of coordinate directions. The domain of interest is divided into non-overlapping elements each of which is defined on some domain Ω with a boundary $\partial\Omega$. For the two-dimensional linearized Euler equations, $\mathbf{q}$, $\mathbf{f}$, and $\mathbf{s}$ are given by

$$\begin{aligned}
\mathbf{q} &= \begin{pmatrix} \rho \\ p \\ u \\ v \end{pmatrix}, \quad \mathbf{f}_1 = \begin{pmatrix} \overline{U} & 0 & \overline{\rho} & 0 \\ 0 & \overline{U} & \gamma\overline{P} & 0 \\ 0 & 1/\overline{\rho} & \overline{U} & 0 \\ 0 & 0 & 0 & \overline{U} \end{pmatrix}\mathbf{q}, \\
\mathbf{f}_2 &= \begin{pmatrix} \overline{V} & 0 & 0 & \overline{\rho} \\ 0 & \overline{V} & 0 & \gamma\overline{P} \\ 0 & 0 & \overline{V} & 0 \\ 0 & 1/\overline{\rho} & 0 & \overline{V} \end{pmatrix}\mathbf{q}, \\
\mathbf{s} &= \left(\begin{array}{cc} 0 & 0 \\ 0 & (1-\gamma)(\overline{U}_x + \overline{V}_y) \\ -(\overline{U}\overline{U}_x + \overline{V}\overline{U}_y)/\overline{\rho} & 1/\overline{\rho}_x \\ -(\overline{U}\overline{V}_x + \overline{V}\overline{V}_y)/\overline{\rho} & 1/\overline{\rho}_y \end{array} \right. \\
&\qquad \left. \begin{array}{cc} 0 & 0 \\ (\gamma-1)\overline{P}_x & (\gamma-1)\overline{P}_y \\ \overline{V}_y & -\overline{U}_y \\ -\overline{V}_x & \overline{U}_x \end{array} \right)\mathbf{q}. \qquad (2)
\end{aligned}$$

$\mathbf{q}$ is the vector of dependent variables. An over-line has been used to denote local temporal-mean quantities, and subscripted values denote differentiation. ρ and p are the density and pressure, and u and v are the x and y directed velocities, respectively. The ratio of specific heats is γ. The equations have been made dimensionless using the ambient speed of sound c_o as the reference velocity.

The discontinuous Galerkin method is obtained by approximating the solution in each element Ω in terms of an appropriate set of basis functions b_m.

$$\mathbf{q} \approx \sum_{m=1}^{N} \boldsymbol{q}_m b_m = \boldsymbol{q}_m b_m$$

where $\{b_m, \quad m = 1, 2, \ldots, N\}$ is a set of basis functions. The coefficients of the approximate solution q_m are the new unknowns, and equations governing these unknowns are obtained by an integral projection of the governing equations onto the basis set. The weak conservation form is obtained by integrating by parts.

$$\begin{aligned}
\int_\Omega b_k \frac{\partial \mathbf{q}}{\partial t} J \, d\Omega &- \int_\Omega \frac{\partial b_k}{\partial \xi_i} \mathbf{J}^{-1}\mathbf{f}_i J \, d\Omega \\
+ \int_{\partial\Omega} b_k (J\mathbf{J}^{-1}\mathbf{f}_i n_i)^R ds &= \int_\Omega b_k \mathbf{s} J \, d\omega \qquad (3)
\end{aligned}$$

for $k = 1, 2, \ldots, N$,

$$\mathbf{J} = \frac{\partial(x,y)}{\partial(\xi,\eta)},$$

and $J = |\mathbf{J}|$. The solution $\mathbf{q}$ is approximated as an expansion in terms of the basis functions; thus, both $\mathbf{q}$ and $\mathbf{f}_i$ are discontinuous at the boundary between neighboring elements. The discontinuity in $\mathbf{q}$ between adjacent elements is treated with an approximate Riemann flux, which is denoted by the superscript R. The Jacobian of the transformation from the global coordinates (x, y) to the local coordinates (ξ, η) of the element is $\mathbf{J}$. The basis set must be complete, but many classes of functions could be used. A common choice is a set of polynomials of the form $\xi^i \eta^j$ that are defined local to the element. The basis set for degree p contains all polynomials of the form $\xi^i \eta^j$ such that the integers $i + j \le p$. A possible basis set in two dimensions for $p = 2$ is $\{1, \xi, \eta, \xi^2, \xi\eta, \eta^2\}$.

To implement the quadrature-free approach, the flux $\mathbf{f}_i$ must also be written as an expansion in terms of basis functions:

$$\mathbf{f}_i(\mathbf{q}) \approx \boldsymbol{f}_{ij}(\mathbf{q}) b_j,$$

and a similar expansion is made for the approximate Riemann flux and the source $\mathbf{s}$. Because the functional behavior of all of the variables is known, the integrations in equation (3) can be performed analytically. To obtain the values of the integrals for a particular set of coefficients requires the multiplication of a matrix times the vector of the coefficients of the flux polynomial. Writing the integrations in this form results in

$$\begin{aligned}
& J M_{km} \frac{\partial \boldsymbol{q}_m}{\partial t} - A_{ikj} J \mathbf{J}^{-1} \boldsymbol{f}_{ij} \\
& + B_{lkn} \left(J\mathbf{J}^{-1}\boldsymbol{f}_{ln}\right)^R = J M_{km} \boldsymbol{s}_m \qquad (4)
\end{aligned}$$

where l is an index that runs over the number of sides of the element, and $\mathbf{f}_i n_i \approx \boldsymbol{f}_{ln}\overline{b}_n$ on the lth edge. The overline in $\overline{b}_n$ denotes an edge basis. The mass matrix M_{km} and the tensor A_{ikj} are given by

$$\begin{aligned}
\mathbf{M} &= M_{km} = \int_\Omega b_m b_k \, d\Omega, \qquad (5) \\
\mathbf{A}_i &= A_{ikj} = \int_\Omega b_j \frac{\partial b_k}{\partial \xi_i} \, d\Omega \qquad (6)
\end{aligned}$$

for $1 \le k, j, m \le N$, and i ranges over the number of dimensions. Each $\mathbf{A}_i$ can be viewed as a matrix in k, j with elements A_{ikj}.

