DGSAND: A Discontinuous Galerkin Sand box

Jay Sitaraman

April 8, 2021

1 Background

It has been on my wish list for a long time to write a simple Discontinuous Galerkin code that can be used for testing efficiency and algorithm modifications towards creating a more efficient GPU centric implementation. Finally, I have been able to take a few weeks away from the routine work and put some focussed time on reading papers and writing a code that actually works at least for a simple problem. There are several other open source DG codes produced by many different groups available on the internet. Therefore, it is natural to wonder whetherthere is a need for having yet another code. I don't have a good answer for this other than saying that doing something by myself is the only way I know to learn well enough to be able to explain the same to others. I am grateful to Mike Brazell and Hari Sitaraman for sharing their more sophisticated discontinuous galerkin codes for me to take inspiration from. I am also grateful to Steve Tran and Beatrice Roget for their 1-D codes that provided the mathematical basis for the implementation here. I have tried to simplify the notations in the mathematical description below as much as possible to make it accessible to engineers. In general, I have found most of the finite element method papers rely more than their necessary share on mathematical notation for brevity and appearence of eruditeness. This does tend to frustrate competent engineers who could otherwise have contributed meaningfully to a project.

2 Software

This code is written in pure C with the intention that it can be easily ported to CUDA/HIP as and when it is possible to do so without any major changes to underlying source. I have also tried to minimize the number of

if conditions and possible race scenarios by arranging the loops in fashion slightly different from standard finite volume/finite element codes. There is definitely still a very large room for optimization. I have also stayed away from C++ and usage of any templating because of my own aversion to their over-use in modern codes and also concerns on their efficiency (both of these concerns may be misplaced). Instead, I have opted for a straight arrays that can be indexed in to for picking out the appropriate geometric and field values per element. All filling is done in standard C order order as field values for each degree of freedom for every field for every element. Finally I have also tried (at least for now) to not have any dependencies and stick to as simple a compilation system as possible to build the code.

3 Mathematical Framework

The conservation law for first order hyperbolic systems are given by :

$$\frac{\partial Q}{\partial t} + \nabla \cdot F(Q) = 0 \tag{1}$$

where Q is the conserved variable and $F(Q)$ is the flux of this variable [**?**]. In a Discontinuous Galerkin Finite Element Method, implemented in a semi-discrete form (i.e finite elements in space only), the space is split into N_e non-overlapping elements. Each element is then equipped with a basis set (which are almost always polynomials) that approximate the variation of a field within that element. The coefficients of these polynomials are the unknowns that need to be solved using the chosen numerical method. The finite element method, in this sense is a glorified curve-fitting approach, where the method used to reduce the constraint that the unknowns must satisfy the partial differential equation and its boundary condition into a set of algebraic equations usually defines the type of the method. In a "Discontinuous" approach, all of the basis set of an element is constrained to have identically zero values outside of that element. This stands in contrast to a more well known "continuous" approach where the basis associated with a degree of freedom will have non-zero values on every element that shares it. Following notations used by Atkins [**?**], the basis set is represented as:

$$B \equiv \{b_k, 1 \leq k \leq N(e, p, d)\} \tag{2}$$

Each basis b_k to a particular order is a polynomial of order p of the barycentric coordinates within the reference element, e.g. in 2-D it would be a polynomial composed of monomials from the set $\{1, x, y, x^2, xy, y^2, x^2y, xy^2, x^3..\}$

etc where $\{x, y\}$ represent location within an element. $N(e, p, d)$ represents the number of linearly independent basis functions required to fully represent the spatial variation upto polynomial order p for element type e in dimensions d. In this work, a hierarchical basis is chosen for ease of implementation. The chosen basis for triangles (only ones implemented at this time) are taken from MontJoie user manual [**?**]. Once the basis set is chosen, the field at any location within an element can be approximated (to a given order p) as as a summation as shown below

$$Q_i(u) = \sum_{k=1}^{k=N_i} q_{ki} b_{ki}(u) \tag{3}$$

Here u is the coordinate vector, e.g. $u = \{x, y\}$ in 2D and $u = \{x, y, z\}$ in 3D etc. The subscript i represents a given element i. The entire vector field Q is the union of the approximate solutions that are valid within each element. To reiterate, the basis functions b_{ki} are zero everywhere outside element i. So one can express the field Q approximately as

$$Q_f = \sum_{i=1}^{i=N_e} \sum_{k=1}^{k=N_i} (q_{kf})_i b_{ki} \qquad f \in [1, nfields] \tag{4}$$

