The Discrete Operator Approach to the Numerical Solution of Partial Differential Equations

Conference Paper · June 2011

DOI: 10.2514/6.2011-3377

2 authors:

James Sutherland
University of Utah
79 PUBLICATIONS 1,328 CITATIONS
SEE PROFILE

Tony Saad
University of Utah
50 PUBLICATIONS 288 CITATIONS
SEE PROFILE

The Discrete Operator Approach to the Numerical Solution of Partial Differential Equations

James C. Sutherland[*] and Tony Saad[†]
University of Utah, Salt Lake City, UT 84112, USA

The design of robust computational physics codes has always been a challenge to application programmers. One of the key difficulties in writing multiphysics codes stems from the inefficient handling of spatial discretization and field operations. For example, in the context of the finite volume (FV) method, one often deals with structured and unstructured meshes that are either staggered or collocated. To handle this array of options, a complex data structure is required to represent arbitrary meshes. For structured grids, this approach imposes an unnecessary computational overhead that increases significantly with problem size. Instead, the programmer must write specific components to handle structured meshes thus defeating the purpose of code portability. Furthermore, dedicated discretization schemes are required for different types of meshes, thus increasing software complexity. Although modern computational codes rely extensively on a variety of libraries for linear algebra operations, they lack a framework that provides proper application-independent discretization tools. In this manuscript, we present a novel computational paradigm that separates the spatial discretization and field operations from the physics. This paradigm is based on the abstraction of the mathematical operators describing physical processes. An operator corresponds to a precise mathematical object that performs a certain calculation on a field. A field corresponds to any scalar or vector variable required in the solution process. In our model, an operator is represented discretely by a sparse matrix while a field is represented by a vector. At the outset, the discretization process corresponds to a sparse-matrix-vector multiplication. This approach completely decouples the physics from the spatial and field operations thus providing an avenue for improved code design and usability.

Nomenclature

$\boldsymbol{\tau}$	=	Stress tensor
ψ	=	Generic scalar field
ρ	=	Density
$\mathcal{S}$	=	Surface
$\mathcal{S}_{ij}$	=	Staggered surface direction
τ_{ij}	=	Stress tensor
$\boldsymbol{u}$	=	Velocity field
φ	=	Generic scalar field
$\mathcal{V}$	=	Volume
$\mathcal{V}_i$	=	Staggered volume location
p	=	Pressure
t	=	Time

Operators

$\mathcal{G}$	=	Discrete gradient operator
$\mathcal{I}$	=	Discrete integral operator
$\mathcal{L}$	=	Generic operator
$\mathcal{R}$	=	Discrete interpolant operator

[*]Assistant professor, Department of Chemical Engineering.
[†]Post Doctoral Fellow, Institute for Clean and Secure Energy. Member AIAA.

I. Introduction

Computational physics is a relatively new field that now stands with experiment and theory at the heart of modern scientific investigations. While experimental and theoretical approaches have traditionally provided the necessary tools to analyze natural processes, computational methods deliver fine-grained information to an arbitrary level of accuracy and precision. It is also intersting to note that analytical techniques constitute the infrastructure of all numerical methods.[1] Although the ideas of numerical approximation of differential quantities were known since the early development of the calculus, their use was limited due to the lack of computational tools. However, with the advent of microprocessors in the 19th century, more attention was given to the use of computers to solve differential equations. The pioneering work of Richardson[1] is recognized as being the first modern application of computational techniques to practical problems.

Subsequent progress of numerical techniques was strongly correlated with developments in computer architecture and programming languages. In the early stages of computational physics, scientists were focused on solving problems of physical interest rather than efficient software design and programming. As numerical methods matured and problems became more complex, a paradigm shift occured and now software engineering is (and should be) an integral part of any computational endeavor.

One of the central challenges in writing multiphysics codes is the efficient handling of spatial discretization and field operations. One often deals with structured and unstructured meshes that are either staggered or collocated. To handle this array of options, a complex data structure is required to represent arbitrary meshes. Unfortunately, for structured grids, this approach imposes an unnecessary computational overhead that increases significantly with problem size. Structured grids do not require advanced data representation as one is able to iterate through the grid via proper indexing. Instead, the programmer must write specific components to handle structured meshes thus defeating the purpose of code portability. Furthermore, dedicated discretization schemes are required for different meshes, therefore increasing software complexity.

The current foundation of computational software design is based on partial separation of the physics from the numerics. While modern codes rely on external libraries for linear algebra operations such as iterative solvers, they lack a framework that provides application-independent discretization tools. In this manuscript, we present a novel computational paradigm that allows separation of the spatial discretization and field operations from the physics. This paradigm is based on the abstraction of the mathematical operators describing physical processes. An operator corresponds to a precise mathematical object that performs a certain calculation on a field. A field corresponds to any scalar or vector variable required in the solution process. In its most general form, an operator is represented discretely by a sparse matrix while a field is represented by a vector. At the outset, the discretization process corresponds to a matrix-vector multiplication. This approach decouples the physics from the spatial and field operations thus complementing the physics-numerics cycle and providing an avenue for better code design and usability.

This paper is organized as follows. We first introduce the theoretical aspects of fields and operators as well as their proper representation on computers. Next we apply the discrete operator approach to the transport equations of a compressible fluid at low Mach number such as those arising in reactive flow systems. These will be presented for a staggered orthogonal grid using the finite volume method. We then introduce our software implementation of the discrete operator approach and discuss various design considerations as well as its development status. Finally we provide an example for programming a diffusive flux expression and conclude the paper with a summary of our implementation and list of its advantages.

