A GPU Accelerated Discontinuous Galerkin Conservative Level Set Scheme for Simulating Atomization

by

Zechariah J. Jibben

A Research Prospectus Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Submitted for Approval November, 2013 to the
Graduate Supervisory Committee:

Marcus Herrmann, Chair
Kyle Squires
Ronald Adrian
Kangping Chen
Michael Treacy

Arizona State University

November, 2013

Abstract

This research prospectus presents a process for interface capturing via an arbitrary-order, nearly quadrature free, discontinuous Galerkin (DG) scheme for the conservative level set (CLS) method (Olsson et al., 2005, 2008). The DG numerical method is utilized on both advection and reinitialization, and executed on a refined level set grid (RLSG) (Herrmann, 2008) for effective use of processing power. Computation is performed on a massively parallel scale utilizing CPU and GPU architectures to make the method feasible at high order. Finally, a sparse data structure is implemented to take full advantage of parallelism on the GPU, where performance relies on well-managed memory operations.

Following the accurate CLS (ACLS) method (Desjardins, 2008), normal vectors necessary for reinitialization are calculated from the level set scalar in the vicinity of the interface alone, thereby avoiding variations resulting from noise away from the interface. To accomplish this in an arbitrary order DG scheme, an arbitrary order DG fast sweeping method (FSM) is developed to solve the Eikonal equation. The resulting signed distance function, which is independent of local fluctuations in the level set scalar, is then differentiated on a shared three-cell basis to construct the normal vector field.

With solution variables projected into a k^{th} order polynomial basis, a $k+1$ order convergence rate is found for both advection and reinitialization tests using the method of manufactured solutions. Accelerating advection via GPU hardware is found to provide a near 60x speedup factor comparing a 1.9GHz AMD Opteron 6186 CPU in serial vs. a Nvidia Tesla C2050 GPU, with higher speedup factors for higher degree polynomials. Arbitrarily high convergence rates combined with speedup factors that increase with polynomial degree motivate the development and use of a GPU accelerated, arbitrary order DG method.

Future work involves further development of the DG fast sweeping method and accelerating reinitialization via the GPU. This approach is intended to be coupled to a Navier-Stokes' flow solver and executed in parallel on heterogeneous architectures.

Acknowledgements

This work was supported by the National Science Foundation grant CBET-1054272. Computations were performed using the ASU Advanced Computing Center Saguaro cluster and the Los Alamos National Laboratory Moonlight ASC and Darwin CCS-7 clusters.

Table of Contents

List of Figures

Figure Page

List of Tables

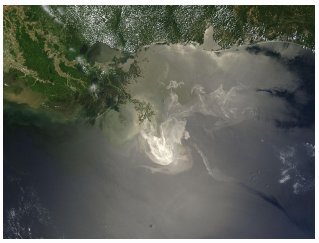
Chapter 1

Introduction

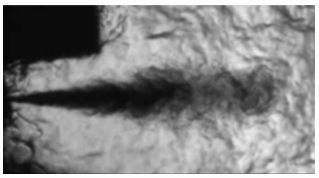
1.1 Motivation

A pressing problem in engineering is the modeling of fluid interactions involving immiscible interfaces. These flows occur in a variety of natural phenomena and technical applications, for instance biological systems, lava flow, oil leaks, and medical sprays. Inside jet turbines, internal combustion engines, and liquid rocket engines, fuel is dispersed and atomized into air. For complete combustion of the fuel, the fuel must be evaporated into the surrounding air. This task is aided through atomization, which is performed by fuel injection. The quality of the mixture resulting from fuel injection and liquid atomization therefore has a direct impact on the overall performance of the engine, as well as pollutant production. Unfortunately, experiments are difficult or impossible to perform at operating conditions, simply because optical access is obstructed by the engine structure and a haze of fuel droplets surrounds interesting structures. Furthermore, there is no known way of solving the system of nonlinear governing equations analytically. On the other hand, numerical methods can be utilized to produce approximate solutions to these equations and describe the flow and have become increasingly accurate and capable as computational power has become more available. As a result, computational tools and

(a) Pahoeoe fountain [11]



(b) Deepwater Horizon oil spill [19]



(c) Diesel jet [28]

Figure 1.1: Multiphase flow examples

simulations that predict multiphase flows are vital to a multitude of engineering applications and our understanding of the physics.

Current state of the art simulations often rely on experimental data to describe atomization in a statistical sense, thereby relying on tuning parameters to produce accurate results. As previously mentioned, experiments are difficult or even impossible to perform in most operating conditions, opening the inference demanded by this approach to doubt. Therefore, developing a simulation which solves the governing nonlinear partial differential equations directly is essential.

Numerical simulation of the Navier-Stokes equations with appropriate interfacial modeling has been the studied for several decades now. Interface models are discussed in more detail in Ch. 2. For now, it is sufficient to say that the primary issues are mass conservation and surface tension calculation, as many methods suffer from either producing non-negligible mass errors or being limited to low order. The accurate conservative level set (ACLS) method [7] is used to allow high order calculations of surface tension while also offering a much improved mass conservation.

The governing equations have been solved by a variety of numerical approaches as well, with finite difference, finite volume, and spectral methods being the most popular. To produce accurate predictions, a high order solvers are essential. Although they are more computationally expensive on the same mesh size compared to low order methods, they converge to the correct solution on far coarser meshes. Unfortunately, all of the above methods only achieve higher orders by increasing the size of the stencil. This presents a challenge and drawback for computing on a massively parallel scale, since it demands more CPU time be spent on communication. Often the overhead cost of interprocessor communication is overwhelming compared to the cost of arithmetic operations immediately relevant to solving the equations. In recent years, discontinuous Galerkin (DG) methods have become increasingly popular because they allow high order conver-

gence rates while keeping a small stencil, only dependent on immediate neighbors. DG methods can, however, be quite computationally expensive. In many cases, their high order implementations are prohibitively expensive since each solution variable update requires a high number of floating point operations. Fortunately, this drawback makes DG methods almost tailor-made for GPU architectures, which are ideal for performing many numerical operations with comparatively little memory transfer. Therefore, GPU hardware is leveraged to mitigate the cost of high order DG methods.

This research prospectus details the methods and algorithms developed for a predictive numerical laboratory for fluid flow fields involving immiscible interfaces. This includes descriptions of ACLS, an arbitrary order DG scheme, and implementations on both central processing units (CPUs) and graphics processing units (GPUs). Following this description, a series of verification results are provided, where the method of manufactured solutions (MMS) and several simple analytical solutions are compared to simulation outputs.

1.2 Governing Equations

Herrmann [12] gives a good overview of the governing equations of a fluid interaction involving immiscible interfaces. These are the Navier-Stokes' equations, along with a surface tension term $\boldsymbol{T}_\sigma$ that is nonzero only at the interface location $\boldsymbol{x}_f$.

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} = -\frac{1}{\rho}\nabla p + \frac{1}{\rho}\nabla \cdot \left(\mu \left(\nabla \boldsymbol{u} + \nabla^{\mathrm{T}} \boldsymbol{u}\right)\right) + \boldsymbol{g} + \frac{1}{\rho}\boldsymbol{T}_\sigma \tag{1.1}$$

$$\boldsymbol{T}_\sigma(\boldsymbol{x}) = \sigma\kappa\delta\left(\boldsymbol{x} - \boldsymbol{x}_f\right)\hat{\boldsymbol{n}} \tag{1.2}$$

where the continuity equation for incompressible flow is

$$\nabla \cdot \boldsymbol{u} = 0. \tag{1.3}$$

Here, $\boldsymbol{u}$ is the velocity, ρ is density, p is pressure, μ is dynamic viscosity, $\boldsymbol{g}$ is the gravitational body force, σ is the surface tension constant, κ is the local surface curvature, and $\hat{\boldsymbol{n}}$ is the local interface normal. As previously mentioned, accurate calculation of the surface

tension requires a good method for capturing the interface location and computing the curvature and normals at high order.

1.3 Notation

Throughout this paper, Einstein notation is used with Latin indices to imply summation. It is convenient, however, to use Greek indices in situations where implied summation is not desired. For example,

$$a_i b_i = \sum_i a_i b_i$$

$$a_\alpha b_\alpha \neq \sum_\alpha a_\alpha b_\alpha.$$

Chapter 2

The Accurate Conservative Level Set Method

There are several approaches to modeling interface topology evolution. Interface tracking is common in systems involving solids, while interface capturing is more popular in fluid systems. Of capturing methods, volume of fluid methods (VOF) and level set methods are the most common. The VOF approach has the benefit of discretely conserving mass, while traditional level sets do not share this property. On the other hand, level sets have the benefit of high order accuracy, normals, and curvature.

2.1 Traditional Level Set Method

The concept of level sets is to model the fluid interface, shown in Fig. 3.1, as an isosurface of some scalar function. The traditional level set method, described well by [7, 21, 23], transports the interface via the advection equation. The advection equation simply states that since the interface, transported by the local flow velocity, remains along the $\phi = \text{const.}$ contour by definition, the material derivative is set equal to zero.

$$\frac{\partial \phi}{\partial t} + \boldsymbol{u} \cdot \nabla \phi = 0 \tag{2.1}$$

A popular choice for the level set scalar, originally suggested by Chopp [4], is the signed distance function, which satisfies the Eikonal equation $|\nabla \phi| = 1$ and gives $|\phi(\boldsymbol{x}, t)| = \left|\boldsymbol{x} - \boldsymbol{x}_f\right|$. The contour $\phi = 0$ implicitly defines the location of the phase interface, and the scalar is positive in one phase and negative in the other. This choice features smooth gradients that allow the curvature to be easily calculated. The advection equation, however, will not maintain the signed distance function form of the level set scalar. In fact, without being treated, ϕ will become increasingly distorted over time with sharper and sharper gradients. This makes the surface tension quite difficult to calculate over time, and results in mass losses/gains. For numerical accuracy, and much improved mass conservation, it is necessary to periodically reinitialize ϕ. Several approaches to

reinitialization exist, many of which involve a "brute-force" strategy. These approaches are often either computationally expensive or move the interface front, however. A PDE approach suggested by Sussman et al. [29] is the most common, which involves solving a Hamilton-Jacobi equation that converges when the Eikonal equation is satisfied, thereby restoring the signed distance function form of the level set scalar.

$$\frac{\partial \phi}{\partial \tau} + S\left(|\nabla \phi| - 1\right) = 0 \tag{2.2}$$

Here, S is a sign function that approaches zero as $\phi \to 0$, so that in the theoretical limit mass is conserved. However, the discretized form of the equation does not conserve mass, and in many cases worsens the mass errors. As a result, the scope of engineering applications where the traditional level set maintains physical accuracy is limited.