The total number of unknowns ($(q_{kf})_i$) are $ndof = nfields * \sum_{i=1}^{i=N_e} N_i$, where nfields is the number of conservation equations at a given spatio-temporal location, e.g. for 2-D Navier-Stokes it will be 4 consiting of conservation of mass (1 eqn), conservation of momentum (2 eqns) and conservation of energy (1 eqn). Substituting in Eq. 4 in Eq. 1 and omitting the subscripts i and f for clarity one obtains

$$\frac{\partial(\sum \sum q_k b_k)}{\partial t} + \nabla \cdot F(\sum \sum q_k b_k) = 0 \tag{5}$$

If the flux F is differentiable across element interfaces one could use Eq. 5 directly to obtain $ndof$ equations for the unknown vector $(q_{kf})_i$. Weaker or integral formulations are more general since they do not require differentiability or even continuity across element interfaces. In general, weaker formulations are constructed by multiplying Eq. 5 by a test function and integrating across the entire domain (Galerkin formulation). One of the constraint for the test functions are that they have to maintain linear independence to create a well posed system of algebraic equations. A convenient choice of the test functions are the basis function themselves which leads to

the following weak integral form.

$$\int_{\Omega} b_j \left[\frac{\partial(\sum \sum q_k b_k)}{\partial t} + \nabla \cdot F(\sum \sum q_k b_k) \right] d\Omega = 0 \tag{6}$$

Where Ω represents the entire space where the conservation law is to be enforced. Noting again that b_{j_i} are b_{ki} are non-zero only within an element i, this space of integration can be reduced to be just within each element for the basis that are associated with that element.

$$\int_{\Omega_i} b_{j_i} \left[\frac{\partial(\sum (q_{kf})_i b_{ki})}{\partial t} + \nabla \cdot F(Q_i)_f \right] d\Omega_i = 0 \tag{7}$$

Equation 7 represents one of the semi-discrete equations in element i associated with basis b_{j_i}. Note that $(Q_i)_f = \sum \sum (q_k)_f b_k$, where f is the index of the field and $F(Q_i)_f$ is the f^{th} row of the flux of Q_i. There are N such equations per field within each element. Combining all of the equations for element i for a given field f and dropping the subscript i for basis and the unknowns yields N equations that can be written as:

$$\left[\int_{\Omega_i} b_j b_k d\Omega_i \right] \frac{\partial(q_k)_f}{\partial t} + \int_{\Omega_i} b_j \nabla \cdot F(Q)_f d\Omega_i = 0, \quad j,k \in [1,N] \tag{8}$$

Note that b_j are b_k are pure spatial functions and hence can be taken outside of the time derivative. The unknowns (polynomial coefficients) are functions of time and the time derivative operator is applied to them. The first term in Eq. 8, [..], is termed the mass matrix in finite element parlance since it scales a velocity derivative akin to a mass-spring-damper equation common in structural dynamics. Each term of the $N \times N$ mass matrix is the integration of the product of each pair of basis functions (including itself) within that element. Considering the second term, of equation 8, one can observe it is the product of scalar with the divergence of a vector. The product rule of divergence is

$$\nabla \cdot (bF) = b\nabla.F + \nabla b.F \tag{9}$$

This is nothing but the generalization of the product rule of derivative in single dimension, which is simply $\frac{\partial(uv)}{\partial x} = u\frac{\partial v}{\partial x} + \frac{\partial u}{\partial x}v$. From Eq. 9, $b\nabla.F = -\nabla b.F + \nabla \cdot (bF)$, substituting in Eq. 8 yields:

$$\left[\int_{\Omega_i} b_j b_k d\Omega_i \right] \frac{\partial(q_k)_f}{\partial t} - \int_{\Omega_i} \nabla b_j \cdot F(Q)_f d\Omega_i + \int_{\Omega_i} \nabla \cdot (b_j F(Q)_f) d\Omega_i = 0 \tag{10}$$

Where Q_i^f is the approximation within the element i for field f given by $\sum q_{k\,i}^f b_{ki}$. Using Gauss-divergence theorem the third term, which is a volume