II. Theoretical Formulation

Our focus in this presentation is on time dependent partial differential equations that are to be solved on staggered[2], structured meshes using the finite volume method. Discretization methods on staggered grids have many desirable characteristics such as conservation of momentum, circulation, and kinetic energy.[3,4] The latter is particularly important for large eddy simulations.[5]

We start by writing a general transport equation as

$$\frac{\partial \varphi}{\partial t} = \mathcal{A}(\varphi, \vec{x}, t)\,, \tag{1}$$

where φ is the dependent variable and $\mathcal{A}(\varphi)$ is a nonlinear operator that represents a variety of physical processes such as convection, diffusion, and any other source terms. In what follows, we introduce the concepts of fields and

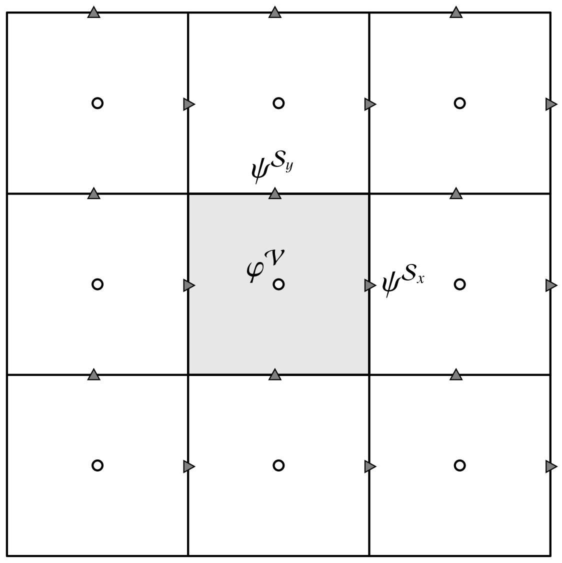

Figure 1. Non-Staggered structured mesh showing volume and face fields.

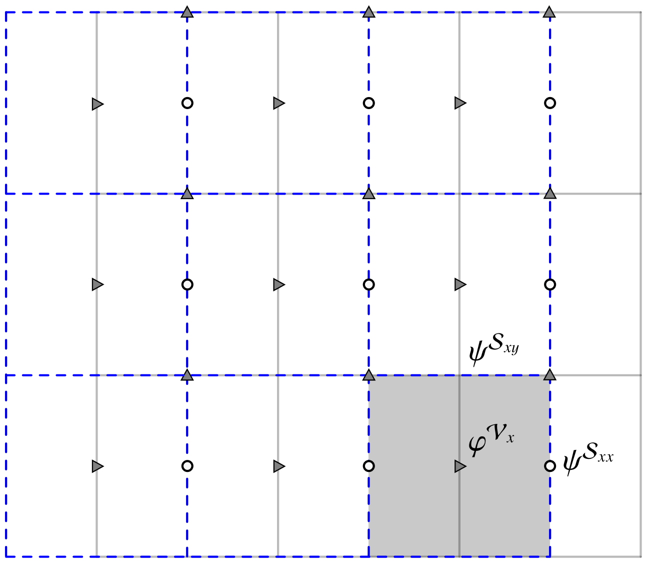
$\psi^{\mathcal{S}_{xy}}$
$\varphi^{\mathcal{V}_x}$
$\psi^{\mathcal{S}_{xx}}$

Figure 2. x-staggered structured mesh showing volume field $\varphi^{\mathcal{V}_x}$ as well as x-face, and y-face fields $\psi^{\mathcal{S}_{xx}}$ and $\psi^{\mathcal{S}_{yx}}$, respectively.

operators.

A. Fields

A field φ is defined as any scalar, vector, or tensor quantity that is associated with particular values on a mesh. Viscosity, velocity, and stress are examples of scalar, vector, and tensor fields, respectively. Fields can be constant or variable in time and space. For example, in a homogeneous incompressible isothermal flow, viscosity is a constant scalar field while velocity is calculated based on the momentum equations. Fields are located on predefined points on a mesh, typically corresponding to volume or surface centroids. Hence, we define two types of fields, volume and surface fields. In practice, a field is represented by a one dimensional contiguous array whether it corresponds to a scalar, vector, or tensor variable. For vector and tensor fields, we reprsent each component as a scalar field. For example, the velocity field is represented as three scalar fields corresponding to its three components, u, v, and w.

1. Non-Staggered Grids

For a non-staggered grid, a volume field is given the superscript $\mathcal{V}$ while a surface field is referred to via $\mathcal{S}_i$. Here, $i = x, y, z$ is the direction of the face normal on which the field is located. For instance, a non-staggered volumetric field φ is written as $\varphi^{\mathcal{V}}$ while a non-staggered x-face field ψ is designated by $\psi^{\mathcal{S}_x}$ as shown in Fig. 1.

2. Staggered Grids

To distinguish between staggered and non-staggered control volumes, we add the subscripts x, y, and z for staggered locations. These refer to the offset in the axial, transverse, and azimuthal directions, respectively. While for non-staggered meshes a volumetric subscript is not required, staggered volume fields are written as $\mathcal{V}_x$, $\mathcal{V}_y$, and $\mathcal{V}_z$, respectively. The staggered surfaces can be described using a tensor $\mathcal{S}_{ij}$ where i denotes the surface normal direction and j denotes the staggered direction. This is written as

$$\begin{bmatrix} \mathcal{S}_{xx} & \mathcal{S}_{xy} & \mathcal{S}_{xz} \\ \mathcal{S}_{yx} & \mathcal{S}_{yy} & \mathcal{S}_{yz} \\ \mathcal{S}_{zx} & \mathcal{S}_{zy} & \mathcal{S}_{zz} \end{bmatrix}. \tag{2}$$

Using this nomenclature, an x-staggered volume field φ is denoted by $\varphi^{\mathcal{V}_x}$, while $\psi^{\mathcal{S}_{xy}}$ refers to an y-staggered x-face field as illustrated in Fig. 2. Occasionally, one may want to refer to an entire staggered field with no specified direction. In this case, either a vector subscript $\vec{x}$ or index i may be employed, e.g. $\varphi^{\mathcal{V}_{\vec{x}}}$ or $\varphi^{\mathcal{V}_i}$.