The derivation of the boundary integral terms is complicated only by the fact that the solutions on either side of the element boundary are represented in terms of different coordinate systems. This problem is resolved by expressing the solution on both sides of the element boundary in terms of a common edge-based coordinate system (a simple coordinate transformation). This allows the boundary integral to be expressed in terms of an edge matrix $\mathbf{B}_l$ times a vector that is composed of the coefficients of the approximate Riemann flux expressed in terms of the edge-based coordinate system. The edge matrices are given by

$$\mathbf{B}_l = B_{lkn} = \int_{\partial\Omega} b_k\left(\xi(\overline{\xi}_l), \eta(\overline{\xi}_l)\right) \overline{b}_n \, d\overline{\xi}_l \qquad (7)$$

where $\overline{b}_n$ are the basis functions, and $\overline{\xi}$ is the coordinate on the lth edge. The index n ranges up to N_e. Most elements are constrained to shapes that map into one of a few fixed simple computational elements (such as a unit square or an equilateral triangle in two dimensions) with J and $\mathbf{J}$ constant so that the matrices $\mathbf{M}$, $\mathbf{A}_i$ and $\mathbf{B}_l$ are the same for all elements of a given type. The products $\tilde{\mathbf{A}}_i = \mathbf{M}^{-1}\mathbf{A}_i$ and $\tilde{\mathbf{B}}_l = \mathbf{M}^{-1}\mathbf{B}_l$ can be precomputed and stored at a considerable savings in terms of both computer storage and computational time. This constraint is only to facilitate an efficient implementation and can be relaxed at selected elements if the need arises. A detailed derivation of the matrices $\mathbf{M}$, $\mathbf{A}_i$, and $\mathbf{B}_l$ is given in reference 1. Completely general quadrilaterals cannot be mapped into the same similarity element, so most quadrilateral elements are restricted to parallelepipeds which can be linearly mapped into a square. However, arbitrary triangles can be mapped into the same similarity element. Therefore, triangles are used to handle geometric complexity without incurring a large storage penalty. The triangular and quadrilateral elements may be intermixed freely to obtain very general types of meshes.

The final form of the semi-discrete equation is

$$\begin{aligned} \frac{\partial \boldsymbol{q}_m}{\partial t} &= \frac{1}{J}\Big[M_{km}^{-1}A_{ikj}(J\mathbf{J}^{-1}\boldsymbol{f}_{ij}) \\ &- M_{km}^{-1}B_{lnk}\left(J\mathbf{J}^{-1}\boldsymbol{f}_{ln}\right)^R\Big] + \boldsymbol{s}_m \\ &= \frac{1}{J}\Big[\tilde{A}_{imj}\tilde{\boldsymbol{f}}_{ij} - \tilde{B}_{lmn}\tilde{\boldsymbol{f}}_{ln}^R\Big] + \boldsymbol{s}_m \qquad (8) \end{aligned}$$

The groupings $\tilde{\boldsymbol{f}}_{ij} = J\mathbf{J}^{-1}\boldsymbol{f}_{ij}$ and $\tilde{\boldsymbol{f}}_{ln}^R = (J\mathbf{J}^{-1}\boldsymbol{f}_{ln})^R$ have been used because that is what is normally stored. Because all elements of a given type are mapped into the same similarity element, the coefficients $\tilde{A}_{imj}$ and $\tilde{B}_{lmn}$ can be precomputed once and applied throughout the calculation. Equation (8) is advanced in time by using the three-stage Runge-Kutta method of Shu and Osher.[3] Analysis of the stability of this approach can be found in reference 1.

Optimization Procedures

An object-oriented C++ computer program that implements this method has been developed and validated. The program has been ported to parallel computing platforms using MPI calls. The initial port required only minimal changes to the code and was performed in only a few weeks.[2] The speed of the port is attributed to the modular nature of the code and the C++ language. Because most parallel architectures involve cache-based processors rather than vector processors, the loop and variable orderings are altered to minimize cache misses. In the original code, the index identifying the element varied the fastest. This is preferable for vector machines because it is the longest length, and operations across the elements can be pipelined. However, for cache-based machines, how many times the data has to be fetched is more important. In the discontinuous Galerkin method, most of the work on each element is done independently. Therefore, the variables should be ordered to keep all of the information for a given element together. In this way, all of the relevant information for a given element is pulled into the cache simultaneously, worked on, and then replaced with the data for the next element. For example, the flow variables are all stored in the same array. Each flow variable is represented by a sum over a set of basis functions. The data for an element is kept local by having the index for the coefficient of the basis function varying the fastest, then the index for the flow variable, and finally the index for the element. All of the arrays in the code are now ordered in this fashion.

Simply reordering the arrays results in a significant but not overwhelming reduction in run time. However, the new orderings make it easier to unroll loops in a beneficial fashion. The biggest gains are obtained by completely unrolling small loops over the number of variables and number of edges. Even unrolling over the terms in the basis set is beneficial. This will be addressed later in the paper. Reordering the arrays and performing some unrolling results in an overall performance improvement from 30 to 70 megaflops. Some individual subroutines, such as the flux integrations, obtain floating point operation (flop) rates over 150 megaflops.

The remainder of the paper will address specific implementation details and coding procedures that affect performance. First, the dominant routines for constant-coefficient, linear equations are addressed. These include the calculation of the volume and edge integrals, and the transformation of information from the volume to edge coordinates. Next, the calculation of polynomial products is examined. Products must be calculated in the flux routines, and are the most expensive computations for incompressible flow and the linearized Euler equations with a nonuniform mean flow. Finally, the expense of finding the inverse of a polynomial is demonstrated. Polynomial inversion is required for the full Euler equations.