2.2 The Conservative Level Set Method

Specifically to improve mass conservation while maintaining a smooth level set scalar, a new formulation of the level set method was proposed by Olsson et al. [21, 22] where the level set scalar $G(\boldsymbol{x}, t)$ is treated as a conserved variable. This is done by rewriting the advection equation,

$$\frac{\partial G}{\partial t} + \boldsymbol{u} \cdot \nabla G = 0,$$

in conservative form as

$$\frac{\partial G}{\partial t} + \nabla \cdot (G\boldsymbol{u}) = 0 \tag{2.3}$$

using the solenoidal property of incompressible velocity fields, Eq. (1.3).

The interface is modeled as a 0.5-isosurface of the level set scalar $G(\boldsymbol{x}, t)$. Then, $G > 0.5$ on one side of the interface and $G < 0.5$ on the other. The level set scalar in the form of a Heaviside function, jumping discontinuously from 0 to 1 across the interface, would conserve mass exactly. To avoid numerical errors, however, a smeared out Heaviside

function is used. Olsson et al. suggest a hyperbolic tangent profile to accomplish this.

$$G(\boldsymbol{x}, t) = \frac{1}{2}\left(\tanh\left(\frac{\phi(\boldsymbol{x}, t)}{2\varepsilon}\right) + 1\right) \tag{2.4}$$

Here, $\phi(\boldsymbol{x}, t)$ is the signed distance function to the interface. The profile thickness is proportional to ε, which is set equal to half the cell width Δx. The reason for this choice, as opposed to a polynomial or piecewise function, is that a conservative reinitialization equation which restores the level set scalar to this form exists. As always, the advection equation distorts G, in this case dissipating the scalar. Reinitialization sharpens the level set scalar, thereby maintaining the initial profile. Olsson et al. [22] suggests a conservative PDE which invokes a compression term along the direction normal to the interface, and a diffusive term in that same direction.

$$\frac{\partial G}{\partial \tau} + \nabla \cdot (G\,(1 - G)\,\hat{\boldsymbol{n}}) = \nabla \cdot (\varepsilon\,(\nabla G \cdot \hat{\boldsymbol{n}})\,\hat{\boldsymbol{n}}) \tag{2.5}$$

Olsson computes the normal vector,

$$\hat{\boldsymbol{n}} = \frac{\nabla G}{|\nabla G|} = \frac{\nabla \phi}{|\nabla \phi|} \tag{2.6}$$

from either the level set scalar or a signed distance function. [7] suggests the accurate conservative level set method (ACLS) which instead proposes that the normal vector ought not constructed from the local G field. Rather, the normals are constructed from the level set scalar only in the vicinity of the interface. With this approach, oscillations or variations throughout the domain that do not cross the $G = 0.5$ threshold do not create new interfaces due to reinitialization (see Fig. 2.1). On the other hand, if the normal vector treats isolated variations as interfaces, reinitialization will attempt to enforce a hyperbolic tangent profile on numerical errors. To compute normals, then, Eq. (2.6) is evaluated given a signed distance function ϕ, which is computed from the level set scalar field G in the vicinity of the 0.5-isosurface. An arbitrary-order fast sweeping method is investigated for

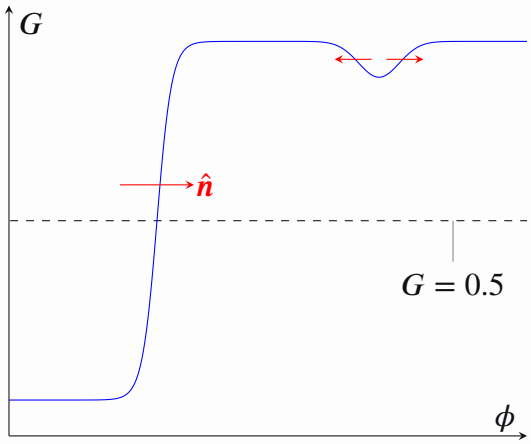


Figure 2.1: Hyperbolic tangent profile with flaw away from interface

determining ϕ throughout the computational domain, and the entire process is described in greater detail in Ch. 4.

The preceding reinitialization equation, then, constructs G as a conserved variable and enforces the hyperbolic tangent profile with thickness ε. Since the normal vector is dependent only on the interface, and not the local solution variables, it also does not allow spontaneous creation of mass. When needed, it is to be evaluated in pseudo-time, τ, until convergence.

Eqs. (2.3) and (2.5), together with determining the interface normal from the interface vicinity alone, represent the system of equations to be solved and our formulation of the CLS method. Combining this approach with an arbitrary-order Runge-Kutta (RK) discontinuous Galerkin (DG) method further improves the accuracy and mass conservation of level set methods.

2.3 The Refined Level Set Grid

To improve the efficiency of the simulation, the refined level set grid proposed by Herrmann [12] is implemented. The level set equations are solved on a Cartesian mesh separate from the flow solver, and cells are organized into blocks of a predefined size. To reduce the necessary computational work the equations are only solved in blocks within

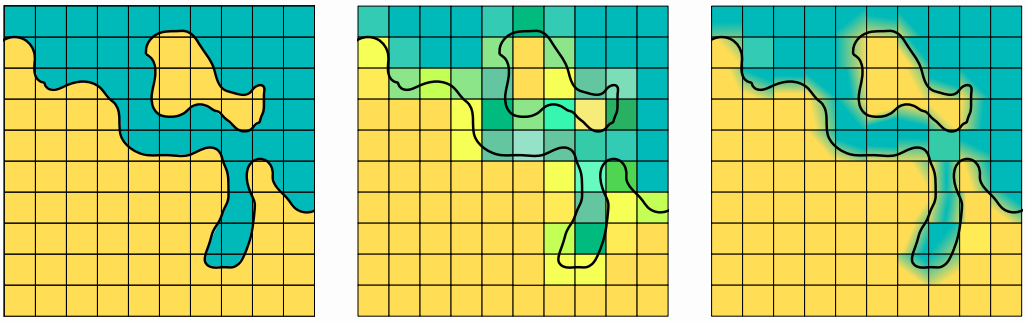
a certain distance of the interface. In fact, advection and reinitialization only need to be solved inside a region surrounding the interface itself, called the $\mathscr{T}$-band, which does not need to fill entire blocks. Four important bands are generated: a) the $\mathscr{A}$-band, consisting only of cells which contain the interface, b) the $\mathscr{T}$-band, in which advection and reinitialization are solved, c) the $\mathscr{W}$-band, which contains all ghost cells immediately neighboring the $\mathscr{T}$-band, and d) the $\mathscr{X}$-band, which fills the entire domain. For details on band generation and parallelization of the refined level set grid, see [12].

Chapter 3

The Discontinuous Galerkin Method

The discontinuous Galerkin method is motivated by arbitrarily high convergence rates that can be achieved with a small stencil, containing only immediate neighbors. DG accomplishes this by allowing sub-cell variation and storing information about derivatives locally in the form of basis function coefficients. LeSaint and Raviart [13] proved that this method can formally achieve a $k+1$ order convergence rate with k^{th} degree polynomials, while Cockburn and Shu [5] found this to also be achievable for nonlinear problems in practice. This section describes the scheme construction.

3.1 Spatial Discretization



(a) Fluid Interface (b) Finite Volume (c) Discontinuous Galerkin

Figure 3.1: Interface Discretization

Originally developed by Reed and Hill [24] at Los Alamos National Laboratory, the discontinuous Galerkin numerical method can be thought of as a generalization of the finite volume (FV) method. As with FV methods, the physical domain Ω is discretized into cells with domain $\mathcal{I}_\kappa$. For the purposes of this paper, the domain is mapped onto a Cartesian mesh.

$$\Omega = \bigcup_{\kappa=1}^{N_{\text{cells}}} \mathcal{I}_\kappa$$

The finite volume method then assigns an average value of your solution variable to each cell. On the other hand, the discontinuous Galerkin method allows sub-cell variation by performing a spectral decomposition of the solution variables within each cell. That is, $\boldsymbol{G}$, $\boldsymbol{u}$, and $\hat{\boldsymbol{n}}$ are projected into the basis $\{b_i\}$ as

$$\boldsymbol{f}(\boldsymbol{x},t) \;= \sum_{i=1}^{N_f} \boldsymbol{f}_i^{\kappa}(t)\, b_i(\boldsymbol{\xi})\,, \tag{3.1}$$

where the series is truncated at N_f (for the purposes of this paper, $N_g = N_u = N_n$ and $\Delta x = \Delta y = \Delta z$). In this sense, a finite volume method is equivalent to a discontinuous Galerkin method with $N_g = N_u = N_n = 1$. The coefficients are found by integrating with respect to sub-cell position variables,

$$\boldsymbol{f}_n^{\kappa} = \int_{\mathscr{K}} \boldsymbol{f}\left(\boldsymbol{x}_{\kappa} + \frac{1}{2}\Delta x_i \xi_i\right) b_n(\boldsymbol{\xi})\ \mathrm{d}V\,. \tag{3.2}$$

The position vector is mapped between sub-cell coordinates, $\boldsymbol{\xi} \in \mathscr{K} = [-1,1]^3$, and physical coordinates $\boldsymbol{x} \in \Omega$ using Eq. (3.3). This depends on the location of the cell center $\boldsymbol{x}_{\kappa}$ and cell dimensions $\boldsymbol{\Delta x}$, as shown in Fig. 3.2. The domain $\mathscr{K}$, with the surface domain $\partial\mathscr{K}$ and neighbors $\mathscr{K}^f$ are depicted in Fig. 3.3.

$$\begin{aligned} \mathscr{K} \to \Omega: \quad & x_{\alpha} = x_{\alpha,\kappa} + \xi_{\alpha}\frac{\Delta x_{\alpha}}{2} \\ \Omega \to \mathscr{K}: \quad & \xi_{\alpha} = \frac{x_{\alpha} - x_{\alpha,\kappa}}{\Delta x_{\alpha}/2} \end{aligned} \tag{3.3}$$

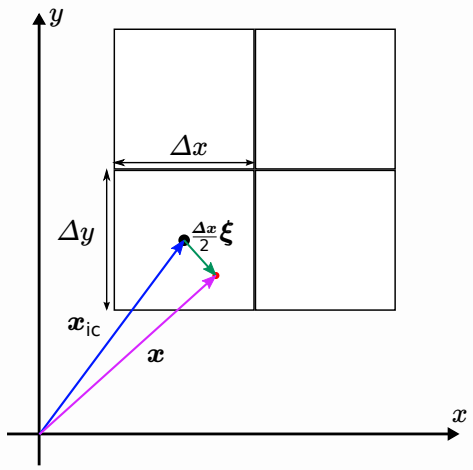


Figure 3.2: Position Vectors

The normalized Legendre polynomial basis is selected for their orthonormality property,

$$\int_{\mathscr{K}} b_i b_j \, \mathrm{d}V = \delta_{ij}, \tag{3.4}$$

and they are constructed by performing Gram-Schmidt orthonormalization on the space of 3D monomials $\xi^\alpha \eta^\beta \zeta^\gamma$. Then, for a maximum monomial degree k, the number of terms in the spectral expansion is $N_g = (k+1)^3$.