B. Operators

An operator is defined as an object used to calculate new fields based on existing ones. It consumes a particular field type and produces another. In general, an operator $\mathcal{L}$ acting on a field φ is written as

$$\left[{}^{\text{out type}}\mathcal{L}^{\text{in type}}\right](\varphi)^{\text{in type}}. \tag{3}$$

Note that in Eq. (3), the field type of φ must match the input type of the operator. It is therefore not necessary to explicity designate the type of field being consumed as it may be inferred from the input type of the operator. For example, $\left[{}^{\mathcal{S}_{yx}}\mathcal{L}^{\mathcal{V}_x}\right](\varphi)$ is an operator that consumes an x-staggered volume field and produces and x-staggered y-face field.

1. Operator Types

We identify three types of operators:

- Gradient: $\mathcal{G}$
- Interpolant: $\mathcal{R}$
- Integral: $\mathcal{I}$

A gradient operator is used to calculate the directional derivatives of a field while an interpolant operator can be used to convert between a variety of field types. Gradient and Interpolant operators can process both volume and face fields. Their output depends on the stencil used and can generally be of any field type. For example, an interpolant may be invoked to convert between volume and surface fields or from staggered to non-staggered locations. Finally, an integral operator can consume both volume and surface fields to produce a volume field. The location of this field (i.e. staggered or not) depends on that of the integrand. For instance, assuming that axial momentum ρu resides at x-staggered cell centroids, then

$$\int_{\mathcal{V}_x} \frac{\partial \rho u}{\partial t}\, \mathrm{d}\mathcal{V}_x = \left[{}^{\mathcal{V}_x}\mathcal{I}^{\mathcal{V}_x}\right]\left(\frac{\partial \rho u}{\partial t}\right). \tag{4}$$

This operation takes an x-staggered volume field as both, input and output. On the other hand, given an x-staggered x-face field q, then the following surface integral

$$\int_{\mathcal{S}_{xx}} q\, \mathrm{d}\mathcal{S}_{xx} = \left[{}^{\mathcal{V}_x}\mathcal{I}^{\mathcal{S}_{xx}}\right](q), \tag{5}$$

produces an x-staggered volume field. It is also important to note that when the input is a surface field, the integral corresponds to a surface integral.

2. Operator Stacking

We define operator *stacking* as successive operations on a field. These nested operations are actually matrix-matrix multiplications of the stacked operators. Evaulation of stacked operators proceeds from right to left. The output type of a nested operator must therefore match the input type of the one to its left. This may be written as

$$\left[{}^{\text{o}_n}\mathcal{L}_n^{\text{i}_n}\right]\left[{}^{\text{o}_{n-1}}\mathcal{L}_{n-1}^{\text{i}_{n-1}}\right]\cdots\left[{}^{\text{o}_2}\mathcal{L}_2^{\text{i}_2}\right]\left[{}^{\text{o}_1}\mathcal{L}_1^{\text{i}_1}\right](\varphi); \quad i_{j+1} \equiv o_j,\ \ j \geq 2. \tag{6}$$

In this sense, the stacked operators form a linked chain with a single field type output. The volume integral of the pressure gradient describes one such example, shown here in the x-direction

$$\int_{\mathcal{V}_x} \frac{\partial p}{\partial x}\, \mathrm{d}\mathcal{V}_x = \left[{}^{\mathcal{V}_x}\mathcal{I}^{\mathcal{V}_x}\right]\left[{}^{\mathcal{V}_x}\mathcal{G}^{\mathcal{V}}\right](p). \tag{7}$$

Here, $\left[{}^{\mathcal{V}_x}\mathcal{G}^{\mathcal{V}}\right](p)$ calculates the gradient of a volume field p and produces an x-staggered field. Then, the integral is carried out over the staggered field and stores the result at the same staggered location. In this case, evaluation of the pressure gradient is required first before the integration is carried out. Alternatively, one may calculate the compound operator $\left[{}^{\mathcal{V}_x}IG^{\mathcal{V}}\right] = \left[{}^{\mathcal{V}_x}\mathcal{I}^{\mathcal{V}_x}\right]\left[{}^{\mathcal{V}_x}\mathcal{G}^{\mathcal{V}}\right]$ by performing a matrix–matrix multiplication and then apply the newly formed operator to the pressure field.

3. Matrix and Stencil-Loop Representations

To understand how an operator is constructed, consider the calculation of the axial derivative of a non-staggered volume field $p^{\mathcal{V}}$ on a one-dimensional, five-cell uniform grid. Assume that we wish to locate the resulting field on the x-faces of the non-staggered mesh. Using central differencing, we obtain

$$\begin{bmatrix} p'_1 \\ p'_2 \\ p'_3 \\ p'_4 \\ p'_5 \end{bmatrix} = \begin{bmatrix} -\frac{1}{\Delta x} & \frac{1}{\Delta x} & 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{\Delta x} & \frac{1}{\Delta x} & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{\Delta x} & \frac{1}{\Delta x} & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{\Delta x} & \frac{1}{\Delta x} & 0 \\ 0 & 0 & 0 & 0 & -\frac{1}{\Delta x} & \frac{1}{\Delta x} \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p3 \\ p4 \\ p5 \end{bmatrix}. \tag{8}$$

It is evident that an operator can be represented by an $m \times n$ matrix whose size depends on the input and output fields. It can be easily shown that m is the size of the output field array while n is the size of the input field array. The entire operation then corresponds to a matrix-vector multiplication that may be efficiently implemented using available libraries.