Volume and Edge Integrals/Transformations

For constant-coefficient, linear equations, the most expensive computations in the quadrature-free DG approach involve the volume and edge integrals, and the transformation of information from the volume to edge coordinates. This is not only true on a flop count basis, but in terms of actual CPU time as well. The code includes an extensive set of timing routines which are used to identify computationally intensive routines. In the quadrature-free approach, the integrations of the volume and edge fluxes are performed in two steps. The integrals of the basis functions are calculated once at the beginning of the calculation and stored in a matrix form. In general, it is less expensive to compute the desired integrals by multiplying this matrix by the coefficients of the basis functions than by Gaussian quadrature.[1] However, the matrices are often quite sparse, and considerably less work is required than a full matrix-vector multiply. The code was further optimized by exploiting the sparsity of the matrices used to compute the volume and edge terms. The integrations involve the multiplication of $\tilde{A}_{imj}\tilde{\boldsymbol{f}}_{ij}$ for the volume flux contributions and $\tilde{B}_{lmn}\tilde{\boldsymbol{f}}^{R}_{ln}$ for the edge fluxes. For two-dimensional equations, i takes on values of one and two, and j and k range from unity to the number of basis functions, N. For a basis set that includes polynomials up to degree p,

$$N = (p+1)(p+2)/2 \quad \text{and} \quad N_e = p+1.$$

In many implementations, N is no more than twenty-one, corresponding to $p = 5$ or sixth order for smooth flows. However, the choice is completely arbitrary. The volume term in equation (8) includes two matrix-vector multiplications for each variable in the vector of unknowns. The square matrices are $N \times N$. In the edge term, n ranges over the number of edge basis functions which is N_e. Hence, $\tilde{\mathbf{B}}_l$ is an $N \times N_e$ matrix. For a triangle, three such multiplications must be performed; whereas, four are required for a quadrilateral.

Another computationally intensive operation in the DG method is the transformation of the solution variables in the volume coordinates to each edge coordinate. This can be expressed as

$$\overline{\mathbf{q}}_l = \mathbf{T}_l\mathbf{q} \quad \text{or} \quad \overline{\boldsymbol{q}}_{ln} = T_{lnj}\boldsymbol{q}_j. \tag{9}$$

Once again, this entails the multiplication of $\mathbf{T}_l$, an $N_e \times N$ matrix, times a vector for each of the unknowns on every side of the element. Usually, the edge flux is computed from $\overline{\mathbf{q}}_l$, but in some instances it is actually less expensive to transform the volume flux to the edges in a similar fashion to the solution variables. In this case, the operations in equation (9) would be repeated for the flux. With constant-coefficient, linear equations, the computation of the edge flux is an order N_e operation, so it is less expensive to compute the flux directly.

Blocking

For basis functions of the form $\xi^i\eta^j$, most of the zeroes in the matrices used to compute the integrals are a result of symmetry in the similarity element to which each arbitrary element is mapped. Figure 1 shows a reference square and equilateral triangle. In the current work the origins of the local coordinates are at the geometric centers of the elements. The triangle possesses symmetry in the ξ direction, so the integration of any function even in ξ is zero. For the triangle, this accounts for approximately half the terms in the $\tilde{\mathbf{A}}_i$ matrices. The $\tilde{\mathbf{B}}_l$ and $\mathbf{T}_l$ matrices are generally full for the triangle except for the one that is at a constant value of η which is denoted as the first edge. Table 1 summarizes the number of nonzero entries in each of the important matrices as a function of the maximum degree of the polynomials in the basis set.

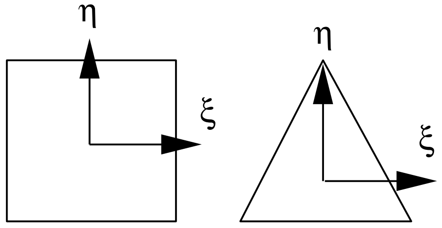

Fig. 1 Similarity elements

p	N	$\tilde{\mathbf{A}}_1$	$\tilde{\mathbf{A}}_2$	$\tilde{\mathbf{B}}_1$	$\tilde{\mathbf{B}}_{2,3}$	$\mathbf{T}_1$	$\mathbf{T}_{2,3}$
1	3	1	1	3	5	3	5
2	6	15	19	9	14	6	14
3	10	48	52	20	38	10	30
4	15	107	117	39	73	15	53
5	21	216	225	63	125	21	89

Table 1 Number of nonzero entries in the matrices used for volume and edge calculations on a triangle when $\xi^i\eta^j$ type basis functions are used.

The ordering of the basis functions is arbitrary and can be chosen to group the nonzero entries in the matrices into blocks. The complexity in the pattern of zeroes comes from the differentiation in the volume term and the combination of the edge and volume bases in the edge term. For the triangle blocking is obtained by grouping together all of the basis functions with even powers of ξ. As an example for $p = 2$, the basis set would be ordered $\{1, \eta, \xi^2, \eta^2, \xi, \xi\eta\}$. This produces blocked matrices of the form

$$\mathbf{M},\ \mathbf{A}_2 = \begin{pmatrix} x & x & x & x & 0 & 0 \\ x & x & x & x & 0 & 0 \\ x & x & x & x & 0 & 0 \\ x & x & x & x & 0 & 0 \\ 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & x & x \end{pmatrix},$$

$$\mathbf{A}_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & x & x \\ x & x & x & x & 0 & 0 \\ x & x & x & x & 0 & 0 \end{pmatrix} \tag{10}$$

An x is used to denote a non-zero entry. Because all of the matrices are blocked similarly, the inversion and multiplication by $\mathbf{M}$ produces $\tilde{\mathbf{A}}_i$ matrices that are blocked in the identical fashion to that shown in equation (10). Hence, the total number of operations that must be performed is reduced by 50% for the volume term. Table 2 summarizes the savings realized for a problem with 2048 triangles. All cases are run on a SGI Octane with an R10000 CPU running at 225 mHz, and the CPU time is measured in seconds. The % Tot time is the ratio of the CPU time spent in

Order		CPU Time	% Tot Time
4	Unblocked	21.9	23.6
	Blocked	18.4	20.7
2	Unblocked	6.39	15.2
	Blocked	6.07	14.7

Table 2 CPU comparison for blocked and unblocked versions of the volume flux calculation with 2048 triangles and 202 steps.

the volume flux calculation to the CPU time for the entire calculation. This gives an indication of the relative importance of making further improvements to the computational rate of the particular routine. It is not meant to be compared between the optimized and unoptimized routines because the total times are different between the two calculations. The modest improvement from blocking is partially caused by the cached based nature of the workstation. Although fewer floating point operations must be performed, the same amount of data must be loaded. Furthermore, the routines under investigation are heavily unrolled. A factor of two change in the CPU time has already been obtained through unrolling. In addition, one must use two loops to perform the multiplications of the two blocks independently; whereas, only one is required when blocking is not used. Table 3 shows that when the problem size is smaller, a greater improvement is obtained for the $p = 4$ case because all of the data fits in the high speed cache.