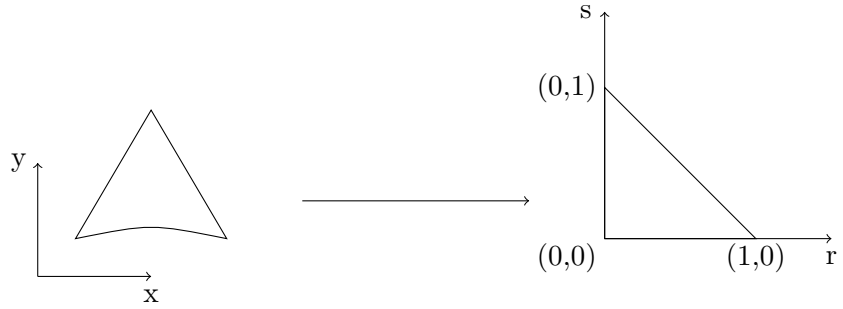
integral that can be expressed as a surface integral over the boundary surface of the element ($\partial\Omega_i$), one obtains:

$$\left[\int_{\Omega_i} b_j b_k d\Omega_i\right] \frac{\partial (q_k)_f}{\partial t} - \int_{\Omega_i} \nabla b_j \cdot F(Q)_f d\Omega_i + \int_{\partial\Omega_i} b_j F(Q)_f \cdot \vec{ds} = 0 \quad (11)$$

At this point we have to develop methods for evaluating the integrals above within and around each element to a sufficient accuracy.

3.1 Coordinate Transformation

Every element can be mapped on to a reference element as shown below. Elements in physical space may have curved boundaries, but the reference elements are unit dimensional with planar faces. An example for a triangle is shown below



The transformation between $(x, y, z) \in \Omega$ physical coordinates in 3D and the $(r, s, t) \in \Lambda$ coordinates within the reference element can be written as:

$$\begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial s} & \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial s} & \frac{\partial y}{\partial t} \\ \frac{\partial z}{\partial r} & \frac{\partial z}{\partial s} & \frac{\partial z}{dt} \end{bmatrix} \begin{bmatrix} dr \\ ds \\ dt \end{bmatrix} = \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix} \quad (12)$$

Where

$$J = \frac{\partial (x, y, z)}{\partial (r, s, t)} = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial s} & \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial s} & \frac{\partial y}{\partial t} \\ \frac{\partial z}{\partial r} & \frac{\partial z}{\partial s} & \frac{\partial z}{dt} \end{bmatrix} \quad (13)$$

is the Jacobian of transformation. Similarly in 2D the transformation will be:

$$J = \frac{\partial (x, y)}{\partial (r, s)} = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial s} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial s} \end{bmatrix} \quad (14)$$

The integrals in Eq. 11 has both volume integral and surface integral terms and these need to be transformed to the reference element to perform the integration.

3.1.1 Transformation of Volume integrals

Considering 2-D, the differential volume (really area in 2D) for integration is often denoted in literature as $d\Omega = dxdy$, however it is actually $|dx\hat{e_x} \times dy\hat{e_y}|$. In Cartesian coordinate system, the unit vectors $\hat{e_x}$ and $\hat{e_y}$ are orthogonal to each other and hence the differential area reduces to simply $dxdy$. The vector $dx\hat{e_x}$ can be expressed as $dx\hat{e_x} = (\frac{\partial x}{\partial r}dr)\hat{e_r} + (\frac{\partial x}{\partial s}ds)\hat{e_s}$ using vectors $\hat{e_r}$ and $\hat{e_s}$ of the the reference coordinate system. Note that $\hat{e_r}$ and $\hat{e_s}$ are still in the physical coordinate system and not necessarily orthogonal to each other. However, $\hat{e_r} \times \hat{e_s}$ always points in the same direction. Using these ideas, we can expand

$$|dx \times dy| = \left| \left((\frac{\partial x}{\partial r}dr)\hat{e_r} + (\frac{\partial x}{\partial s}ds)\hat{e_s} \right) \times \left((\frac{\partial y}{\partial r}dr)\hat{e_r} + (\frac{\partial y}{\partial s}ds)\hat{e_s} \right) \right| \quad (15)$$

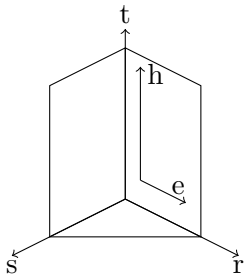
$$= \left| \frac{\partial x}{\partial r}\frac{\partial y}{\partial s} - \frac{\partial y}{\partial r}\frac{\partial x}{\partial s} \right| drds \quad (16)$$

$$= |J|drds \quad (17)$$

Where $|J|$ is the determinant of the Jacobian of transformation. In similar vain, the differential volume in 3D is given by the triple product $dx\hat{e_x}.(dy\hat{e_y} \times dz\hat{e_z})$ in physical coordinate space, which transforms to $|J|drdsdt$ within the reference element.