For unstructured meshes, operators are constructed using connectivity information from the mesh to determine the sparsity structure as well as the coefficients of an operator. For structured meshes, connectivity information is easily inferred from the number of grid points. Therefore, it is more efficient to implement the matrix-vector multiplication as a stencil loop over the fields. This allows one to improve the overall performance of a code as well as reducing memory requirements.

4. Discretization Order

Operators can be implemented to an arbitrary discretization order. In that case, the size of the stencil and its coefficients change accordingly. For the matrix representation, this implies a change in the sparsity structure of the mesh, while for the stencil-loop representation, specific changes to the iterators (inside the stencil loop) must be accounted for.

III. Application to Transport Equations

By way of illustration, we apply the discrete operators to the staggered finite volume discretization of a compressible flow. For simplicity, we will not include the transport of chemical species. At the outset, we have the following system of equations

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \boldsymbol{u}), \tag{9}$$

$$\frac{\partial (\rho \boldsymbol{u})}{\partial t} = -\nabla p - \nabla \cdot (\rho \boldsymbol{u} \otimes \boldsymbol{u}) - \nabla \cdot \boldsymbol{\tau}, \tag{10}$$

where

$$\boldsymbol{\tau} = -\mu \left[\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T\right] + \tfrac{2}{3}\mu \delta \nabla \cdot \boldsymbol{u}, \tag{11}$$

is the Newtonian stress tensor. In Eq. (11), δ is the second order identity tensor. The use of a staggered grid requires setting the solution variables at different locations.[2,6] In this discussion, we assume that all scalar variables are located at non-staggered cell centers, while momentum components are located at staggered control volumes. These are summarized in Table 1 . Application of the finite volume method proceeds by first integrating the continuity and

Table 1. Summary of common transport variables and their corresponding field types and locations on a mesh.

Variable	Name	Field Type	Location	Designation
ρ	Density	Volumetric	Non-staggered	$\rho^{\mathcal{V}}$
p	Pressure	Volumetric	Non-staggered	$p^{\mathcal{V}}$
u	Axial velocity	Volumetric	x-staggered	$u^{\mathcal{V}_x}$
v	Transverse velocity	Volumetric	y-staggered	$v^{\mathcal{V}_y}$
w	Azimuthal velocity	Volumetric	z-staggered	$w^{\mathcal{V}_z}$

momentum equations over control volumes corresponding to the variable being solved for.[6] This may be accomplished by setting

$$\int_{\mathcal{V}} \frac{\partial \rho}{\partial t}\,\mathrm{d}\mathcal{V} = -\int_{\mathcal{V}} \nabla\cdot(\rho \boldsymbol{u})\,\mathrm{d}\mathcal{V} = -\int_{\mathcal{S}} \rho\,\boldsymbol{u}\cdot\boldsymbol{n}\,\mathrm{d}\mathcal{S}, \tag{12}$$

$$\int_{\mathcal{V}_x} \frac{\partial \rho u}{\partial t}\,\mathrm{d}\mathcal{V}_x = -\int_{\mathcal{S}_{\vec{x}x}} \rho u\boldsymbol{u}\cdot\boldsymbol{n}\,\mathrm{d}\mathcal{S}_{\vec{x}x} - \int_{\mathcal{S}_{\vec{x}x}} (\boldsymbol{\tau}\cdot\boldsymbol{i})\cdot\boldsymbol{n}\,\mathrm{d}\mathcal{S}_{\vec{x}x} - \int_{\mathcal{V}_x} \frac{\partial p}{\partial x}\,\mathrm{d}\mathcal{V}_x, \tag{13}$$

$$\int_{\mathcal{V}_y} \frac{\partial \rho v}{\partial t}\,\mathrm{d}\mathcal{V}_y = -\int_{\mathcal{S}_{\vec{x}y}} \rho v\boldsymbol{u}\cdot\boldsymbol{n}\,\mathrm{d}\mathcal{S}_{\vec{x}y} - \int_{\mathcal{S}_{\vec{x}y}} (\boldsymbol{\tau}\cdot\boldsymbol{j})\cdot\boldsymbol{n}\,\mathrm{d}\mathcal{S}_{\vec{x}y} - \int_{\mathcal{V}_y} \frac{\partial p}{\partial y}\,\mathrm{d}\mathcal{V}_y, \tag{14}$$

$$\int_{\mathcal{V}_z} \frac{\partial \rho w}{\partial t}\,\mathrm{d}\mathcal{V}_z = -\int_{\mathcal{S}_{\vec{x}z}} \rho w\boldsymbol{u}\cdot\boldsymbol{n}\,\mathrm{d}\mathcal{S}_{\vec{x}z} - \int_{\mathcal{S}_{\vec{x}z}} (\boldsymbol{\tau}\cdot\boldsymbol{k})\cdot\boldsymbol{n}\,\mathrm{d}\mathcal{S}_{\vec{x}z} - \int_{\mathcal{V}_z} \frac{\partial p}{\partial z}\,\mathrm{d}\mathcal{V}_z. \tag{15}$$

The solution process of these equations requires proper handling of the pressure. One commonly used technique to handle pressure-velocity coupling is the pressure projection method.[7,8] Discussion of pressure projection and related methods is out of the scope of this paper.

The discrete operators can now be applied to equations (12)-(15), starting with the continuity equation (12)

$$\left[{}^{\mathcal{V}}\mathcal{I}^{\mathcal{V}}\right]\left(\frac{\partial \rho}{\partial t}\right) = -\left[{}^{\mathcal{V}}\mathcal{I}^{\mathcal{S}_{\vec{x}}}\right]\left[{}^{\mathcal{S}_{\vec{x}}}\mathcal{R}^{\mathcal{V}_{\vec{x}}}\right](\rho\boldsymbol{u})^{\mathcal{V}_{\vec{x}}}\cdot\boldsymbol{n}. \tag{16}$$

Here, the left hand side is the integral of a volume field and is stored at non-staggered locations. For the right hand side, given that $\rho\,\boldsymbol{u}$ is a staggered volume field, it must first be interpolated to non-staggered sufraces before integration is carried out. Hence the use of the interpolant operator $\left[{}^{\mathcal{S}_{\vec{x}}}\mathcal{R}^{\mathcal{V}_{\vec{x}}}\right]$.