Order		CPU Time	% Tot Time
4	Unblocked	12.6	27.6
	Blocked	9.6	22.7
2	Unblocked	3.0	15.3
	Blocked	2.7	14.1

Table 3 CPU comparison for blocked and unblocked versions of the volume term calculation with 128 triangles and 2515 steps.

The edge matrices are relatively full, but the edge matrix for the η=constant line and certain orders can be blocked by reordering the edge basis functions. For the $p = 2$ example this is $\{1, \xi^2, \xi\}$ which produces

$$\mathbf{B}_1 = \begin{pmatrix} x & x & 0 \\ x & x & 0 \\ x & x & 0 \\ x & x & 0 \\ 0 & 0 & x \\ 0 & 0 & x \end{pmatrix}, \quad \mathbf{T}_1 = \begin{pmatrix} x & 0 & 0 \\ x & 0 & 0 \\ 0 & x & 0 \\ x & 0 & 0 \\ 0 & 0 & x \\ 0 & 0 & x \end{pmatrix}. \qquad (11)$$

A penalty is associated with this type of blocking because all of the $\mathbf{T}$ matrices did have an upper triangular region of zeroes that is lost. Also, to take advantage of all of the sparsity in the $\mathbf{T}_1$ matrix is difficult. Furthermore, complete blocking can not be obtained above $p = 3$. The improvement in performance obtained by taking advantage of this type of blocking did not produce any significant results.

Although squares possess symmetry in both the ξ and η directions, the $\tilde{\mathbf{A}}_i$ have a more complex pattern and cannot be blocked any differently from what was done for the triangle. Table 4 gives the the number of nonzero entries in the $\tilde{\mathbf{A}}_i$, $\tilde{\mathbf{B}}_l$, and $\mathbf{T}_l$ matrices as a function of degree. Because the square is symmetric in each direction, the matrices for the different edges and directions have the same number of zeroes. Since simple blocking cannot account for all of the sparsity, a different strategy must be used for the square.

p	N	$\tilde{\mathbf{A}}_i$	$\tilde{\mathbf{B}}_l$	$\mathbf{T}_l$
1	3	1	3	3
2	6	6	9	6
3	10	18	20	10
4	15	39	39	15
5	21	75	63	21

Table 4 Number of nonzero entries in the matrices used for volume and edge calculations on a square when $\xi^i\eta^j$ type basis functions are used.

Specialized Matrix-Vector Multiplication Routines

The most straightforward and effective means to account for all of the sparsity in the matrices is to write specialized matrix-vector multiplication routines that only perform operations involving the nonzero entries. A byproduct of this is that all of the loops that are normally used to perform the multiplication are now completely unrolled. The only real complication with this approach is that a different routine must be used for each element type and each degree. To simplify the task of writing all of the needed routines, a computer code was written that produces C macros that perform the multiplications in an optimal fashion. These macros are pulled into the appropriate section of code at compile time, and C++ allows the different multiplication routines to be associated with elements of the proper degree. Tables 5 and 6 demonstrates the savings in CPU time obtained by using the specialized multiplication routines. The savings are significant for both the large and small problems sizes as a savings of a factor of two is obtained in both cases.

	Specialized		Loops	
	CPU	% Tot	CPU	% Tot
Volume Flux	7.0	16.4	13.0	20.0
Edge Flux	5.2	12.1	12.5	19.2
Transformation	3.8	8.8	13.9	21.2

Table 5 CPU time comparison for specialized routines with unrolled loops accounting for sparsity to routines using loops. Calculations used 512 $p = 4$ squares and 503 steps.

	Specialized		Loops	
	CPU	% Tot	CPU	% Tot
Volume Flux	11.8	15.1	21.9	20.3
Edge Flux	11.9	15.1	23.8	22.1
Transformation	7.9	9.9	18.4	17.1

Table 6 CPU time comparison for specialized routines with unrolled loops accounting for sparsity to routines using loops. Calculations used 2048 $p = 4$ squares and 202 steps.

Although the matrices for triangles don't have as many zeroes as squares, the same complete unrolling technique can be applied for them. C macros were also generated for this case. Tables 7 and 8 compare the CPU times in seconds for the specialized matrix-vector multiplication routines with completely unrolled loops to the original code. The CPU time for the volume flux calculation is further reduced from the time required for the blocked version as shown in table 2. Therefore, some additional benefit is obtained from completely unrolling the loops. Furthermore, the edge calculations also benefit from the specialized routines. The benefit is also obtained for other values of p. For $p = 2$, the specialized routines cost about 2/3 that of the original versions. The CPU time requirements for the square are appreciably less for the volume computation, but commensurate for the edges. The additional edge for the square offsets the gains from the sparsity in comparison with the triangle.

	Specialized		Loops	
	CPU	% Tot	CPU	% Tot
Volume Flux	8.8	22.2	13.0	26.3
Edge Flux	5.0	12.6	7.9	16.0
Transformation	3.8	9.6	6.6	13.4

Table 7 CPU time comparison for specialized routines with unrolled loops accounting for sparsity to routines using loops. Calculations used 512 $p = 4$ triangles and 503 steps.

	Specialized		Loops	
	CPU	% Tot	CPU	% Tot
Volume Flux	16.1	20.0	21.9	23.6
Edge Flux	13.2	16.5	17.0	18.3
Transformation	8.2	10.2	13.1	14.1

Table 8 CPU time comparison for specialized routines with unrolled loops accounting for sparsity to routines using loops. Calculations used 2048 $p = 4$ triangles and 202 steps.

p	N	$\tilde{\mathbf{A}}_i$	$\tilde{\mathbf{B}}_l$	$\mathbf{T}_l$
1	3	1	1	3
2	6	3	6	6
3	10	7	10	10
4	15	13	15	15
5	21	22	21	21

Table 9 Number of nonzero entries in the matrices used for volume and edge calculations on a square when Legendre polynomials are used as the basis functions.