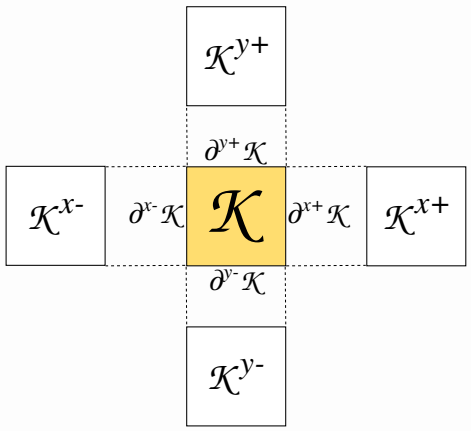


Figure 3.3: The local cell domain $\mathscr{K}$, surface $\partial\mathscr{K} = \bigcup_{f=\text{faces}} \partial^f \mathscr{K}$, and neighbors $\mathscr{K}^f$

These expansions are then substituted into Eq. (2.3) and Eq. (2.5). By writing deriva-

tives in terms of sub-cell coordinates via Eq. (3.3),

$$\frac{\partial}{\partial x_\alpha} = \frac{2}{\Delta x_\alpha}\frac{\partial}{\partial \xi_\alpha},$$

performing an inner product with b_n (integrate over the cell domain $\mathscr{K}$), taking advantage of orthonormality, and using the divergence theorem, we arrive at a system of coupled ordinary differential equations describing the time evolution the coefficients g_n for all cells. The RKDG method has been implemented in the context of the CLS method previously by Czajkowski and Desjardins [6]. However, their method held the velocity and normal vectors constant within a cell, only expanding the level set scalar to full order in the discontinuous basis. The result was a scheme limited to second order. Here, all variables are projected with full order into the DG basis, allowing for higher convergence rates.

$$\frac{\mathrm{d}g_n^\kappa}{\mathrm{d}t} = u_k^{\kappa,j} g_i^\kappa \frac{2}{\Delta x}\int_{\mathscr{K}} b_k b_i \frac{\partial b_n}{\partial \xi_j}\,\mathrm{d}V - \frac{2}{\Delta x}\oint_{\partial\mathscr{K}} \overline{\boldsymbol{G}\boldsymbol{u}} b_n \cdot \mathrm{d}\hat{\boldsymbol{S}} \tag{3.5}$$

$$\begin{aligned}\frac{\mathrm{d}g_n^\kappa}{\mathrm{d}\tau} &= n_k^{\kappa,j} g_i^\kappa \frac{2}{\Delta x}\int_{\mathscr{K}} b_k b_i \frac{\partial b_n}{\partial \xi_j}\,\mathrm{d}V - n_k^{\kappa,j} g_i^\kappa g_j^\kappa \frac{2}{\Delta x}\int_{\mathscr{K}} b_k b_i b_j \frac{\partial b_n}{\partial \xi_j}\,\mathrm{d}V \\ &\quad - \varepsilon g_i^\kappa n_k^{\kappa,a} n_l^{\kappa,d}\left(\frac{2}{\Delta x}\right)^2 \int_{\mathscr{K}} \frac{\mathrm{d}b_i}{\mathrm{d}\xi_a} b_k b_l \frac{\mathrm{d}b_n}{\mathrm{d}\xi_d}\,\mathrm{d}V \\ &\quad - \frac{2}{\Delta x}\oint_{\partial\mathscr{K}} \overline{G(1-G)}\hat{\boldsymbol{n}} b_n \cdot \mathrm{d}\hat{\boldsymbol{S}} + \varepsilon\left(\frac{2}{\Delta x}\right)^2 \oint_{\partial\mathscr{K}} \overline{\overline{\left(\nabla_\xi G \cdot \hat{\boldsymbol{n}}\right)\hat{\boldsymbol{n}}}} b_n \cdot \mathrm{d}\hat{\boldsymbol{S}}\end{aligned} \tag{3.6}$$

All that remains is to determine appropriate flux functions and a time stepping mechanism.

Note that all of the above volume integrals are written only in terms of the Legendre polynomial basis functions and the local cell domain. They can therefore be evaluated prior to running the simulation and tabulated. It is found that they form sparse arrays (discussed in Sec. 3.4), and therefore present an opportunity for considerable speed-up. A similar process is desired for the flux integrals, and is described in the following section.

3.2 Flux Handling

The surface integrals in Eq. (3.5) and Eq. (3.6), of course, are evaluated along the cell boundary. On this surface, however, exists a discontinuous jump for all variables described by the DG expansion. This raises an important question in computational mathematics that is handled by a multitude of approaches: which solution is used as the integrand? For this discussion, a generalized form of these PDEs is considered.

$$\frac{\partial G}{\partial t} + \nabla \cdot \boldsymbol{f}(G) = 0 \tag{3.7}$$

Here, the flux $\boldsymbol{f}(G)$ must be evaluated along cell boundaries. In this case, it boils down to determining the set of coefficients for G, $\boldsymbol{u}$, and $\hat{\boldsymbol{n}}$, as well as a collection of precomputed surface integral arrays, to be used in tensor multiplication routines.

Several different flux calculation methods are discussed in [5]. There are two criterion used here to determine which numerical fluxes to use: the approach should be (a) computationally inexpensive, and (b) easily formulated in a quadrature-free sense.

Upwinding

In the linear advection case, $\boldsymbol{f}(G) = G\boldsymbol{u}$, the simple upwind flux is chosen–it meets both criteria and is the simplest to implement. The flux function is chosen by picking out the direction in which information propagates, which is determined by the velocity. The use of a multidimensional upwind flux for a DG scheme is described by Marchandise et al. [18] and repeated here.

$$\hat{\boldsymbol{f}}^{\text{up}}\left(G^{\text{out}}, G^{\text{in}}\right) = \begin{cases} \boldsymbol{f}^{\text{out}} & \text{if } \boldsymbol{u}_{\text{fc}} \cdot \hat{\boldsymbol{N}}^{\kappa} \leq 0 \\ \boldsymbol{f}^{\text{in}} & \text{if } \boldsymbol{u}_{\text{fc}} \cdot \hat{\boldsymbol{N}}^{\kappa} > 0 \end{cases} \tag{3.8}$$

Here, $\hat{\boldsymbol{N}}^{\kappa}$ is the outward face normal and $\boldsymbol{u}_{\text{fc}}$ is the velocity $\boldsymbol{u}$ evaluated at the face center. f^{out} is the flux evaluated outside of the cell domain, while f^{in} is evaluated inside.

With this numerical flux, the advection equation with discontinuous Galerkin spatial discretization becomes

$$\frac{\mathrm{d}g_n^\kappa}{\mathrm{d}t} = u_k^{\kappa,j} g_i^\kappa \frac{2}{\Delta x} \int_{\mathscr{K}} b_k b_i \frac{\partial b_n}{\partial \xi_j} \,\mathrm{d}V + u_k^{\mathrm{up},j} g_i^{\mathrm{up}} \frac{2}{\Delta x} \int_{\partial\mathscr{K}} N_j b_k^{\mathrm{up}} b_i^{\mathrm{up}} b_n \,\mathrm{d}S. \tag{3.9}$$

The resulting system of coupled ODEs is now far more straightforward to solve compared to the initial PDE. All that remains is to select an appropriate time-stepping mechanism.

One potential problem with an upwind flux for a discontinuous Galerkin method is that the choice of $\hat{\boldsymbol{f}}$ is dependent on the sign of the velocity $\boldsymbol{u}$. However, the DG formulation permits sub-cell variation of the solution variables, allowing $\boldsymbol{u}$ to change direction entirely within the span of a cell face. In such situations, the choice of $\hat{\boldsymbol{f}}^{\mathrm{up}}$ becomes ambiguous, barring a discontinuous flux function. The DG-CLS method, however, ought to be implemented on a mesh fine enough that the grid length scale is much smaller than the scale at which velocity field variations occur.

Local Lax-Friedrichs

For nonlinear flux functions, such as the convective term $\boldsymbol{f}(G) = G(1-G)\hat{\boldsymbol{n}}$ in the reinitialization equation, the upwind flux is insufficient. Instead, a local Lax-Friedrichs method is implemented.

$$\begin{aligned} \widehat{\boldsymbol{f}\cdot\hat{\boldsymbol{N}}}^{\mathrm{LLF}}\left(G^-, G^+\right) &= \frac{1}{2}\left(\left(\boldsymbol{f}(G^-) + \boldsymbol{f}\left(G^+\right)\right)\cdot\hat{\boldsymbol{N}} - C\left(G^+ - G^-\right)\right) \\ C &= \max_{\min(G^-,G^+) \le s \le \max(G^-,G^+)} \left|\hat{\boldsymbol{N}}\cdot\boldsymbol{f}'(s)\right| \end{aligned} \tag{3.10}$$

where the "±" superscript indicates the solution on the ± side of the face.