Next, we write the momentum equations in operator form. For brevity, this will be illustrated using the x-momentum equation. Using

$$\left[{}^{\mathcal{V}_x}\mathcal{I}^{\mathcal{V}_x}\right]\left(\frac{\partial \rho u}{\partial t}\right) = F_x - \left[{}^{\mathcal{V}_x}\mathcal{I}^{\mathcal{V}_x}\right]\left[{}^{\mathcal{V}_x}\mathcal{G}^{\mathcal{V}}\right](p)^{\mathcal{V}}, \tag{17}$$

where

$$F_x = -\int_{\mathcal{S}_{\vec{x}x}} \rho u\boldsymbol{u}\cdot\boldsymbol{n}\,\mathrm{d}\mathcal{S}_{\vec{x}x} - \int_{\mathcal{S}_{\vec{x}x}} (\boldsymbol{\tau}\cdot\boldsymbol{i})\cdot\boldsymbol{n}\,\mathrm{d}\mathcal{S}_{\vec{x}x}. \tag{18}$$

The discrete pressure gradient in Eq. (17) will produce an x-staggered volumetric pressure field given a non-staggered volumetric pressure. In general, a gradient operator can be implemented to produce fields at arbitrary locations. Alternatively, nestedta interpolants may be used to produce the desired field type. The last term in Eq. (17) requires interpolation of the density to x-staggered volumes, hence the interoplant operator.

A. Discretization of F_x

Starting with the momentum convective flux in Eq. (18), we have,

$$\int_{\mathcal{S}_{\vec{x}x}} \rho u \boldsymbol{u} \cdot \boldsymbol{n}\, \mathrm{d}\mathcal{S}_{\vec{x}x} = \int_{\mathcal{S}_{xx}} \rho u u \mathrm{d}\mathcal{S}_{xx} + \int_{\mathcal{S}_{yx}} \rho u v\, \mathrm{d}\mathcal{S}_{yx} + \int_{\mathcal{S}_{zx}} \rho u w\, \mathrm{d}\mathcal{S}_{zx}. \tag{19}$$

This requires two interpolations, namely, momentum and velocity. The momentum $\rho u \equiv (\rho u)^{\mathcal{V}_x}$ must be interpolated to x-staggered x, y, z-faces, respectively. This is accomplished by using an appropriate volume-to-surface interpolant such as

$$(\rho u)^{\mathcal{S}_{xx}} = \left[{}^{\mathcal{S}_{xx}}\mathcal{R}^{\mathcal{V}_x}\right](\rho u)\,; \qquad (\rho u)^{\mathcal{S}_{yx}} = \left[{}^{\mathcal{S}_{yx}}\mathcal{R}^{\mathcal{V}_x}\right](\rho u)\,; \qquad (\rho u)^{\mathcal{S}_{zx}} = \left[{}^{\mathcal{S}_{zx}}\mathcal{R}^{\mathcal{V}_x}\right](\rho u)\,. \tag{20}$$

Next, the velocity components should be interpolated to the corresponding staggered faces. These are first calculated by interpolating the density to staggered cell centers using the following

$$u^{\mathcal{V}_x} = (\rho u)^{\mathcal{V}_x} / \left[{}^{\mathcal{V}_x}\mathcal{R}^{\mathcal{V}}\right](\rho); \qquad v^{\mathcal{V}_y} = (\rho v)^{\mathcal{V}_y} / \left[{}^{\mathcal{V}_y}\mathcal{R}^{\mathcal{V}}\right](\rho); \qquad w^{\mathcal{V}_z} = (\rho w)^{\mathcal{V}_z} / \left[{}^{\mathcal{V}_z}\mathcal{R}^{\mathcal{V}}\right](\rho), \tag{21}$$

The next step involves interpolating the staggered velocities to the staggered faces. This is accomplished by using an appropriate interpolant

$$u^{\mathcal{S}_{xx}} = \left[{}^{\mathcal{S}_{xx}}\mathcal{R}^{\mathcal{V}_x}\right] u^{\mathcal{V}_x}; \qquad v^{\mathcal{S}_{yx}} = \left[{}^{\mathcal{S}_{yx}}\mathcal{R}^{\mathcal{V}_y}\right] v^{\mathcal{V}_y}; \qquad w^{\mathcal{S}_{zx}} = \left[{}^{\mathcal{S}_{zx}}\mathcal{R}^{\mathcal{V}_z}\right] w^{\mathcal{V}_z}. \tag{22}$$

Note that for uniform structured grids, the staggered volume fields coincide with the faces of the non-staggered cells. Finally, the integral of the momentum convective flux is at hand

$$\left[{}^{\mathcal{V}_x}\mathcal{I}^{\mathcal{S}_{xx}}\right](\rho u)^{\mathcal{S}_{xx}}\, u^{\mathcal{S}_{xx}} + \left[{}^{\mathcal{V}_x}\mathcal{I}^{\mathcal{S}_{yx}}\right](\rho u)^{\mathcal{S}_{yx}}\, v^{\mathcal{S}_{yx}} + \left[{}^{\mathcal{V}_x}\mathcal{I}^{\mathcal{S}_{zx}}\right](\rho u)^{\mathcal{S}_{zx}}\, w^{\mathcal{S}_{zx}}. \tag{23}$$