Basis Functions

Although the matrices are already sparse for the square, one can employ a different set of basis functions with special properties to create much greater sparsity. For a square computational element, the orthogonal Legendre Polynomials are a good choice because they have a weight function of unity, and a tensor product can be used to represent a multidimensional basis set. Table 9 summarizes the number of nonzero entries in the important matrices when Legendre Polynomials are used as the basis set. Let $P_n(x)$ and $P_m(y)$ represent the Legendre polynomials of orders n in x and m in y, respectively. For the basis set $\{P_0(x), P_1(x), P_1(y), P_2(x), P_1(x)P_1(y), P_2(y), ...\}$, the mass matrix becomes

$$\mathbf{M} = M_{j,k} = \int_\Omega b_j b_k \, d\Omega \; = \left\{ \begin{array}{l} \frac{2}{2m+1}\frac{2}{2n+1} \text{ for } m = n \\ 0 \text{ for } m \neq n \end{array} \right\} \tag{12}$$

where m and n are the orders of the Legendre polynomials in b_j and b_k, respectively. This mass matrix is diagonal. The $\tilde{\mathbf{A}}_i$ matrices can also be calculated relatively simply using the recurrence relation[4]

$$\frac{dP_n}{dx} = \frac{P_{n-2}}{dx} + (2n-1)P_{n-1} \tag{13}$$

and orthogonality. The important integral is

$$\int_\Omega P_j(\xi)\frac{\partial P_k(\xi)}{\partial \xi_i} \, d\Omega = \left\{ \begin{array}{l} 2 \text{ for } k > j \text{ and } j+k \text{ odd} \\ 0 \text{ otherwise} \end{array} \right\} \tag{14}$$

The resulting volume matrices have the form

$$\tilde{\mathbf{A}}_1,\ \tilde{\mathbf{A}}_2 \ = \left(\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ x & 0 & 0 & 0 & 0 & 0 \\ y & 0 & 0 & 0 & 0 & 0 \\ 0 & x & 0 & 0 & 0 & 0 \\ 0 & y & x & 0 & 0 & 0 \\ 0 & 0 & y & 0 & 0 & 0 \end{array}\right) \qquad (15)$$

for $p = 2$. The x's are nonzero for $\tilde{\mathbf{A}}_1$, and the y's for $\tilde{\mathbf{A}}_2$. The fill in these matrices is on the order of N for $p = 5$ or less. The sparsity decreases slightly as the order increases. For $p = 5$, twenty-two nonzero entries exist in each of the 21×21 $\tilde{\mathbf{A}}_i$ matrices. Examination of the matrices in (15) reveals that the last three columns are all zero. These columns would multiply the coefficients of the highest order terms in the polynomial expansion. Hence, the volume flux is only needed to degree $p-1$ instead of p when orthogonal polynomials are used. The reason that the highest order terms are not needed is because the derivative of the basis function in equation (6) can be represented completely by terms of lower order. For example, the derivative of a Legendre Polynomial is given by[4]

$$\frac{dP_n(x)}{dx} = \sum_{k=1}^{(n+1)/2} (2n+3-4k)P_{n-2k+1}. \qquad (16)$$

The lower order terms representing the derivative will always be orthogonal to highest order terms that are being used.

The fill for the $\tilde{\mathbf{B}}_l$ matrices is N when Legendre polynomials are also used as the edge basis functions. The transformation matrix from the volume to the edge coordinates is very sparse for the square because each edge of the square is either an $\xi =$ constant or an $\eta =$ constant line. Hence, the transformation matrix really just simulates evaluating the volume polynomial along a constant line. The $\mathbf{T}_1$ matrix has the form

$$\mathbf{T}_1 = \left(\begin{array}{ccc} x & 0 & 0 \\ 0 & x & 0 \\ x & 0 & 0 \\ 0 & 0 & x \\ 0 & x & 0 \\ x & 0 & 0 \end{array}\right). \qquad (17)$$

All of the $\mathbf{T}$ matrices have N non-zero values for both the standard and Legendre basis sets. Tables 10 and 11 compare the CPU time in seconds for computations using the Legendre and standard, nonorthogonal polynomials for basis functions. Macros are employed for the calculations with both the nonorthogonal and orthogonal polynomials. A substantial decrease in the CPU time is obtained when using the Legendre polynomials. The % Tot column gives the percentage of the total CPU time spent in that particular routine.

	Legendre		Nonorthogonal	
	CPU	% Tot	CPU	% Tot
Volume Flux	5.0	12.9	7.0	16.4
Edge Flux	3.1	8.0	5.2	12.1
Transformation	3.8	9.9	3.8	8.8

Table 10 CPU time comparison for different basis functions for calculation with 512 $p = 4$ squares and 503 steps.

	Legendre		Nonorthogonal	
	CPU	% Tot	CPU	% Tot
Volume Flux	8.5	11.8	11.8	15.1
Edge Flux	8.6	12.0	11.9	15.1
Transformation	7.9	11.3	7.9	9.9

Table 11 CPU time comparison for different basis functions for calculation with 2048 $p = 4$ squares and 202 steps.

Although Legendre polynomials are not orthogonal on the triangle, they can still be used as the basis functions. The work and sparsity of the matrices is nearly identical to that of the standard polynomial set, and therefore the CPU times in tables 7 and 8 are also relevant for Legendre Polynomials with triangles. The advantage of using Legendre polynomials for the triangle is that they could be easily intermixed with squares using the same basis set. Because the parallelepiped is the most general shape that can be mapped into a square, quadrilateral elements are not very useful for handling geometric complexity. However, one would prefer to use them over most of the domain because they are more efficient. When a common basis set is used, it would be easy to apply the triangles to handle geometric complexity, and the quadrilaterals to fill up the majority of the domain. Even if different basis sets are used, triangles could still be easily intermixed with squares if they share the same edge bases at their interfaces.

Orthogonal polynomials on the triangle can be generated using Gram-Schmidt orthogonalization in order to obtain sparse volume matrices for the triangle. The fill for the volume matrices when using orthogonal polynomials is only slightly higher than that obtained with Legendre Polynomials on the square, and the run times for the volume flux calculations are similar to those in tables 10 and 11. However, these polynomials are not of the form of a tensor product, so one cannot use a subset of the terms from the volume representation as the edge basis. Hence, the edge matrices are still full on the second and third edges. One could use area coordinates for the triangle so that each edge would lie on a line that is a zero value of one of the coordinates, but this would require dealing with a third coordinate direction in two dimensions. A similar effect could by obtained by using a right triangle as the similarity element so that two edges lie on constant lines of the coordinates, but this would lose the symmetry properties of the isosceles triangle.