Plugging the flux function into Eq. (3.10) gives

$$\begin{aligned} \widehat{\boldsymbol{f}\cdot\hat{\boldsymbol{N}}}^{\mathrm{LLF}}\left(G^-, G^+\right) &= \frac{1}{2}\left(\left(G^-\left(1-G^-\right)\hat{\boldsymbol{n}}^- + G^+\left(1-G^+\right)\hat{\boldsymbol{n}}^+\right)\cdot\hat{\boldsymbol{N}} - C\left(G^+ - G^-\right)\right) \\ C &= \max_{\min(G^-,G^+) \le s \le \max(G^-,G^+)} \left|(1-2s)\,\hat{\boldsymbol{n}}\cdot\hat{\boldsymbol{N}}\right| \end{aligned} \tag{3.11}$$

Finally, this is used to evaluate the convective integral in Eq. (3.6),

$$\begin{aligned}
&\oint_{\partial\mathscr{K}} \overline{G\left(1-G\right)\hat{\boldsymbol{n}}b_n} \cdot \mathrm{d}\hat{\boldsymbol{S}} \;= \\
&\quad \frac{1}{2}\int_{\partial^f\mathscr{K}} \boldsymbol{G}^{f-}\left(1-\boldsymbol{G}^{f-}\right)\hat{\boldsymbol{n}}^{f-}\cdot\hat{\boldsymbol{N}}^f b_n \,\mathrm{d}S \;+\; \frac{1}{2}\int_{\partial^f\mathscr{K}} \boldsymbol{G}^{f+}\left(1-\boldsymbol{G}^{f+}\right)\hat{\boldsymbol{n}}^{f+}\cdot\hat{\boldsymbol{N}}^f b_n \,\mathrm{d}S \\
&\quad -\; \frac{C^f}{2}\int_{\partial^f\mathscr{K}} \left(\boldsymbol{G}^{f+}-\boldsymbol{G}^{f-}\right) b_n \,\mathrm{d}S \\
&\quad = \frac{1}{2}\left(g_i^{f+}\hat{\boldsymbol{n}}_k^{f+} + g_i^{f-}\hat{\boldsymbol{n}}_k^{f-}\right)\cdot\hat{\boldsymbol{N}}^f \int_{\partial^f\mathscr{K}} b_i^{f-}b_k^{f-}b_n \,\mathrm{d}S \\
&\qquad -\frac{1}{2}\left(g_i^{f+}g_j^{f+}\hat{\boldsymbol{n}}_k^{f+} + g_i^{f-}g_j^{f-}\hat{\boldsymbol{n}}_k^{f-}\right)\cdot\hat{\boldsymbol{N}}^f \int_{\partial^f\mathscr{K}} b_i^{f-}b_j^{f-}b_k^{f-}b_n \,\mathrm{d}S \\
&\qquad -\frac{C^f}{2}\left(g_i^{f+}\int_{\partial^f\mathscr{K}} b_i^{f-}b_n \,\mathrm{d}S - g_i^{f-}\int_{\partial^f\mathscr{K}} b_i^{f-}b_n \,\mathrm{d}S\right),
\end{aligned} \tag{3.12}$$

where the superscript $f\pm$ refers to coefficients in the cell on the $\pm$ side of the face f.

Reconstruction

The diffusive flux in the reinitialization equation is handled by yet a third approach, called reconstruction. This method is necessary because the two previous approaches rely on a preferred direction, which simply does not exist for diffusive flux. A reconstruction method was suggested by Luo et al. [16]. This involves projecting the solution from two neighboring cells into one set of coefficients that are associated with a basis extended across the two cells. That is, coefficients associated with neighboring domains $\mathscr{K}^+$ and $\mathscr{K}^-$ are projected into a single domain $\widetilde{\mathscr{K}}$ bisected by the cell face, as shown in Fig. 3.4. In the x-direction,

$$\begin{aligned}
\tilde{\xi} &= \frac{\xi^- - 1}{2} = \frac{\xi^+ + 1}{2} \\
\tilde{\eta} &= \eta^+ = \eta^-
\end{aligned} \tag{3.13}$$

The flux is then evaluated from the coefficients associated with the shared basis:

$$\hat{\boldsymbol{f}}^{\text{recons.}}\left(G^-, G^+, \hat{\boldsymbol{n}}^-, \hat{\boldsymbol{n}}^+\right) = \boldsymbol{f}\left(\widetilde{G}, \tilde{\hat{\boldsymbol{n}}}\right). \tag{3.14}$$

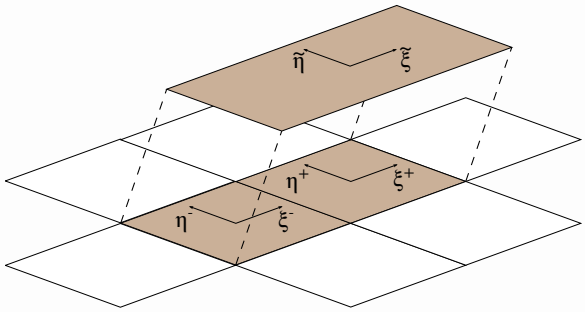


Figure 3.4: Two-cell projection in x-direction

The orthonormality condition is then used to find the coefficients for $\boldsymbol{G}$ and $\hat{\boldsymbol{n}}$ in the shared basis across the α face of a given cell.

$$\widetilde{f}_n^{\alpha} \;=\; f_{i,\kappa}^{\alpha-} \int_{\widetilde{\mathscr{K}}^{\alpha-}} b_i \tilde{b}_n^{\alpha} \,\mathrm{d}\widetilde{V}^{\alpha} \;+\; f_{i,\kappa}^{\alpha+} \int_{\widetilde{\mathscr{K}}^{\alpha+}} b_i \tilde{b}_n^{\alpha} \,\mathrm{d}\widetilde{V}^{\alpha} \tag{3.15}$$

Here, $\widetilde{\mathscr{K}}^{\alpha\pm}$ refers to the ± half of the domain $\widetilde{\mathscr{K}}$. These coefficients are then used in the relevant segment of the flux integral,

$$\oint_{\partial\mathscr{K}} \overline{\overline{\left(\nabla_{\xi} \boldsymbol{G} \cdot \hat{\boldsymbol{n}}\right) \hat{\boldsymbol{n}} b_n}} \cdot \mathrm{d}\hat{\boldsymbol{S}} \;=\; \int_{\partial^f \mathscr{K}} \left(\nabla_{\xi} \widetilde{\boldsymbol{G}}^f \cdot \tilde{\hat{\boldsymbol{n}}}^f\right) \tilde{\hat{\boldsymbol{n}}}^f \cdot \hat{\boldsymbol{N}}^f b_n \,\mathrm{d}S \tag{3.16}$$

where we sum along the faces f. In the shared coordinate space, the cell face simply lies down the center of the extended dimension. For example, a face whose normal is aligned with the x-axis sits on $\tilde{\xi} = 0$. Changing variables to this space gives the integral

$$= \int_{\widetilde{\xi_f}=0} \left(\nabla_{\xi} \widetilde{\boldsymbol{G}}^f \cdot \tilde{\hat{\boldsymbol{n}}}^f\right) \tilde{\hat{\boldsymbol{n}}}^f \cdot \hat{\boldsymbol{N}}^f b_n \,\mathrm{d}\widetilde{S}^f. \tag{3.17}$$

The last step requires transforming $\nabla_{\xi} \to \nabla_{\tilde{\xi}}$, which of course creates a factor of 1/2 in the ξ_f-direction.

$$\frac{\partial}{\partial \xi_{\alpha}} = \left(1 - \delta_{\alpha f}/2\right) \frac{\partial}{\partial \widetilde{\xi}_{\alpha}} \tag{3.18}$$

Finally, expanding the solution variables into their coefficient representation,

$$\oint_{\partial\mathscr{K}} \overline{\overline{\left(\nabla_{\xi} \boldsymbol{G} \cdot \hat{\boldsymbol{n}}\right) \hat{\boldsymbol{n}} b_n}} \cdot \mathrm{d}\hat{\boldsymbol{S}} \;=\; \tilde{g}_i^f \tilde{n}_j^{f,k} \tilde{\hat{\boldsymbol{n}}}_l^f \cdot \hat{\boldsymbol{N}}^f \left(1 - \delta_{kf}/2\right) \int_{\widetilde{\xi_f}=0} \frac{\partial \tilde{b}_i^f}{\partial \widetilde{\xi}_k} \widetilde{b}_j^{\,f} \widetilde{b}_l^{\,f} b_n \,\mathrm{d}\widetilde{S}^f \tag{3.19}$$

Combining Eqs. (3.6), (3.12), and (3.19) gives the form of the DG scheme for reinitialization following spatial discretization and flux evaluation.

$$\begin{aligned}\frac{\mathrm{d}g_n^\kappa}{\mathrm{d}\tau} &= n_k^{\kappa,j} g_i^\kappa \frac{2}{\Delta x}\int_{\mathscr{K}} b_k b_i \frac{\partial b_n}{\partial \xi_j}\,\mathrm{d}V \;-\; n_k^{\kappa,j} g_i^\kappa g_j^\kappa \frac{2}{\Delta x}\int_{\mathscr{K}} b_k b_i b_j \frac{\partial b_n}{\partial \xi_j}\,\mathrm{d}V \\ &\quad - \varepsilon g_i^\kappa n_k^{\kappa,a} n_l^{\kappa,d}\left(\frac{2}{\Delta x}\right)^2 \int_{\mathscr{K}} \frac{\mathrm{d}b_i}{\mathrm{d}\xi_a} b_k b_l \frac{\mathrm{d}b_n}{\mathrm{d}\xi_d}\,\mathrm{d}V \\ &\quad - \frac{1}{\Delta x}\left(g_i^{f+}\hat{\boldsymbol{n}}_k^{f+} + g_i^{f-}\hat{\boldsymbol{n}}_k^{f-}\right)\cdot\hat{\boldsymbol{N}}^f \int_{\partial^f\mathscr{K}} b_i^{f-} b_k^{f-} b_n\,\mathrm{d}S \\ &\quad + \frac{1}{\Delta x}\left(g_i^{f+} g_j^{f+}\hat{\boldsymbol{n}}_k^{f+} + g_i^{f-} g_j^{f-}\hat{\boldsymbol{n}}_k^{f-}\right)\cdot\hat{\boldsymbol{N}}^f \int_{\partial^f\mathscr{K}} b_i^{f-} b_j^{f-} b_k^{f-} b_n\,\mathrm{d}S \\ &\quad + \frac{C^f}{\Delta x}\left(g_i^{f+}\int_{\partial^f\mathscr{K}} b_i^{f-} b_n\,\mathrm{d}S - g_i^{f-}\int_{\partial^f\mathscr{K}} b_i^{f-} b_n\,\mathrm{d}S\right) \\ &\quad + \varepsilon\left(\frac{2}{\Delta x}\right)^2 \tilde{g}_i^f \tilde{n}_j^{f,k}\tilde{\hat{\boldsymbol{n}}}_l^f\cdot\hat{\boldsymbol{N}}^f\left(1-\delta_{kf}/2\right)\int_{\widetilde{\xi_f}=0}\frac{\partial\tilde{b}_i^f}{\partial\widetilde{\xi}_k}\widetilde{b}_j^{\,f}\widetilde{b}_l^{\,f} b_n\,\mathrm{d}\widetilde{S}^f\end{aligned} \tag{3.20}$$

The coefficients associated with an extended basis, denoted with a tilde (e.g., $\tilde{g}_i^f$), are calculated via Eq. (3.15).