3.1.2 Transformation of Surface integral

The integrals of type $\int_{d\Omega}(..).\vec{\partial s}$ is the same as $\int_{d\Omega}(..) \cdot \hat{n}|ds|$, where $\hat{n}$ is the unit normal on the element surface (this would vary for elements with curved boundaries). Consider the prismatic reference element below:



The differential area vector along a boundary face equipped with an edge-wise (e) and height-wise (h) coordinate system is given by :

$$\vec{\partial s} = (\hat{e} \times \hat{h})dedh \quad (18)$$

The vectors $\hat{e}$ and $\hat{h}$ need to be expressed in the physical coordinate system to reconstruct the expression for the differential area vector in reference coordinates. Using the derivatives w.r.t (h, e) as directional changes these vectors can be expressed as:

$$\hat{e} = \frac{\partial x}{\partial e}\hat{e_x} + \frac{\partial y}{\partial e}\hat{e_y} + \frac{\partial z}{\partial e}\hat{e_z} \tag{19}$$

$$\hat{h} = \frac{\partial x}{\partial h}\hat{e_x} + \frac{\partial y}{\partial h}\hat{e_y} + \frac{\partial z}{\partial h}\hat{e_z} \tag{20}$$

The vectors $\hat{e}$ and $\hat{h}$ cannot however be arbitrary, they have to be constrained to lie on the physical face of the element. This constraint can be also enforced by making sure they lie on the planar face of the reference element. Each of the terms in Eq. 20, $\frac{\partial x}{\partial e}, \frac{\partial y}{\partial e}, \frac{\partial z}{\partial e}$.. etc can be expanded as follows in terms of the reference coordinates as

$$\frac{\partial x}{\partial e} = \frac{\partial x}{\partial r}\frac{\partial r}{\partial e} + \frac{\partial x}{\partial s}\frac{\partial s}{\partial e} + \frac{\partial x}{dt}\frac{dt}{\partial e} \tag{21}$$

$$\frac{\partial y}{\partial e} = \frac{\partial y}{\partial r}\frac{\partial r}{\partial e} + \frac{\partial y}{\partial s}\frac{\partial s}{\partial e} + \frac{\partial y}{dt}\frac{dt}{\partial e} \tag{22}$$

$$\frac{\partial z}{\partial e} = \frac{\partial z}{\partial r}\frac{\partial r}{\partial e} + \frac{\partial z}{\partial s}\frac{\partial s}{\partial e} + \frac{\partial z}{dt}\frac{dt}{\partial e} \tag{23}$$

The equation for $\hat{h}$ is analogous. On inspection it follows that

$$\hat{e} = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial s} & \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial s} & \frac{\partial y}{\partial t} \\ \frac{\partial z}{\partial r} & \frac{\partial z}{\partial s} & \frac{\partial z}{dt} \end{bmatrix} \begin{bmatrix} \frac{\partial r}{\partial e} \\ \frac{\partial s}{\partial e} \\ \frac{dt}{\partial e} \end{bmatrix} = J \begin{bmatrix} \frac{\partial r}{\partial e} \\ \frac{\partial s}{\partial e} \\ \frac{dt}{\partial e} \end{bmatrix} \tag{24}$$

$$\hat{h} = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial s} & \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial s} & \frac{\partial y}{\partial t} \\ \frac{\partial z}{\partial r} & \frac{\partial z}{\partial s} & \frac{\partial z}{\partial t} \end{bmatrix} \begin{bmatrix} \frac{\partial r}{\partial h} \\ \frac{\partial s}{\partial h} \\ \frac{dt}{\partial h} \end{bmatrix} = J \begin{bmatrix} \frac{\partial r}{\partial h} \\ \frac{\partial s}{\partial h} \\ \frac{dt}{\partial h} \end{bmatrix} \tag{25}$$

The vectors $a = \left[\frac{\partial r}{\partial e}, \frac{\partial s}{\partial e}, \frac{dt}{\partial e}\right]^T$ and $b = \left[\frac{\partial r}{\partial h}, \frac{\partial s}{\partial h}, \frac{dt}{\partial h}\right]^T$ represent the constraint of being on the boundary face and can be found trivially by following the variation of (r, s, t) on the respective face of the reference element. For example, $a = [1, 0, 0]$ and $b = [0, 0, 1]$ for the prismatic reference element shown earlier. The differential area vector can now be expressed in the face coordinate system as:

$$\vec{\partial s} = (\hat{e} \times \hat{h}) dedh = (Ja \times Jb) dedh \tag{26}$$