An alternative orthogonal basis for the triangle is the warped product proposed by Dubiner[5] and utilized by Sherwin and Karniadakas.[6] Dubiner's basis employs a singular mapping of the triangle into a square. A set of Legendre and Jacobi polynomial products with a weighting term is used as the basis set which is orthogonal on the triangle and separable by coordinate direction on the square. A typical term in the set resembles $P_m(\xi)(1-\eta)^m P_n^{2m+1,0}(\eta)$ where $P_n^{\alpha,\beta}(\eta)$ is the nth order Jacobi polynomial.[7] Table 12 summarizes the number of nonzero entries in the volume matrix for Gram-Schmidt and Dubiner's warped basis sets. The sparsity for the Gram-Schmidt basis is better, but the high-order terms are quite complicated. For constant-coefficient, linear problems, this is irrelevant, but it is important for nonlinear problems when products of terms must be calculated. Dubiner's basis also loses the property of not needing the degree p terms in the volume flux.

		Gram-Schmidt		Warped	
p	N	$\tilde{\mathbf{A}}_1$	$\tilde{\mathbf{A}}_2$	$\tilde{\mathbf{A}}_1$	$\tilde{\mathbf{A}}_2$
1	3	1	1	2	2
2	6	3	4	7	8
3	10	7	10	20	20
4	15	13	20	43	40
5	21	35	22	84	70

Table 12 Number of nonzero entries in the matrices used for volume calculations on a triangle when Gram-Schmidt generated orthogonal polynomials and Dubiner's warped basis are used as the basis functions.

Polynomial Products

Although some significant performance improvement can be obtained for the volume and edge integrations and the edge transformation, the time spent in these routines is often overwhelmed by flux calculations when polynomial products must be calculated. These products occur for the linearized Euler equations with a nonuniform mean flow as given in equation (2). The reason for the particular form of equation (2) is to make them conform to (1) while still solving for the primitive variables that are needed for the product terms. Twenty-five products must be calculated on each element to solve equation (2). The solution variables for the fully conservative form of the linearized Euler equations involve terms that are products and sums of the mean and perturbed variables. For example, the conserved variable for the x momentum equation is $\overline{\rho}u + \rho\overline{U}$. It would be extremely expensive to extract the perturbed variables from such a system because each term is a polynomial. It may be possible to rearrange the formulation to avoid the source terms and the derivatives of the mean flow, but this has not been explored in detail.

The most general procedure for calculating a product is projection. For example, one needs to find $f = \overline{\rho}u$. Writing the expansion in terms of the basis set explicitly gives $f_m b_m = \overline{\rho}_i b_i u_j b_j$. Multiplying both sides by b_k and integrating yields

$$M_{km} f_m = \int_\Omega b_i b_j b_k \; d\Omega \;\; \overline{\rho}_i u_j . \tag{18}$$

The mass matrix appears on the left hand side. For orthogonal sets, the mass matrix is diagonal, and there is simply a scaling term on each f_m. Hence, one would be left with a typical formula for finding the coefficients of an orthogonal representation of a function. The difference here is that the function to be fit has a special form because it arose from the product of two series. In general the mass matrix is inverted, and the equation can be written as

$$f_m = \overline{\rho}_i D_{mij} u_j \quad \text{where}$$

$$\mathbf{D}_m = D_{mij} = M_{km}^{-1} \int_\Omega b_i b_j b_k \; d\Omega . \tag{19}$$

Hence, each f_m term involves a double sum over the matrix $\mathbf{D}_m$, which can be precomputed and stored. Ignoring sparsity, calculating f_m is an $O(N^3)$ operation which would clearly dwarf all of the operations discussed earlier. Fortunately, there is some sparsity, but the operation count can still be quite high depending on the details of how the calculations are performed.

Adams[8] investigated the products of Legendre Polynomials and was able to derive an explicit expression for the product in terms of a single series. The expression has the form

$$P_m(x) P_n(x) = \sum_{k=0}^{\min(m,n)} c_k P_{m+n-2k}(x) \tag{20}$$

where the c_k are constants. This sum includes terms of higher order than either P_m or P_n. If they are higher order than any of the terms in the basis set in use, they can be ignored because the orthogonality property assures us that excluding them will not have any effect on the lower order terms. Equation (20) can be used to determine the f_m directly, or equation (19) can be used. Table 13 presents the operation counts to resolve a product into a single series when using a basis set of Legendre polynomials. All floating point operations are given equal weight of unity. The counts are approximate values reported by a symbolic manipulation package after the application of some optimization procedures to account for duplicate computations. Because the volume flux from the order p terms does not contribute for the Legendre basis, the table gives the operation counts for an expansion of the product to order $p-1$ and p. All of the terms to order p are required when the volume flux is transformed to the edges rather than calculating the edge fluxes directly. Transforming the fluxes tends to be less expensive for the linearized Euler equations, and precludes the need to store representations of the mean flow on the edges. Although the operation counts in table 13 are scaling closer to $O(N^2)$ than $O(N^3)$, the counts still grow rapidly with p. A mitigating factor is that all of the work is done on a small set of data that easily fits in cache. Hence, high flop rates are obtained for these operations.

Because truncation of the product formula in (20) is only strictly valid for square domains, orthogonal bases on triangular domains are also examined. Table 13 lists the operation counts when Gram-Schmidt generated orthogonal polynomials on the triangle are used as the basis set. These counts are considerably greater. Dubiner's warped product basis actually has smaller operation counts for the triangle, but they are still too large to be practical at high order. In actuality, even the work for the Legendre basis would limit most applications to $p = 4$ or less.

		Degree of Expansion			
		Square (Legendre)		Triangle (Gram-Schmidt)	
p	N	$p-1$	p	$p-1$	p
1	3	7	13	10	31
2	6	39	64	96	194
3	10	162	244	477	902
4	15	450	634	1695	2525
5	21	1114	1501	5081	7204

Table 13 Operation counts to perform products when orthogonal polynomials are used as the basis set. Counts for expansions including terms up to degree $p-1$ and p are given.