3.3 Temporal Discretization

The spatial discretization inherent to the DG method and the discussed methods for handling numerical flux result in a system of $2N_g N_{\text{cells}}$ ordinary differential equations (ODEs), $\frac{\mathrm{d}}{\mathrm{d}t}g_m^\kappa = L\left(\{g_i^\kappa\}\right)$. Despite being a larger system of equations, ODEs are far more easily solved. At this point, a good choice of temporal discretization will complete the RKDG method.

Of course, simple forward Euler time stepping will solve the equations. However, a more accurate choice will allow the method to achieve the full $k+1$ convergence rate. To accomplish this, an explicit $k+1$ order Runge-Kutta total variation diminishing (TVD) approach is implemented, as used in [5] and described by Gottlieb [10].

1. set $g_{m,\kappa}^{(0)} = g_{m,\kappa}^n$

2. solve for intermediate stages

$$g_{m,\kappa}^{(i)} = \sum_{k=0}^{i-1} \left(\alpha_{i,k} g_{m,\kappa}^{(k)} + \Delta t \beta_{i,k} L\left(\{ g_{m,\kappa}^{(k)} \} \right) \right), \quad i = 1, ..., N_{\mathrm{RK}} \tag{3.21}$$

3. set $g_{m,\kappa}^{n+1} = g_{m,\kappa}^{(N_{\mathrm{RK}})}$

where [5, 10] provide stable values for $\alpha_{i,k}, \beta_{i,k}$, which are reprinted in Table 3.1.

Table 3.1: Stable values for $\alpha_{i,k}, \beta_{i,k}$ presented by Cockburn and Shu [5] and Gottlieb [10]

order	$\alpha_{i,k}$	$\beta_{i,k}$
2	1	1
	$\frac{1}{2}, \frac{1}{2}$	$0, \frac{1}{2}$
2	1	1
	$\frac{3}{4}, \frac{1}{4}$	$0, \frac{1}{4}$
	$\frac{1}{3}, 0, \frac{2}{3}$	$0, 0, \frac{2}{3}$

Finally, the time step size is limited by CFL conditions, as found in a von Neumann stability analysis. The CFL condition for convective terms is provided by [5]:

$$\max |\boldsymbol{f}'(G)| \frac{\Delta t}{\Delta x} \leq \frac{1}{2k+1} \tag{3.22}$$

The flux function derivatives, knowing that $|\hat{\boldsymbol{n}}| = 1$ and the CLS method restricts $0 \leq G \leq 1$, are

$$\begin{aligned} \text{advection:} \quad & \max |\boldsymbol{f}'(G)| = \max |\boldsymbol{u}| \\ \text{reinitialization convective term:} \quad & \max |\boldsymbol{f}'(G)| = 1 \end{aligned} \tag{3.23}$$

The diffusive term in the reinitialization equation restricts time step size by [15]:

$$\varepsilon \frac{\Delta t}{\Delta x^2} \leq \frac{\beta(k)}{(2k+1)^2 \sqrt{d}} \tag{3.24}$$

where $\beta(k)$ is a function of polynomial order (several values are given in Table 3.2). Note that, in practice, $\varepsilon \sim \Delta x$ so that Δt scales with Δx rather than its square.

Table 3.2: Stable values for β presented by Lörcher et al. [15]

k	1	2	3	4	5	6	7
β	1.46	0.80	0.40	0.24	0.16	0.12	0.09

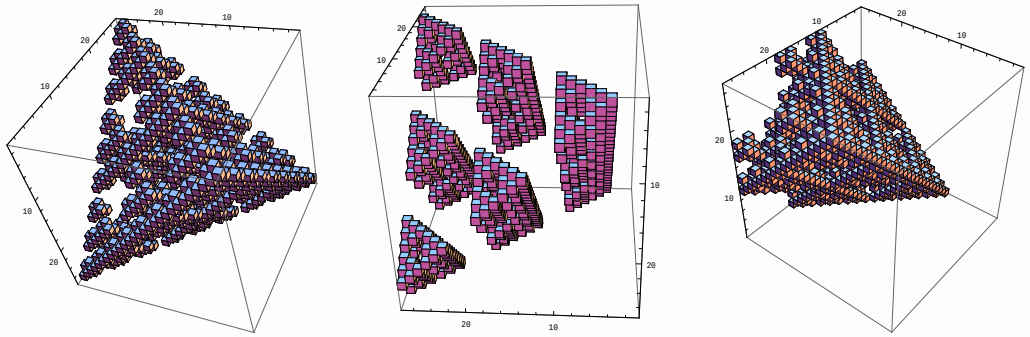
3.4 Sparsity

The schemes for advection and reinitialization, Eqs. (3.9) and (3.20), are written in terms of integrals dependent only on the basis functions and cell domain. These integrals can be precomputed analytically using symbolic software such as Mathematica, Maple, or SymPy, and stored in an array for reference later (see Ap. A). This avoids the use of quadrature, saving computation time. Most importantly, the resulting 3D arrays are sparse, resulting from the orthogonality of the Legendre polynomial basis (see Table 3.3 and Fig. 3.5). For reference, the advection volume integrals are named Ax, Ay, and Az, and "-" face surface integrals are SAxm, SAym, and SAzm. Together, the high number of operations with comparatively few solution variables and the integral array sparsity make this method ideal for GPU computation.

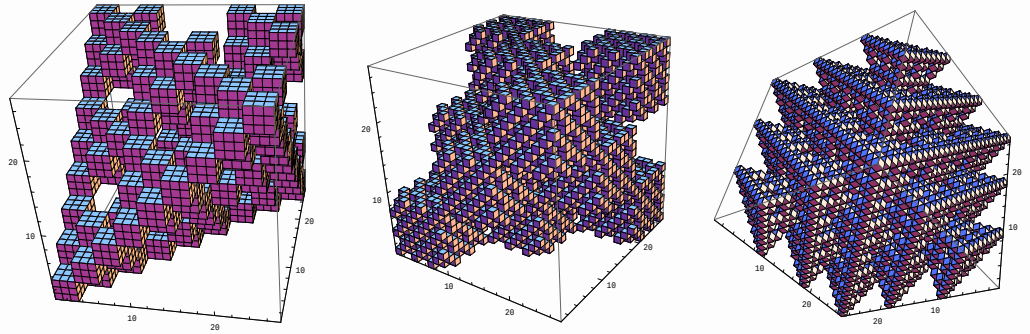
Table 3.3: Matrix Fill Fraction for Advection Integral Arrays

Polynomial	2D Simulation Integrals			3D Simulation Integrals		
Degree	# of elements	Volume	Surface	# of elements	Volume	Surface
1	64	12.5%	50.0%	512	6.25%	25.0%
2	729	10.6%	40.7%	19683	4.30%	16.6%
3	4096	10.1%	35.9%	262144	3.63%	12.9%
4	15625	9.68%	33.6%	1953125	3.25%	11.3%

On the computer science side of things, it is quite beneficial to store these arrays in a manner that takes advantage of their sparse structure. For a CPU-based simulation, effi-



(a) Ax (b) Ay (c) Az



(d) SAxm (e) SAym (f) SAzm

Figure 3.5: Sparsity illustration for $k = 2$ 3D advection integral arrays. Cubes are placed at array locations containing nonzero elements.

cient memory management offers some improvement. However, for reasons that will be described later, efficient memory management on a GPU is vital. Furthermore, the sparse structure of these arrays advocates a quadrature-free formulation. A quadrature-based scheme, relying on numerical integration, evaluates the integrals without separating coefficients and basis functions. As a result, such an approach does not know ahead of time that much of the work can be bypassed. The sparsity found in the quadrature-based scheme automatically takes care of this if an appropriate storage format is used.

A compressed row storage (CRS) format based on [8] was chosen, allowing the integrals to be stored in 1D arrays along with several corresponding 1D arrays of ints giving nonzero entry locations. By doing so, the amount of data that must be sent to the GPU is reduced and parallelization on the GPU is simplified. For more details, refer to Ch. 5.

Chapter 4

Normal Vector Calculation

In order to complete the DG-CLS scheme, it is necessary to develop an arbitrary-order approach for computing normal vectors and curvature. The focus of this chapter is normal vectors, leaving curvature calculation to future work. As discussed in Ch. 2, the normal vectors should be dependent on the level set scalar G only in the vicinity of the interface [7]. In contrast to the CLS method proposed in [22], local variations cannot be allowed to impact the normal vector field. To accomplish this, the normals are evaluated using the following general outline:

1. Compute the signed distance function ϕ inside $\mathscr{A}$-band by inverting the hyperbolic tangent profile for G, Eq. (2.4) (note the nonlinear integrand necessitates quadrature).
$$\phi_n \;=\; 2\varepsilon \int_{\mathscr{K}} \operatorname{atanh}\left(2 g_i b_i - 1\right) b_n \,\mathrm{d}V \tag{4.1}$$
2. Compute ϕ outside $\mathscr{A}$-band by the fast sweeping method.
3. Compute $\nabla\phi$.
    a) Project ϕ into two (three for 3D) enriched polynomial spaces of order $(3k+2)^3$ extended across three-cell stencils, yielding $\widetilde{\phi}$.
    b) Calculate $\nabla\phi$ from $\widetilde{\phi}$.
4. Compute $\hat{\boldsymbol{n}}$ from Eq. (2.6) with $\nabla\phi$ (again the nonlinear integrand necessitates quadrature).
$$n_n^k = \int_{\mathscr{K}} \frac{\mathrm{d}\phi_i^k b_i}{\sqrt{\sum_{j=1}^{3}\left(\mathrm{d}\phi_k^j b_k\right)^2}} b_n \,\mathrm{d}V \tag{4.2}$$

4.1 The Fast Sweeping Method

Through $\Omega \setminus \mathscr{A}$, the physical domain outside the $\mathscr{A}$-band, the signed distance function is determined by a special case of the Eikonal equation (which is a special form of a time-independent Hamilton-Jacobi equation),

$$|\nabla \phi| = 1, \tag{4.3}$$

with $\phi \in \mathscr{A}$ as a Dirichlet boundary condition. The solution approach is developed for the general Eikonal equation with $F(\boldsymbol{x}) > 0$,

$$|\nabla \phi| = F(\boldsymbol{x}) \tag{4.4}$$

for the purpose of allowing MMS verification in the future.