In 2-D, the $\hat{h}$ is just equal $\hat{e_z}$. So the elemental area (really arc length) reduces to

$$\vec{\partial s} = (Ja \times \hat{e_z}) de \tag{27}$$

For a unit reference triangle, with coordinates $(0,0),(1,0),(0,1)$, the vector $a = [\frac{\partial r}{\partial e}\frac{\partial s}{\partial e}]^T$ is equal to $[1,0]$, $[-1,1]$ and $[0,-1]$ for edges 1, 2 and 3 respectively. Several references (e.g. Ref [**?**]), at least to my understanding, rewrite the tranformation of a differential area to the reference element as $J\vec{n_b}dedh$, where $\vec{n_b}$ is the area scaled normal of the reference element face. Note that $\vec{n_b}$ is just $\vec{a}\times\vec{b}$. To obtain this result from Eq. 26, however J must be a rotation matrix such that it satisfies $J^{-1} = J^T$.

3.2 Gradient Evaluation

Gradient terms such as ∇b, ∇q can be evaluated in the reference coordinates Λ, by using the analytical derivatives of the basis functions, i.e.

$$(\nabla b)_\Lambda = \left[\frac{\partial b}{\partial r}, \frac{\partial b}{\partial s}, \frac{\partial b}{\partial t}\right] \tag{28}$$

$$(\nabla Q)_\Lambda = \sum_{k=1}^{k=N} q_k (\nabla b_k)_\Lambda \tag{29}$$

The gradient in physical coordinates can be expressed by tranformation of coordinates as:

$$(\nabla b) = \left[\frac{\partial b}{\partial r}, \frac{\partial b}{\partial s}, \frac{\partial b}{\partial t}\right] \begin{bmatrix} \frac{\partial r}{\partial x} & \frac{\partial r}{\partial y} & \frac{\partial r}{\partial z} \\ \frac{\partial s}{\partial x} & \frac{\partial s}{\partial y} & \frac{\partial s}{\partial z} \\ \frac{\partial t}{\partial x} & \frac{\partial t}{\partial y} & \frac{dt}{\partial z} \end{bmatrix} = (\nabla b)_\Lambda J^{-1} \tag{30}$$

3.3 Semi-discrete equation in reference coordinates

Equation 11 can be now expressed in the reference element coordinates as:

$$\left[\int_\Lambda b_j b_k |J| d\Lambda\right] \frac{\partial (q_k)_f}{\partial t} - \int_\Lambda \nabla b_j J^{-1} \cdot F(Q)_f |J| d\Lambda + \int_{\partial\Lambda} b_j F(Q)_f \cdot (Ja \times Jb) d\lambda = 0 \tag{31}$$

Here Λ and $d\Lambda$ are always within and around the reference element, i.e. their integration limits are always $[0,1]$. The Jacobian $J \equiv J_i$ provides the scaling and rotation that maps the physical element to the reference element.

Equation 31 can be further sub-divided as :

$$mass\_matrix = [M] = \left[\int_{\Lambda} b_j b_k |J| d\Lambda\right] \tag{32}$$

$$volIntegral = \int_{\Lambda} \nabla b_j J^{-1} \cdot F(Q)_f |J| d\Lambda \tag{33}$$

$$faceIntegral = \int_{\partial\Lambda} b_j F(Q)_f \cdot (Ja \times Jb) d\lambda \tag{34}$$

$$[M] \frac{\partial (q_k)_f}{\partial t} = volIntegral - faceIntegral \tag{35}$$

Once the right hand side of the above equation is evaluated, one can easily integrate in time using an explicit Runge-Kutta type method. Implicit time stepping methods require linearization of the right hand side and will be dealt with later. Analytical evaluation of the integrals on the RHS is possible if all of the integrands can be expanded as polynomial terms in (r, s, t). A quadrature free approach [**?**] seeks to do exactly that. However, the process of analytically determining all of the coefficients quickly becomes complicated for non-linear flux functions and curved elements – expansions of both fluxes and Jacobians has to computed as functions of polynomial coefficients. This approach is not attempted at this point, although it is in the realm of possibilities to improve efficiency. Currently standard quadrature based integration techniques, where each integral is found as a summation is used.