The stress vector for the x-momentum equation is conveniently written as

$$\boldsymbol{\tau} \cdot \boldsymbol{i} = \tau_{ix} = \begin{pmatrix} \tau_{xx} & \tau_{yx} & \tau_{zx} \end{pmatrix} = \begin{pmatrix} -2\mu\frac{\partial u}{\partial x} + \frac{2}{3}\mu\nabla\cdot\boldsymbol{u} & -\mu\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right) & -\mu\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right) \end{pmatrix}. \tag{24}$$

Starting with τ_{xx}, we first recognize that τ_{xx} is an $\mathcal{S}_{xx}$ field, i.e. it resides on an x-face of an x-volume field. Note that all terms in the expression for τ_{xx} must be located at $\mathcal{S}_{xx}$. Then, we have

$$\tau_{xx} = \left[{}^{\mathcal{S}_{xx}}\mathcal{R}^{\mathcal{V}}\right](\mu)\left(-2\left[{}^{\mathcal{S}_{xx}}\mathcal{G}^{\mathcal{V}_x}\right](u) + \frac{2}{3}\left[{}^{\mathcal{S}_{xx}}\mathcal{R}^{\mathcal{V}}\right](\nabla\cdot\boldsymbol{u})\right). \tag{25}$$

In a similar fashion, τ_{yx} and τ_{zx} can be written as (note that τ_{yx} and τ_{zx} are $\mathcal{S}_{yx}$ and $\mathcal{S}_{zx}$ fields, respectively)

$$\tau_{yx} = -\mu\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right) = -\left[{}^{\mathcal{S}_{yx}}\mathcal{R}^{\mathcal{V}}\right](\mu)\left(\left[{}^{\mathcal{S}_{yx}}\mathcal{G}^{\mathcal{V}_y}\right](v) + \left[{}^{\mathcal{S}_{yx}}\mathcal{G}^{\mathcal{V}_x}\right](u)\right), \tag{26}$$

$$\tau_{zx} = -\mu\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right) = -\left[{}^{\mathcal{S}_{zx}}\mathcal{R}^{\mathcal{V}}\right](\mu)\left(\left[{}^{\mathcal{S}_{zx}}\mathcal{G}^{\mathcal{V}_z}\right](w) + \left[{}^{\mathcal{S}_{zx}}\mathcal{G}^{\mathcal{V}_x}\right](u)\right). \tag{27}$$

By combining these equations, one has the complete operator specification of the momentum equation. This method may be repeated for any type of differential equation and provides a convenient means for designing a computational code. Furthermore, writing equations using operators helps programmers focus on the physics of the problem at hand.

IV. Software Implementation

Implementation of the strategies proposed above requires the use of modern programming languages. Specifically, we choose C++ because it supports both runtime and compile-time polymorphism.[9–11] Our implementation takes form in a publicly available library called SpatialOps[‡]. The SpatialOps library provides a convenient interface for discrete operators using C++ template programming. The use of template programming is prompted by the need to

[‡]SpatialOps may be downloaded from: https://software.crsim.utah.edu/trac/wiki/SpatialOps.

embed arbitrary type information in the operator classes directly. Embedding type information provides the necessary support for operators to identify the types of fields that they operate on and produce. At the outset, one is able to enforce compatibility between fields and operators at compile time, resulting in highly robust code.

The SpatialOps library makes the following assumptions about the mesh it is being used with. First of all, Operators and fields are defined on a patch. A patch is a region of space where the governing equations are solved. Unlike a full mesh, several patches may be associated with a single mesh. This is ackin to multiple processors calculating a given problem. We also assume that a patch has ghost cells associated with it. Ghost cells are additional cells that allow implementation of boundary conditions and facilitate domain decomposition parallelization. Finally, fields and operators are associated with a linear algebra package which provides matrix-vector and matrix-matrix operations.

To represent a field and an operator, the SpatialOps library provides two key base classes: the SpatialField class and the SpatialOperator class. As the names designate, these base classes provide the necessary interface for all derived and specialized operators and fields.

A. SpatialField Class

A SpatialField holds a view of memory that is allocated elsewhere (e.g. by a framework). Alternatively, it can allocate memory and manage it internally. This flexibility allows a SpatialField to interface to existing frameworks that provide memory management by simply wrapping existing memory, thereby avoiding unnecessary memory copies. The declaration of the SpatialField class is given in Listing 1.

The unique set of template parameters in the SpatialField class implies a distinct field type. These template parameters represent:

- `LinAlg` : the type of linear algebra system used for sparse matrix-vector multiplications. This allows lightweight interfaces to be built to various linear algebra libraries and the SpatialField objects can be interoperable with a wide range of such libraries.
- `FieldLocation` : the storage location (e.g. volume field) and other type traits for the field.
- `GhostTraits` : information about the ghost structure on the field. Ghost fields are frequently used in assigning boundary conditions as well as in domain decomposition and MPI parallelism.
- `T` : the fundamental type for this field. By default this is a double, but it could be changed to allow for more sophisticated types that allow automatic differentiation[12,13] for example.

Additional details about the template parameters and the interfaces that they must implement is documented in the source code.