Finite difference based schemes often utilize the fast marching method (FMM). Comparatively, this method is quite efficient and scales with $O(n \log n)$. However, it is not clear how this method might be generalized to a DG discretization. Here, the fast sweeping method (FSM) is implemented. It is comparatively less efficient, despite having $O(n)$ scaling. However, it's DG formulation is more apparent and it is easily parallelized. Other works have utilized the FSM to develop 2^{nd} order DG Eikonal equation solvers [17, 33]. These studies are extended by exploring a generalized, arbitrary-order DG-FSM solver.

Overview

The fast sweeping method suggested by Boué and Dupuis [2] involves iterating through the computational domain with alternating orderings (called sweeping), updating the solution variable in a Gauss-Seidel sense. In 2D:

1. $i = 1...N_{\text{imax}},\ j = 1...N_{\text{jmax}}$
2. $i = N_{\text{imax}}...1,\ j = 1...N_{\text{jmax}}$
3. $i = N_{\text{imax}}...1,\ j = N_{\text{jmax}}...1$

4. $i = 1...N_{\text{imax}},\ j = N_{\text{jmax}}...1$

A similar sequence is used in 3D, except with eight sweeps instead of four. By sweeping in different orderings, the characteristics of the problem are more quickly mapped out and information can be more quickly transported through the domain. These sweep orderings are repeated until reaching convergence, which occurs when the L_∞ difference between two sweeps is less than some δ, which here is taken to be 10^{-11}. The solution values are updated by first determining the upwind direction, which is found using the fact that the solution is non-decreasing along the characteristics [14, 31]. With the signal propagation direction determined from causality, the DG coefficients are updated by solving a nonlinear system of algebraic equations. Here, Newton's method is suggested, while previously the 2nd order problem has been solved by Li et al. [14] and Zhang et al. [33] using a quadratic solver.

Unfortunately, the DG form of the fast sweeping solver presented by [33] is unstable unless provided with two pieces of information: 1) a good initial guess and 2) causality flags indicating the flow of information. Luo [17] suggests an alternative 2nd order DG fast sweeping solver that does not require the presence of a finite-difference based solver and is far simpler to implement. However, it is not clear how to generalize his method to arbitrary order. The approach described by [33] is given here, simplified for constant cell spacing, generalized for arbitrary DG order, and modified for computing the *signed* distance function rather than unsigned.

Finite-Difference Sweeper

Zhang produces the initial guess via a first-order finite difference based Godunov fast sweeping solver, which also provides a first order approximation of the causality directions. That is, the FD sweeper provides upwind data telling each cell where it should receive information from. This in essence maps out the characteristics, which are then

corrected from higher order terms by the DG sweeper.

The Eikonal equation is solved on a finite difference grid which exists on the vertices of the DG cells, such that a function on node $F\left(x_i, y_j\right) = F_{ij}$. Integer causality flags caux_{ij} and cauy_{ij} are defined such that a value of 0 indicates information propagating from - to +, 1 indicates the opposite, and 10 means information does not flow along the indicated direction for the node (i, j). These flags are initialized as 10 and stored, depending on the situation, in arrays flagx_{ij} and flagy_{ij}. In 2D, Eq. (4.4) is squared and discretized as

$$\left(\frac{\varphi_{ij} - a}{\Delta x}\right)^2 + \left(\frac{\varphi_{ij} - b}{\Delta y}\right)^2 = F_{ij}^2 \tag{4.5}$$

where a and b are the solution ϕ at a neighboring node selected by upwinding. At interior nodes, these values are updated along with caux and cauy using

$$\begin{cases} \begin{cases} a = \varphi_{i-1,j}, & \text{caux}_{ij} = 0 \quad \text{if } \varphi_{i-1,j} < \varphi_{i+1,j} \\ a = \varphi_{i+1,j}, & \text{caux}_{ij} = 1 \quad \text{otherwise} \end{cases} & \text{if } G > 0.5 \\ \begin{cases} a = \varphi_{i-1,j}, & \text{caux}_{ij} = 0 \quad \text{if } \varphi_{i-1,j} > \varphi_{i+1,j} \\ a = \varphi_{i+1,j}, & \text{caux}_{ij} = 1 \quad \text{otherwise} \end{cases} & \text{if } G < 0.5 \end{cases} \tag{4.6}$$

with a similar definition for b in the y-direction. At nodes bordering the edge of the computational domain, a one-sided definition is used forcing all information to come from the interior.

Eq. (4.4) is then easily solved using the quadratic formula, with special cases to ensure

that ϕ is always non-decreasing/non-increasing.

$$\varphi_{ij} = \begin{cases} \begin{cases} \frac{a\Delta y^2 + b\Delta x^2 + \Delta x \Delta y \sqrt{F_{ij}^2\left(\Delta x^2 + \Delta y^2\right) - (a-b)^2}}{\Delta x^2 + \Delta y^2}, & \text{if } -\Delta y F_{ij} < b - a < \Delta x F_{ij} \\ b + \Delta y F_{ij}, & \text{if } b - a \leq -\Delta y F_{ij} \\ a + \Delta x F_{ij}, & \text{if } b - a \geq \Delta x F_{ij} \end{cases} & \text{if } G > 0.5 \\ \begin{cases} \frac{a\Delta y^2 + b\Delta x^2 - \Delta x \Delta y \sqrt{F_{ij}^2\left(\Delta x^2 + \Delta y^2\right) - (a-b)^2}}{\Delta x^2 + \Delta y^2}, & \text{if } -\Delta x F_{ij} < b - a < \Delta y F_{ij} \\ a - \Delta y F_{ij}, & \text{if } b - a \leq -\Delta x F_{ij} \\ b - \Delta x F_{ij}, & \text{if } b - a \geq \Delta y F_{ij} \end{cases} & \text{if } G < 0.5 \end{cases} \tag{4.7}$$

The causality arrays are updated via

$$\begin{cases} \begin{cases} \text{flagx}_{ij} = \text{caux}_{ij}, \quad \text{flagy}_{ij} = \text{cauy}_{ij} & \text{if } -\Delta y F_{ij} < b - a < \Delta x F_{ij} \\ \text{flagx}_{ij} = 10, \quad \text{flagy}_{ij} = \text{cauy}_{ij} & \text{if } b - a \leq -\Delta y F_{ij} \\ \text{flagx}_{ij} = \text{caux}_{ij}, \quad \text{flagy}_{ij} = 10 & \text{if } b - a \geq \Delta x F_{ij} \end{cases} & \text{if } G > 0.5 \\ \begin{cases} \text{flagx}_{ij} = \text{caux}_{ij}, \quad \text{flagy}_{ij} = \text{cauy}_{ij} & \text{if } -\Delta x F_{ij} < b - a < \Delta y F_{ij} \\ \text{flagx}_{ij} = \text{caux}_{ij}, \quad \text{flagy}_{ij} = 10 & \text{if } b - a \leq -\Delta x F_{ij} \\ \text{flagx}_{ij} = 10, \quad \text{flagy}_{ij} = \text{cauy}_{ij} & \text{if } b - a \geq \Delta y F_{ij} \end{cases} & \text{if } G < 0.5 \end{cases} \tag{4.8}$$

To initialize the sweeper, large positive values are assigned to φ_{ij} at all nodes where $G > 0.5$ and large negative values are assigned at all nodes where $G < 0.5$. As previously noted, inside the $\mathscr{A}$-band where G is near 0.5 the signed distance function is calculated from inverting the hyperbolic tangent profile and held constant through the sweeping solver iterations.

Discontinuous Galerkin Sweeper

A DG solver designed to give the coefficients ϕ_i^κ is described here. The initial condition to the system is calculated from the FD-based Godunov solver described in the previous

section. The DG coefficients up to first order can be calculated explicitly by

$$\begin{aligned}
\phi_0 &= \frac{1}{2}\varphi^{\mathrm{bl}} + \frac{1}{2}\varphi^{\mathrm{tl}} + \frac{1}{2}\varphi^{\mathrm{br}} + \frac{1}{2}\varphi^{\mathrm{tr}} \\
\phi_x &= -\frac{1}{2\sqrt{3}}\varphi^{\mathrm{bl}} - \frac{1}{2\sqrt{3}}\varphi^{\mathrm{tl}} + \frac{1}{2\sqrt{3}}\varphi^{\mathrm{br}} + \frac{1}{2\sqrt{3}}\varphi^{\mathrm{tr}} \\
\phi_y &= -\frac{1}{2\sqrt{3}}\varphi^{\mathrm{bl}} + \frac{1}{2\sqrt{3}}\varphi^{\mathrm{tl}} - \frac{1}{2\sqrt{3}}\varphi^{\mathrm{br}} + \frac{1}{2\sqrt{3}}\varphi^{\mathrm{tr}} \\
\phi_{xy} &= +\frac{1}{6}\varphi^{\mathrm{bl}} - \frac{1}{6}\varphi^{\mathrm{tl}} - \frac{1}{6}\varphi^{\mathrm{br}} + \frac{1}{6}\varphi^{\mathrm{tr}}
\end{aligned} \tag{4.9}$$

where the $\phi_0, \phi_x, \phi_y, \phi_{xy}$ indicate the coefficients corresponding to the constant, linear in x, linear in y, and proportional to xy basis functions, respectively. The superscripts $\mathrm{bl}, \mathrm{tl}, \mathrm{br}, \mathrm{tr}$ indicate the bottom-left, top-left, bottom-right, and top-right nodes, respectively. Then, Eq. (4.4) is squared, variables changed to sub-cell coordinates, and discretized by taking an inner product with a test function b_n. Here, the ideas of Cheng and Shu [3] are followed to produce

$$\left(\frac{2}{\Delta x}\right)^2 \int_{\mathscr{K}} \left|\nabla_\xi \phi\right|^2 b_n \,\mathrm{d}V + \alpha_\kappa^f \left(\frac{2}{\Delta x}\right)^2 \int_{\partial^f \mathscr{K}} [\phi]^f \, b_n \,\mathrm{d}S = \int_{\mathscr{K}} F^2 b_n \,\mathrm{d}V. \tag{4.10}$$

where α^f are called local causality constants by Zhang et al. [33]. His definition is modified to the following for the square of the Eikonal equation:

$$\alpha^\sigma = \begin{cases} \mathrm{m}\left(0, H_\sigma(\nabla\phi)\Big|_{\mathscr{K}^\sigma}\right) = \mathrm{m}\left(0, 2\dfrac{\partial\phi}{\partial x_\sigma}\Big|_{\mathscr{K}^\sigma}\right), & \text{if } \mathrm{flag}\sigma_{ij} = 0 \,\&\, \mathrm{ave}\left(|\nabla\phi|^2\Big|_{\mathscr{K}}\right) \neq 0 \\ \text{skip current cell}, & \text{if } \mathrm{flag}\sigma_{ij} = 0 \,\&\, \mathrm{ave}\left(\nabla\phi\Big|_{\mathscr{K}}\right) = \mathbf{0} \\ 0, & \text{if } \mathrm{flag}\sigma_{ij} = 1 \text{ or } \mathrm{flag}\sigma_{ij} = 10 \end{cases} \tag{4.11}$$

where $H_\sigma = \dfrac{\partial\,|\nabla\phi|}{\partial\phi_\sigma}$ is the derivative of the Hamiltonian with respect to spacial derivatives of ϕ, the $\mathrm{m}(\cdot,\cdot)$ function is a max where $G > 0.5$ and min where $G < 0.5$, and the derivative of ϕ in a neighboring cell is evaluated as the average over the cell. With these definitions, the Legendre basis expansion is inserted into Eq. (4.10) to form a nonlinear

system of algebraic equations:

$$\begin{aligned}\phi_i^{\kappa,\text{new}}\phi_j^{\kappa,\text{new}}\left(\frac{2}{\Delta x}\right)^2\int_{\mathscr{K}}\frac{\partial b_i}{\partial\xi_k}\frac{\partial b_j}{\partial\xi_k}b_n\,\mathrm{d}V \;+\; \alpha_\kappa^f\phi_i^{\kappa,\text{new}}\left(\frac{2}{\Delta x}\right)^2\int_{\partial^f\mathscr{K}}b_i^{\text{in}}b_n^{\text{in}}\,\mathrm{d}S \\ = \;\left(F^2\right)_n+\alpha_\kappa^f\phi_i^f\int_{\partial^f\mathscr{K}}b_i^{\text{out}}b_n^{\text{in}}\,\mathrm{d}S\end{aligned} \tag{4.12}$$

where the $\{\phi_i^{\kappa,\text{new}}\}$ coefficients are updated from the current neighbors following the Gauss-Seidel philosophy. Note also the two surface integrals in Eq. (4.12), one of which is evaluates b_i on the inner side face and the other evaluates b_i on the outer side of the face, as denoted by superscripts. At this point, Zhang has a quadratic system of three equations that he solves by substitution. Here, a larger system of N_g equations must be solved.

Several solution attempts have been made thus far, the first being linearization by changing $\phi_j^{\kappa,\text{new}} \to \phi_j^{\kappa}$. This, however, caused divergence everywhere before converging at unrealistic answers.

A second attempt involves solving for the $n = 1$ coefficient first, and truncating the equation such that there is no dependence on higher terms. This is followed by solving for the $n = 2$ coefficient, again truncating the equation to remove higher-order dependence. This causes the solution to blow up in some cases, however. Consider, for instance, a cell for which $F = \alpha^{x-} = \alpha^{y-} = \alpha^{y+} = 0$ and α^{x+}«1, and note that the volume integral at lowest order is equal to zero as a result of the derivative terms. From Eq. (4.12), the solution is then set by a term divided by a small α. Furthermore, achieving high order convergence rates is contingent on low order coefficients receiving corrections from high order terms, as is done in advection and reinitialization.

The final attempt involved employing a tensor-based Newton's method to find the nearest root to Eq. (4.12) at every grid point κ. Of course, the nonlinearity allows multiple roots to this system, so convergence on the correct one relies on the accuracy of the previous iteration and FD solver.

$$f_n(\boldsymbol{\phi}) = A_{nij}\phi_i\phi_j + B_{ni}\phi_i + C_n = 0 \tag{4.13}$$

From a first initial guess $\boldsymbol{\phi}^{(0)}$, an improved approximation to the root $\boldsymbol{\phi}^{(n+1)} = \boldsymbol{\phi}^{(n)} + \boldsymbol{\delta}$ is found by iterating

$$\begin{aligned} f_i\left(\boldsymbol{\phi}^{(n)}\right) + \boldsymbol{J}_{ij}^{(n)}\delta_j &= 0 \\ \boldsymbol{\delta} &= -\left(\boldsymbol{J}^{(n)}\right)^{-1}\boldsymbol{f}^{(n)} \end{aligned} \tag{4.14}$$

where the Jacobian $\boldsymbol{J}^{(n)}$ is

$$\boldsymbol{J}_{ij}^{(n)} = \frac{\partial f_i\left(\boldsymbol{\phi}^{(n)}\right)}{\partial \phi_j} = A_{ilj}\phi_l^{(n)} + A_{ijl}\phi_l^{(n)} + B_{ij} \tag{4.15}$$

This method, however, only provides 1st order convergence rates regardless of the convergence criterion of Newton's root finder as a result of errors in regions where characteristics intersect. This implies the need of a causality flag correction routine, which is left to future work.

Parallelization

Parallelization for the FSM solver is quite simple compared to parallelization of a FMM solver. The domain is decomposed into blocks of cells, and the alternating sweeps are executed within those blocks only. That is, instead of sweeping globally from $i = 1...N_{\text{imax}}$, the loops are executed within the local block from $i = N_{\text{block imin}}...N_{\text{block imax}}$. Then, inter-processor communication of ghost cell values between neighboring blocks is performed periodically. To avoid a high communication overhead, ghost cell update routines are performed after every sequence of 4 sweeps in 2D or 8 sweeps in 3D.

4.2 Gradient Calculation

Finally, the normal vectors are dependent on the gradient of the signed distance function, which at this point has been found using inversion and sweeping. To differentiate a field across the computational domain, it is possible to do so by simply evaluating

$$\mathrm{d}\phi_n^\alpha = \phi_i \frac{2}{\Delta x_\alpha} \int_{\mathscr{K}} \frac{\mathrm{d}b_i}{\mathrm{d}\xi_\alpha} b_n \,\mathrm{d}V$$

where

$$\frac{\partial \phi}{\partial x_\alpha} = \sum_{i=1}^{N_g} \mathrm{d}\phi_i^{\kappa,\alpha}(t)\, b_i(\boldsymbol{\xi}),$$

However, this approach results in a loss of order since only $N_g - 1$ degrees of freedom in ϕ contribute to $\mathrm{d}\phi$. To ensure that the derivative contains N_g nonzero coefficients and to smooth out discontinuities across cells in the normal vector, ϕ is projected into an enriched polynomial space of order $N_{gex} = (3k+2)^3$ that is extended across two neighbors in the direction of differentiation. This projection is handled in a similar manner to the diffusive flux projection in the reinitialization equation (see Sec. 3.2).

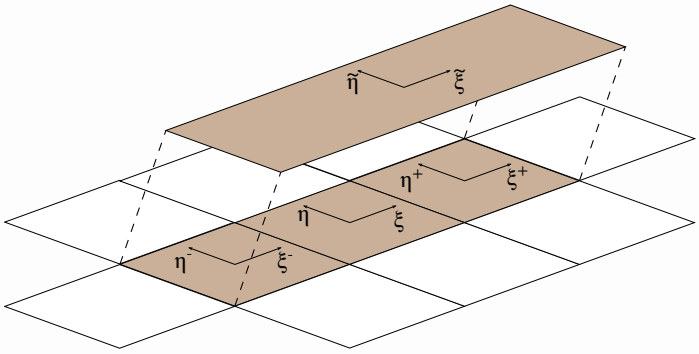


Figure 4.1: Three-cell projection in x-direction

In the x-direction, it can easily be seen that the coordinates are related by

$$\begin{gathered} \tilde{\xi} = \frac{1}{3}\xi = \frac{1}{3}\left(\xi^+ + 2\right) = \frac{1}{3}\left(\xi^- - 2\right) \\ \tilde{\eta} = \eta = \eta^+ = \eta^- \end{gathered} \tag{4.16}$$

The orthonormality condition is then used to find the ϕ coefficients on the extended basis, where the extended domain $\widetilde{\mathscr{K}}$ is partitioned into three sub-domains $\widetilde{\mathscr{K}}^{\alpha-}$, $\widetilde{\mathscr{K}}^{\alpha}$, and $\widetilde{\mathscr{K}}^{\alpha+}$, corresponding to the three original domains in the new extended space (see Fig. 4.1). $\phi_i^{\alpha\pm}$ refers to coefficients in the cell sharing the $\alpha\pm$ face.

$$\widetilde{\phi}_n^{\alpha} = \phi_i^{\alpha-} \int_{\widetilde{\mathscr{K}}^{\alpha-}} b_i^{\alpha-} \tilde{b}_n \,\mathrm{d}\widetilde{V} + \phi_i \int_{\widetilde{\mathscr{K}}^{\alpha}} b_i \tilde{b}_n \,\mathrm{d}\widetilde{V} + \phi_i^{j+} \int_{\widetilde{\mathscr{K}}^{\alpha+}} b_i^{\alpha+} \tilde{b}_n \,\mathrm{d}\widetilde{V} \tag{4.17}$$

The resulting coefficients are then differentiated,

$$\mathrm{d}\phi_n^\alpha = \widetilde{\phi}_i^\alpha \frac{2}{\Delta x_\alpha} \int_{\mathscr{K}} \frac{\partial \tilde{b}_i}{\partial \xi_\alpha} b_n \,\mathrm{d}V \tag{4.18}$$

Eqs. (4.17) and (4.18) are combined, allowing the entire projection and differentiation procedure to be executed with a single equation:

$$\begin{aligned} \mathrm{d}\phi_n^\alpha \;=\; & \phi_k^{\alpha-} \frac{2}{\Delta x_\alpha} \int_{\widetilde{\mathscr{K}}^{\alpha-}} b_k^{\alpha-} \tilde{b}_i \,\mathrm{d}\widetilde{V} \int_{\mathscr{K}} \frac{\partial \tilde{b}_i}{\partial \xi_\alpha} b_n \,\mathrm{d}V \\ & + \phi_k \frac{2}{\Delta x_\alpha} \int_{\widetilde{\mathscr{K}}^{\alpha}} b_k \tilde{b}_i \,\mathrm{d}\widetilde{V} \int_{\mathscr{K}} \frac{\partial \tilde{b}_i}{\partial \xi_\alpha} b_n \,\mathrm{d}V \\ & + \phi_k^{\alpha+} \frac{2}{\Delta x_\alpha} \int_{\widetilde{\mathscr{K}}^{\alpha+}} b_k^{\alpha+} \tilde{b}_i \,\mathrm{d}\widetilde{V} \int_{\mathscr{K}} \frac{\partial \tilde{b}_i}{\partial \xi_\alpha} b_n \,\mathrm{d}V \end{aligned} \tag{4.19}$$

The integral terms, summing $i = 1, N_{gex}$ can be precomputed at an arbitrary level of enrichment, thereby significantly reducing runtime for this routine.