3.4 Face Integral and connection between elements

The defining feature of the DG method is the discontinuity of the solution field at element interfaces. So $F(Q)$ at the interface is really a function of the states Q_L and Q_R or $\hat{F(Q)}.\vec{n} = \hat{F}(Q_L, Q_R, \vec{(n)})$. Therefore, the face integral in the equations above has to be reconciled using a numerical flux function. The simplest approach for purely convective flux is to use an upwind biased flux that can be expressed as:

$$F(Q_L, Q_R).\vec{n} = \frac{1}{2}((F(Q_L) + F(Q_R).\vec{n} - \alpha(Q_R - Q_L)||n||) \tag{36}$$

Flux function based on Roe's approximate Riemann Solver [**?**], which takes similar form as the equation above is implemented. Any other flux function that can accept a left state, right state and a scaled normal vector as arguments and output the numerical flux can be used. Note that for gradient

dependent (diffusive type) fluxes, penalty functions or lift operators need to be implemented to reconcile the discontinuity of derivatives at the element interfaces. These will be implemented shortly.

3.5 Quadrature rules

All volume integrals, whose integrands can be represented as $I(r, s, t)$ are evaluated as

$$\int_{\Lambda} I(r, s, t) d\Lambda = \sum_{g=1}^{g=M_g} w_g I(r_g, s_g, t_g) \tag{37}$$

All face integrals, whose integrands can be represented as $J(r, s, t)$ are evaluated as

$$\int_{d\Lambda} J(r, s, t) d\lambda = \sum_{g=1}^{g=M_f} w_g J(r(e_g, h_g), s(e_g, h_g), t(e_g, h_g)) \tag{38}$$

Where (r_g, s_g, t_g, w_g) and (e_g, h_g, w_g) are the locations and weights of the Gauss quadrature points for volume and face integration respectively.

4 Code architecture

All data is stored in linear double precision arrays that can be indexed into using for each element. The integer array *iptr* provides the index into all volume data (geometry and solution), while the *iptf* is the index list for all face data. The overall code execution follows the following pattern: Computing the right hand side is the most expensive operation. It is performed in 3 steps. First all cells go through all of their bounding faces and project the solution to the face quadrature points. Each face quadrature point has $3 \times nfields$ storage space. The face data is filled according a pre-deterimined fill order such that the cell on the left side of the face fills $[0 : nfields - 1]$, the cell on the right side fills $[nfields : 2 * nfields - 1]$, the last $nfields$ data space is left for the flux function to fill in the flux. It is important to make sure that the face quadrature points are filled in the right order. In 2-D this is easier to achieve with the left cell filling in forward order and right cell filling in backward order. Arguably this is going to be one of the more complex topological issues to solve in 3D. Once all the states on the left and right side are filled, each face computes the Riemann flux using this states. At this point, each element has the full information to completely compute the residual for all of its degrees of freedom and

Algorithm 1 `DGSAND`

```
1: Read_Grids()
2: Find_Connectivity()
3: Allocate_Memory()
4: Initialize_Grid_And_Solution_Coefficients()
5: ProcedureCompute_Geometric_Parameters()
6: for n=1:nsteps do
7:     for r=1:rksteps do
8:         R_r ← ComputeRHS(q_r, ..)
9:     end for
10:    q_{n+1} ← update(q_n,R_1,R_2..)
11: end for
```

every element performs this task. The execution follows a compute-scatter-compute-gather-compute, which avoids any kind of race condition and can easily be multi-threaded. The algorithm is summarized below.

Algorithm 2 `ComputeRHS`

```
1: for i=1,N_e do                                          ▷ For each element
2:     for f=1,Nf_i do                                     ▷ For each face of this element
3:         FILL_FACES()                                    ▷ fill states in to the face data array
4:     end for
5: end for
6: for f=1,Nbf do                                          ▷ For each physical boundary face
7:     FILL_BC()                                           ▷ fill appropriate boundary condition states
8: end for
9: for f=1,Nf do                                           ▷ For all faces
10:    COMPUTE_FACE_FLUXES()                               ▷ find numerical flux on each face
11: end for
12: for i=1,N_e do
13:    R_v ← volumeIntegral()                              ▷ volume integral using M_g quadrature points
14:    for f=1,Nf_i do
15:        R_f ← faceIntegral()                            ▷ face integral using M_f quadrature points
16:    end for
17:    Residual ← R_v − R_f
18: end for
```