When using standard polynomials of the form $\xi^i\eta^j$ as the basis set, the calculation of products using equation (19) can be viewed as a two-step process. The first step is the multiplication of the polynomials into an expansion of degree p. The product of any two terms in the basis set produces another term in the basis set, so the multiplication step is relatively straightforward. For example $c_1(\xi^2\eta^3)c_2(\xi^3\eta) = c_1c_2\ \xi^5\eta^4$. Hence, the multiplication of two polynomials of degree p produces a polynomial of degree $2p$ at a cost of $O(2N^2)$. Because these polynomials are not orthogonal, terms of degree up to p don't provide the best approximation of the product when the terms of degree higher than p are truncated. Hence, the second part of the process in equation (19) is to use the terms up to degree p to represent those from degree $p+1$ to $2p$. The operation counts for the overall process are given in table 14. The work required for the standard polynomials is commensurate with that for the orthogonal polynomial bases. Furthermore, the expansion to p is not ideal. Equation (19) guarantees that the integration of the projected volume flux against the basis set will produce an exact result, so the required integral in equation (3) will also be exact because the derivative of a basis function is another basis function. However, errors are incurred when a projected volume flux is transformed to the edges rather than calculating the edge fluxes directly from the edge variables.

p	N	Operation Count
1	3	23
2	6	165
3	10	532
4	15	1305
5	21	2706

Table 14 Operation counts to perform a projected product of two degree p polynomials when $\xi^i\eta^j$ are used as the basis set.

The alternative to equation (19) for standard polynomials is to actually perform the steps in the process separately. In this fashion, one can approximate the product by truncating terms. Table 15 gives the work to perform the multiplication of two degree p polynomials when retaining terms in the product of degree p, $p+1$, and $2p$. In practice, expansions to degree p are not stable. Expansions to degree $p+1$ have performed adequately for several cases, and the savings compared with table 14 exceeds 75% for the $p = 4$ case. The need to be exact to $p+1$ is in general agreement with the analysis of Cockburn *et al.*[9] that shows that quadrature rules must be exact to degree $2p$ for the volume terms and to degree $2p+1$ for the edge terms. Recall that the volume fluxes are multiplied by the derivative of a basis function before integration, so a degree $p+1$ flux and a degree $p-1$ derivative of a basis function combine to be degree $2p$. On an edge, the combination of a degree $p+1$ flux and a degree p basis function produces a degree $2p+1$ function. In

		Degree of Expansion		
p	N	p	$p+1$	$2p$
1	3	7	12	12
2	6	24	44	57
3	10	60	105	172
4	15	135	207	405
5	21	231	364	816

Table 15 Operation counts to multiply two degree p polynomials when $\xi^i\eta^j$ are used as the basis set.

the present approach, rather than projecting the flux, the $\tilde{\mathbf{A}}_i$ matrices are extended to account for a volume flux of degree $p+1$. Similarly, the $\mathbf{T}_l$ matrices are enlarged. The added work is less than would be incurred by increasing the order of the solution because the matrices only need to have their column spaces increased. Although an expansion to degree $p+1$ typically runs, the truncation of the series can create large errors in certain situations. As the mesh is refined, the coefficients of the truncated terms should become smaller as desired because the basis set can actually be thought of as a Taylor series expansion about the center of the element. However, on a given mesh it is possible for a degree $p = 4$ case to have a mean flow and solution that are well represented by x^3 corresponding to a flux of x^6. This flux would be completely ignored. In other words, for this heuristic case, the real flux would be lost and a zero flux would be used. The x^6 term could be represented by the lower order terms, but then the expensive projection process is required. Clearly, the grid and the smoothness of the solution will determine whether truncation is acceptable. In the orthogonal polynomial case, even a product formula such as equation (20) implicitly has a projection embedded within it. Truncation is also different from what occurs when Gaussian quadrature rules are used to perform the integration. The quadrature finds the integral of the best fit to the data using its basis set. For example, integrating $1+x+x^{10}$ using a quadrature rule that is exact for polynomials of degree 6 will produce a considerably better result than just integrating $1 + x$. However, the quadrature rules generally have to sample the fluxes at many points just to produce acceptable results.

One method to eliminate the need to deal with all of the high degree terms in the linearized Euler equations is to limit the degree of the mean flow. If the mean flow is only represented by a linear variation, only terms to degree $p + 1$ arise in the fluxes. Furthermore, the cost of performing the multiplications becomes much less expensive as there are only three terms for the mean flow. In most cases, the steady, background flow only has large gradients in regions around the body and in wakes. Hence, the mean flow can adequately be described by a linear variation over most of the domain. This approach doesn't present any difficulty for the method, but does add some coding complexity. In addition, the unsteady phenomena under investigation may have scales that are much smaller than the background flow everywhere so that a linear variation of the mean flow is adequate throughout the entire domain.

Polynomial Division

A still more expensive calculation for the quadrature-free approach that is necessary for some nonlinear problems is polynomial division. In the Euler equations, fluxes of the form $\rho u^2 = (\rho u)(\rho u)/\rho$ must be obtained from ρ and ρu. For simplicity, let $\phi = (\rho u)$ and consider $u = \phi/\rho$ or $u\rho = \phi$. Multiplying both sides by the test function b_k and integrating yields

$$\rho_i \int_\Omega b_i b_j b_k \, d\Omega \; u_j = \int_\Omega b_m b_k \, d\Omega \;\; \phi_m \qquad (21)$$

or

$$\begin{aligned} \rho_i R_{ikj} u_j &= \tilde{R}_{kj} u_j = M_{km}\phi_m, \quad \text{where} \\ \tilde{R}_{kj} &= \rho_i \int_\Omega b_i b_j b_k \, d\Omega. \end{aligned} \qquad (22)$$

Essentially, (22) is a matrix equation for the u_j. The right hand side simply involves the multiplication of the mass matrix by ϕ which is at most an $O(N^2)$ operation. However, computing $\tilde{R}_{kj}$ is an $O(N^3)$ operation similar to what was done to compute products. Again, there is considerable sparsity in $\tilde{R}_{kj}$. For $p = 4$, it takes 268 flops for squares and 653 flops for triangles to compute $\tilde{R}_{kj}$ with $\xi^i\eta^j$ basis functions. These counts can be compared with $N^3 = 3375$. Furthermore, $\tilde{R}_{kj}$ is symmetric, which reduces the work again by nearly half. However, $\tilde{R}_{kj}$ must be inverted to find u_j. Even though $\tilde{R}_{kj}$ is symmetric, inversion is still an $O(N^3/3)$ operation. In addition, the matrix is relatively small, and it is difficult to achieve high flop rates in the inversion and backsolves.