For convenience, the SpatialField class overloads the four unary operations of addition (+=), subtraction (-=), multiplication (*=), and divistion (/=). These facilitate elementary operations among fields. Combined with expression template technology, these basic operators allow for simple implementation of complex mathematical expressions involving multiple SpatialField objects. Listing 2 shows a generic implementation for a *= operator using STL-compliant iterators.[14]

B. SpatialOperator Class

An *operator* is required to transform between field types. This imposes strong type safety that allows for compile-time error checking. A SpatialOperator is represented as a sparse matrix, and requires a linear algebra package to support matrix-matrix and matrix-vector operations. A given operator is constructed on all patches of the mesh. In

```
template< typename LinAlg,        // The linear algebra package used
          typename FieldLocation, // The location of this field
          typename GhostTraits,   // The ghost strategy for this field
          typename T=double >     // underlying type for a point in this field
class SpatialField;
```

Listing 1. Declaration of a SpatialField using C++ templates.

```
template< typename VecOps,      typename Location,
          typename GhostTraits, typename T >
operator += ( const MyType& other )
{
  const_iterator iother=other.begin();
  iterator ifld=this->begin();
  const iterator iend=this->end();
  for( ; ifld!=iend; ++ifld, ++iother ){
    *ifld += *iother;
  }
  return *this;
}
```

Listing 2. Generic implementation of operator *= for a SpatialField.

```
template< typename LinAlg,      // linear algebra support for this operator
          typename OpType,      // type of operator
          typename SrcFieldT,   // information on the source field
          typename DestFieldT > // information on the dest field
class SpatialOperator;
```

Listing 3. Declaration of the SpatialOperator base class.

the case of a uniform, structured mesh, this may be accomplished very easily. In the case of an unstructured mesh, topology and connectivity information is required to determine the sparsity pattern and coefficients for each operator. It is important to note that a unique operator type is determined by the operation it serves (e.g. computing a gradient) and the type of mesh it is constructed for. Note that a specific type of operator implies specific field types as well. For example, a divergence operator implies that it acts on a surface field to produce a volume field. Verification can be performed for each operator using analytic functions to ensure proper order of accuracy. Note that this can be performed entirely independent from verification of the application code itself. The declaration of a SpatialOperator is given in Listing 3.

Since each SpatialOperator represents a sparse matrix , application of a SpatialOperator to a SpatialField involves a matrix-vector multiplication. This can be made to take full advantage of system hardware via platform-optimized BLAS and LAPACK libraries. Furthermore, since the SpatialOperator and SpatialField types are determined by ghosting patterns, linear algebra package, and field locations, this provides compile time compatibility checking. In other words, if an operator is inconsistent with the fields that one tries to operate on, the code will not compile.

Every derived operator must provide the the operator type, the source field type, and the destination field type. Furthermore, it must define a method that applies the operator to the field. These are summarized in Table 2 and Table 3 .

V. Example

By way of example, we provide the code for constructing operators for a diffusive flux expression. Calculation of the diffusive flux of a scalar φ requires evaluation of

$$\boldsymbol{J}_\varphi = -D_\varphi \nabla\varphi. \tag{28}$$

Note that $\boldsymbol{J}_\varphi$, D_φ, and $\nabla\varphi$ are surface fields while φ is a volume field. Listing 4 shows the complete implementation of this calculation.

This listing assumes that the types of the gradient, surface, and volume fields are known at compile time. The diffusion coefficient is also assumed to be known at surface locations. Note the absence of any topology-specific code. There are no loops over nodes to calculate the gradient. This is performed as a sparse matrix-vector multiply within the SpatialOperator (a gradient operator in this case).

The template parameters on line 1 of Listing 4 define the types for the gradient operator as well as the fields. The C++ compiler will automatically generate the appropriate code for each specific type that is provided. There-

Required typedef	Description
`OpType`	Operator type
`SrcFieldT`	Source field type
`DestFieldType`	Destination field type

Table 2. Summary of required types for a SpatialOperator class.

Methods	Description
`apply_to_field`	Applies operator to a source field
`apply_to_op`	Applies operator to another operator

Table 3. Summary of methods provided by a SpatialOperator Class.

fore, no distinction is required between surface fields in structured, unstructured, staggered, or collocated schemes. Furthermore, the gradient operator, represented as a sparse matrix, has a specific type that will be known at compile time. In an actual implementation, the last two template parameters would be eliminated and the field types would be determined from the gradient operator, since the field types are redundant information that is contained in the operator type.

```
template<typename GradOp, typename SurfField, typename VolField>
void generic_diffusive_flux( const GradOp& grad,
                             const VolField& phi,
                             const SurfField& diffCoeff,
                             SurfField& diffFlux )
{
  grad.apply_to_field ( phi, diffFlux ); // calculate the gradient of phi
  diffFlux *= diffCoeff;
  diffFlux *= -1.0;
}
```

Listing 4. Implementation of a diffusive flux.

Perhaps the most notable advantage of the code shown in Listing 4 is that the structure of the calculation is separated from the data layout. For example, the same piece of code could be used in situations where $\nabla\varphi$ was stored as

- $\nabla\phi = \left[\begin{array}{ccccccccc} \left.\frac{\partial\varphi}{\partial x}\right|_1 & \cdots & \left.\frac{\partial\varphi}{\partial x}\right|_n & \left.\frac{\partial\varphi}{\partial y}\right|_1 & \cdots & \left.\frac{\partial\varphi}{\partial y}\right|_n & \left.\frac{\partial\varphi}{\partial z}\right|_1 & \cdots & \left.\frac{\partial\varphi}{\partial z}\right|_n \end{array}\right]$,
- $\nabla\phi = \left[\begin{array}{ccccccc} \left.\frac{\partial\varphi}{\partial x}\right|_1 & \left.\frac{\partial\varphi}{\partial y}\right|_1 & \left.\frac{\partial\varphi}{\partial z}\right|_1 & \cdots & \left.\frac{\partial\varphi}{\partial x}\right|_n & \left.\frac{\partial\varphi}{\partial y}\right|_n & \left.\frac{\partial\varphi}{\partial z}\right|_n \end{array}\right]$,
- or it could be defined to produce vector components as $\left[\begin{array}{cccc} \left.\frac{\partial\varphi}{\partial x}\right|_1 & \left.\frac{\partial\varphi}{\partial x}\right|_2 & \cdots & \left.\frac{\partial\varphi}{\partial x}\right|_n \end{array}\right]$ or $\left[\begin{array}{cccc} \left.\frac{\partial\varphi}{\partial y}\right|_1 & \left.\frac{\partial\varphi}{\partial y}\right|_2 & \cdots & \left.\frac{\partial\varphi}{\partial y}\right|_n \end{array}\right]$.

Each of these cases would be distinguished by the definition of the discrete gradient operator as well as the vector fields. However, the same code shown in Listing 4 would work in all cases. Furthermore, if a field were supplied that was incompatible with a given operator, this error would be caught at compile time. This improves robustness of code and helps reduce implementation errors.

VI. Conclusions

In this manuscript, we present a novel paradigm for separating the discretization component from the physics in computational codes. Our model allows application programmers to write code that is independent of the type of mesh being used. This allows large amounts of software to be directly reused among structured, unstructured, colocated, and staggered finite volume approaches. This is accomplished by providing an abstraction of the operators in differential equations. The operator abstraction is possible because the nature of the discretization process is equivalent to a matrix-vector multiplication. Spatial operators therefore encapsulate the discretization coefficients as a sparse matrix. The sparsity of the matrix depends on the type of grid and the order of accuracy being used. At the outset, the discretization process is represented by the application of an operator to a solution variable, resulting in a matrix-vector multiplication.

The discrete nature of these operators requires precise specification of field types they consume and provides an interface for a variety of numerical evaluations. Fields represent any quantity that is required by the solution process and are located at specific points on the mesh. This entails several consistency requirements hence making the code more robust and less prone to errors. The theoretical foundation for this approach is presented along with the proper mathematical nomenclature that allows for easy identification of the required operators and feilds.

Our implementation of the abstract operators approach takes form in a newly developed library called SpatialOps. The library provides a C++ interface to defining fields and operators and uses template programming as its foundation. Its proper operation necessitates a linear algebra library such as UBLAS or TRILLINOS as well as mesh framework such as SAMRAI[15], SIERRA[16], and UINTAH.[17] The library also provides the base classes for deriving a variety of operators with differnt source and destination field types.

This discrete operators approach to computational software design completes the link between physics, numerics, and spatial discretization. The benefits of implementing this framework are numerous, the most important of which is that it provides applications programmers with an avenue to focus on writing robust code that reflects physical models.

Acknowledgments

This research is sponsored by the National Nuclear Security Administration under the Advanced Simulation and Computing program through DOE Research Grant #DE-NA0000740 and by the National Science Foundation through Grant No. PetaApps-0904631.

References

[1]Richardson, L. F., "The Approximate Arithmetical Solution by Finite Differences of Physical Problems Involving Differential Equations, with an Application to the Stresses in a Masonry Dam," *Philosophical Transactions of the Royal Society of London A*, Vol. 210, 1911, pp. 307–357.

[2]Harlow, F. H. and Welch, J. E., "Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface," *Physics of Fluids*, Vol. 8, No. 12, 1965, pp. 2182.

[3]Perot, B., "Conservation Properties of Unstructured Staggered Mesh Schemes," *Journal of Computational Physics*, Vol. 159, No. 1, March 2000, pp. 58–89.

[4]Lilly, D. K., "On the Computational Stability of Numerical Solutions of Time-Dependent Non-Linear Geophysical Fluid Dynamics Problems," *Monthly Weather Review*, Jan. 1965.

[5]Mittal, R. and Moin, P., "Suitability of Upwind-Biased Finite Difference Schemes for Large-Eddy Simulation of Turbulent Flows," *AIAA Journal*, Vol. 35, No. 8, 1997, pp. 1415–1417.

[6]Patankar, S., *Numerical heat transfer and fluid flow*, Hemisphere Pub, 1980.

[7]Najm, H. N. and Knio, O. M., "Modeling Low Mach Number Reacting Flow with Detailed Chemistry and Transport," *Journal of Scientific Computing*, Vol. 25, No. 1, 2005, pp. 263–287.

[8]Almgren, A. S., Bell, J. B., Rendleman, C. A., and Zingale, M., "Low Mach Number Modeling of Type Ia Supernovae. I. Hydrodynamics," *The Astrophysical Journal*, Vol. 637, No. 2, 2006, pp. 922–936.

[9]Stroustrup, B., *The C++ programming language*, Addison-Wesley, 1997.

[10]Alexandrescu, A., *Modern C++ design*, Vol. 98, Addison-Wesley Reading, MA, 2001.

[11]Vandevoorde, D. and Josuttis, N., *C++ templates: the Complete Guide*, Addison-Wesley Professional, 2003.

[12]Aubert, P., Césaré, N. D., and Pironneau, O., "Automatic differentiation in C++ using expression templates and. application to a flow control problem," *Computing and Visualization in Science*, Vol. 3, No. 4, 2001, pp. 197–208.

[13]Bartlett A, R., Gay, D. M., and Phipps, E. T., "Automatic Differentiation of C++ Codes for Large-Scale Scientific Computing," *Computational Science – ICCS 2006*, edited by V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Vol. 3994 of *Lecture Notes in Computer Science*, Springer, Heidelberg, 2006, pp. 525–532.

[14]Musser, D. R., Derge, G. J., and Saini, A., *C++ Programming with the Standard Template Library*, Addison–Wesley, 2001.

[15]Hornung, R. and Kohn, S., "Managing application complexity in the SAMRAI object-oriented framework," *Concurrency and Computation: Practice and Experience*, Vol. 14, No. 5, 2002, pp. 347–368.

[16]Stewart, J. and Edwards, H., "A framework approach for developing parallel adaptive multiphysics applications," *Finite Elements in Analysis and Design*, Vol. 40, No. 12, 2004, pp. 1599–1617.

[17]de St. Germain, J., McCorquodale, J., Parker, S., and Johnson, C., "Uintah: A Massively Parallel Problem Solving Environment," *Ninth IEEE International Symposium on High Performance and Distributed Computing*, IEEE, November 2000, pp. 33–41.