An alternative to equation (22) for a $\xi^i\eta^j$ basis set is to expand the inverse of the polynomial in a multidimensional Taylor series. The difficulty is that the series is infinite. Table 16 gives the operation counts to determine the series to degree p, $p+1$, and $2p$. The series to $p+1$ is nearly twice as expensive as multiplying two polynomials, and the expansion to $2p$ is not feasible for most cases. Furthermore, after obtaining an approximation to $1/\rho$, several polynomial products are required to obtain the desired flux. Hence, one has to make decisions about how many intermediate terms to retain even if the final flux is truncated at $p + 1$.

		Degree of Series		
p	N	p	$p+1$	$2p$
1	3	7	18	18
2	6	24	52	91
3	10	60	115	282
4	15	130	224	680
5	21	240	389	1389

Table 16 Operation counts to find the inverse of a degree p polynomial using a Taylor series when $\xi^i\eta^j$ are used as the basis set.

Three-dimensional problems

In progressing to three-dimensions, the work generally doubles for a standard finite-difference scheme as the number of equations increases from four to five, and the dimensions increases from two to three, i.e. (5/4)(3/2) = 15/8. For the DG method, the work does not scale linearly with the number of dimensions because the number of terms in the basis set increases significantly. For three-dimensional problems, the number of volume and edge basis functions is $N = (p+1)(p+2)(p+3)/6$ and $N_e = (p+1)(p+2)/2$, respectively. The number of volume terms is now a cubic function of the degree p of the solution, and the work on each edge is equivalent to the work that was done in the volume for the two-dimensional cases. Furthermore, there are more edges for three-dimensional elements, and a transformation is required to align adjacent edge coordinate systems.

The work for the volume and edge integrations and transformations generally follows the trends observed in the two-dimensional case. Similarity elements that are cubes possess symmetry in all three coordinate directions, and the fill of the $\tilde{\mathbf{A}}_i$ matrices is approximately $N^2/14$. The sparsity for tetrahedra is considerably less. The work to compute nonlinear fluxes in three-dimensions is a strong function of degree of the solution as shown in tables 17 and 18. Clearly, one cannot even consider retaining all of the terms in polynomial products and -inversion, and even an expansion to $p+1$ is around three times as expensive as that for a two-dimensional solution of the same order.

		Degree of Expansion		
p	N	p	$p+1$	$2p$
1	4	10	22	22
2	10	46	108	165
3	20	148	325	716
4	35	385	784	2285
5	56	868	1652	5986

Table 17 Operation counts to multiply two degree p polynomials when $\xi^i\eta^j\zeta^k$ are used as the basis set.

		Degree of Series		
p	N	p	$p+1$	$2p$
1	4	13	28	28
2	10	56	138	284
3	20	168	372	1259
4	35	418	861	4093

Table 18 Operation counts to find the inverse of a degree p polynomial using a Taylor series expansion when $\xi^i\eta^j\zeta^k$ are used as the basis set.

Conclusions

Several optimization procedures are investigated for the quadrature-free DG approach that take advantage of the sparsity of the matrices involved in the computations. Modest improvements are obtained by blocking the matrices to avoid multiplications by zero. By writing specific code to perform all of the core operations, substantial reductions in the CPU time are obtained. Quadrilateral elements with Legendre polynomials as the basis set are the most efficient for constant-coefficient, linear equations. The orthogonality produces extremely sparse matrices with a fill of N. Using the specialized routines and orthogonal polynomials, a savings of up to 35% in the total run time can be achieved. Because the most efficient formulation involves quadrilaterals, most of the computational domain should be composed of this element type. However, completely general quadrilaterals incur a large storage penalty, unlike triangles. The conflicting requirements for efficiency and minimal storage while handling geometric complexity can easily be resolved by the approach because triangular and quadrilateral elements may be intermixed freely. Furthermore, unstructured grid generators can easily create a triangulation around a body with a rectangular outer boundary that can be matched to a Cartesian mesh.

Even when all of the sparsity is accounted for in the calculation of polynomial products and inversions, flux computations will clearly be the limiting factor for nonlinear problems. However, this scenario is common to most numerical algorithms. An analysis of the operation counts for three-dimensional problems shows that the work required for a nonlinear flux is considerably greater than in two dimensions. Several simplified solution procedures are suggested that will minimize the work over the majority of the domain. The flop counts are kept to a minimum when standard polynomials of the form $\xi^i\eta^j$ are used, and the products and inverses are truncated to a degree one higher than that of the solution variables. Truncating the products results in a savings of up to 75%, but there is still considerable work associated with nonlinear fluxes. Because most of the overhead arises from calculating the flux for complicated physics, using simplified physical models wherever appropriate will greatly enhance the efficiency of the scheme.

References

[1]Atkins, H. L. and Shu, C. W., "Quadrature-Free Implementation of Discontinuous Galerkin Method for Hyperbolic Equations," *AIAA Journal*, Vol. 36, No. 5, 1997, pp. 775–782.

[2]Atkins, H. L., Baggag, A., Ozturan, C., and Keyes, D., "Parallelization of an Object-Oriented Unstructured Aeroacoustics Solver," 1999, 9th SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, TX, March 22-24.

[3]Shu, C. W. and Osher, S., "Efficient Implementation of Essentially Non-Oscillatory Shock-Capturing Schemes," *Journal of Computational Physics*, Vol. 77, No. 77, 1988, pp. 439–471.

[4]Sansone, G., *Orthogonal Functions*, chap. 3, Interscience Publishers, Inc, 1959, p. 194.

[5]Dubiner, M., "Spectral Methods on Triangles and Other Domains," *Journal of Scientific Computing*, Vol. 6, 1991, pp. 345.

[6]Sherwin, S. J. and Karniadakas, G. E., "A Triangular Spectral Element Method; Applications to the Incompressible Navier-Stokes Equations," *Computer Methods in Applied Mechanics and Engineering*, Vol. 123, 1995, pp. 189–229.

[7]Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, chap. 22, Dover Pub., Inc., 1965, pp. 771–802.

[8]Adams, J. C., "Legendre's Coefficients," *Proceedings of the Royal Society of London*, Vol. 27, 1878, pp. 63–71.

[9]Cockburn, B., Hou, S., and Shu, C. W., "TVB Runge-Kutta Local Projection Discontinuous Galerkin Finite Element Method for Conservation Laws IV: The Multidimensional Case," *Mathematical Computations*, Vol. 54, 1990, pp. 545–581.