Chapter 5

GPU Programming Model

Using OpenCL terminology, a GPU operates by executing a function called a *kernel* in parallel on a cluster of *work-items*, which are organized into *work-groups* with an associated memory space we call *tiles*. Each work-item then has a global id and local id, and each work-group has a group id.

There are several nuances of this model to take advantage of, the most important of which is how workloads are managed. For instance, if work-items inside the same work-group are given drastically different workloads (thread divergence), the entire work-group must wait for the slowest member to complete its task before moving on to the next portion of the problem. As a result, it is highly advantageous (and even necessary to truly take advantage of the hardware) to assign uneven workloads to different work-groups, rather than within work-groups. Although the code is written with GPUs in mind, The OpenCL model allows it to be effectively implemented on other architectures (e.g., Intel's many integrated core, or MIC).

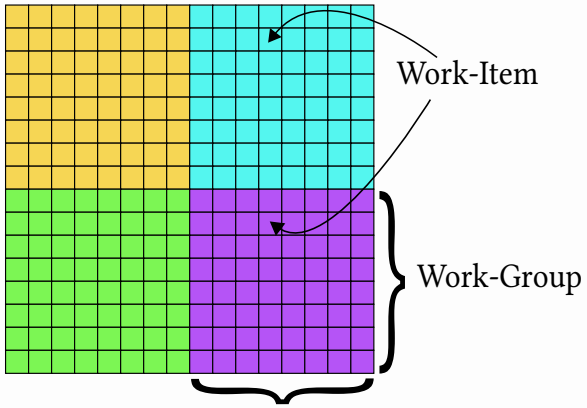


Figure 5.1: OpenCL Execution Model

To take advantage of this aspect of parallelism, and avoid thread divergence, Eq. (3.9) is solved by assigning a single $g_{n,\kappa}$ to each tile. Then, work-items share the workload

of tensor-vector multiplication and summation, avoiding uneven workload distributions between work-items (called thread divergence). This way, if an integral array has a largely zero row n, the entire work-group is given a lighter workload. This allows it to finish earlier and execute a new work-group rather than waiting for individual work-items to finish their work.

A second aspect of GPU programming and OpenCL is the use of different memory spaces for array storage: i) *global* memory, which is available to all work-items but is very slow, invoking an additional 400-600 clock cycles of latency when accessed [20] (for comparison, memory read/write time itself is 8 operations per clock cycle), ii) *local* memory, which is shared between members of the same work-group and may be accessed ∼100x faster than global memory [27], and iii) small *private* registers, which is not shared between work-items but is slightly faster than local memory. An excellent description of this is given by Scarpino [27]. Unfortunately, a limitation exists that local memory arrays cannot be dynamically allocated. To get around this, the CPU can send a request for variable sized blocks of memory to be reserved before queuing kernel execution. Currently, only global memory is used for solution variable and integral arrays. Since a single g_n^κ is assigned to a work-group, there is no benefit to transferring any arrays to local memory. In fact, results show a few percent slow-down in doing so, since a given element on these arrays is only accessed a few times per work-group. However, it is possible to assign a single coefficient degree, g_n, to a work-group and loop through all cells, κ. By doing this, a subset of the integral arrays may be transferred to local memory and reused hundreds or thousands of times, depending on the size of the mesh. This has not yet been pursued, but is left to future work.

In Alg. 1, Eq. (3.9) is considered a series of equations of the form $\Delta g_n^\kappa \mathrel{+}= \sum_{k=1}^{N_u} u_k^\kappa \sum_{i=1}^{N_g} g_i^\kappa Z_{n,k,i}$. Each work-item has its own instance of the variable `my_dg`. Work-items then proceed to sum together a subset of the above equation,

that is, products of elements of velocity u, level set scalar g, and the integral array, denoted Z for generalization in the code. Instead of looping over both k and i, we loop over a single integer l that corresponds to nonzero elements of the compressed array Z[l]. Two arrays Zi2 and Zi3 tabulate the full 3D array coordinates k and i associated with l for each iteration. Finally, each work-group has its own value of Δg_n^κ to compute, with a unique combination of n and κ. Since each work-group only has one value of n, it only needs to loop through a subset of the integral array Z. As a result, it is necessary to pass in two integers Zstart[n] and Zend[n] that give the bounds of this subsection.

In order to take advantage of memory coalescence and evenly distribute the workload, and hence reduce runtime, the local group of work-items must align their access to the global array Z by their local id number [20]. For example, the work-item with local id 7 will access the array element immediately after the work-item with local id 6 and immediately before the work-item with local id 8. To accomplish this, work-items begin the loop offset by their local id, and step through the loop by the local work-group size. On the next

Algorithm 1 GPU 3D Array Multiplication

```
tiX ← local ID of thread
ntX ← size of local work-group                    ▷ this is equal to TILE_SIZE
term ← 0.0
for l ← Zstart[n]+tiX, Zend[n] with step size ntX do
    k ← Zi2[l]
    i ← Zi3[l]
    term += u_k g_i Z[l]          ▷ multiply u and g coeff associated with l^th element of Z
end for
declare local array partialsum of length TILE_SIZE
partialsum[tiX] ← term                            ▷ save private result to local array
term ← reduction_sum_within_tile(partialsum)
```

iteration, l is updated with a step size equal to the number of work-items in the work-group. Finally, after each work-item has saved its result in an array stored in local memory, the elements of the array are summed together via a simple parallel reduction routine.

5.1 Tricks and Workarounds

Programming GPUs comes with the added challenge that OpenCL (and CUDA) currently do not support Fortran [30]. It is, however, possible for Fortran to call C functions. Since OpenCL readily supports host code written in C, functions in C can easily act as a staging area between Fortran and OpenCL. This is done by first giving C access to data allocated in Fortran, which is achieved by creating pointers to that data and sending them as arguments to a C function. Since Fortran natively sends pointers rather than the data itself, this is as simple as writing the first element of an array as the argument to a C function, which C then receives as a pointer to an array contiguous in memory.

Passing more complex data structures, such as arrays nested within arrays of derived data types, is more complicated, but still manageable. Because Fortran pads arrays in such a way that can be difficult to predict, it is simplest to send to C a pointer to the start of each array. This process can be made more compact by defining a derived data type in Fortran containing only a pointer, thereby allowing Fortran to generate an array of pointers (which is not natively available). A pointer to the first element of this array of pointers is then sent to C, which allows C to find the data associated with each variable within an array of derived data types.

Finally, OpenCL does not accept multidimensional arrays. To avoid bulky or obscure code, multidimensional arrays are sent to the GPU as 1D arrays (so long as they are contiguous in memory), and a macro is defined on the OpenCL side to simulate multidimensional behavior.

Chapter 6

Code Verification and Results

Verification seeks to answer the question "Are the equations being solved correctly?". There are a multitude of approaches to verify that a given set of methods, algorithms, and code are correctly solving the governing equations. Validation, which seeks to answer "Are the correct equations being solved?", has been performed extensively on the governing Navier-Stokes equations and is therefore not considered here.

For this study, a testing suite of five techniques is developed. Several of these tests are classic problems, while some are unique. The method of manufactured solutions is used to give a rigorous verification that PDEs are being correctly solved and individual terms are being handled correctly by the RKDG method, but does not verify the CLS method. Zalesak's disk is used to assess the ability of RKDG-CLS to maintain sharp corners. Columns and spheres in reversible velocity fields are used to demonstrate the RKDG-CLS scheme's ability to maintain long, thin ligaments. An analytical solution, called the circle test, is developed to assess the ability of reinitialization to handle a simple situation without modifying the PDE. Currently, the fast sweeping method is being developed and tested against the circle test. In the future, more verification will be necessary to demonstrate the accuracy of the FSM.

In all of these cases, grid convergence tests are also presented to demonstrate the $k+1$ convergence rate for the scheme and compare to WENO methods. Finally, the acceleration provided by GPU hardware is examined.

6.1 The Method of Manufactured Solutions

The method of manufactured solutions (MMS) was originally developed by Salari and Knupp [26] at Sandia National Laboratory. An excellent overview of the method is given by Roache [25]. The motivation for MMS lies in the difficulty of obtaining analytical so-

lutions to the governing equations to many physical systems, fluid dynamics included. In some cases, an analytical solution is found by making simplifying assumptions or requiring certain geometry. However, it is typically desired to apply computational methods and simulations to complex scenarios in practice, making more robust verification techniques essential.

MMS provides this rigorous technique by modifying the governing PDE with an additional source term. Since no methods exist for analytically solving these nonlinear governing PDEs in general, there are no general solutions to test the numerical solver against. So, MMS proposes that instead of attempting to find a solution the problem, you modify the problem to match a solution of your choosing. This means that the solution variables G, $\boldsymbol{u}$, and $\hat{\boldsymbol{n}}$ become arbitrary and may be prescribed. Then, the governing PDE is modified with a source term $Q(\boldsymbol{x}, t)$, which is computed exactly from the chosen solution variable fields. The source function is then projected into the discontinuous basis via Eq. (3.2).

$$\begin{gathered} F\left(G, \frac{\partial G}{\partial t}, \frac{\partial G}{\partial x}, \ldots\right) = 0 \quad \rightarrow \quad F\left(G, \frac{\partial G}{\partial t}, \frac{\partial G}{\partial x}, \ldots\right) = Q(\boldsymbol{x}, t) \\ Q(\boldsymbol{x}, t) = F\left(G_{\text{ex}}, \frac{\partial G_{\text{ex}}}{\partial t}, \frac{\partial G_{\text{ex}}}{\partial x}, \ldots\right) \end{gathered} \tag{6.1}$$

This work focuses on time-independent exact solutions. The initial condition for MMS is chosen to be $G = 0$ everywhere, and the level set scalar G converges over time to the exact solution. Dirichlet boundary conditions are typically selected, assigning the exact solution there. However, periodic boundary conditions may also be enforced if the solution is also periodic. In practice, it is found that this often results in prohibitively long simulations.

Advection

To test the advection equation, Eq. (2.3), with MMS, it is modified with a source term.

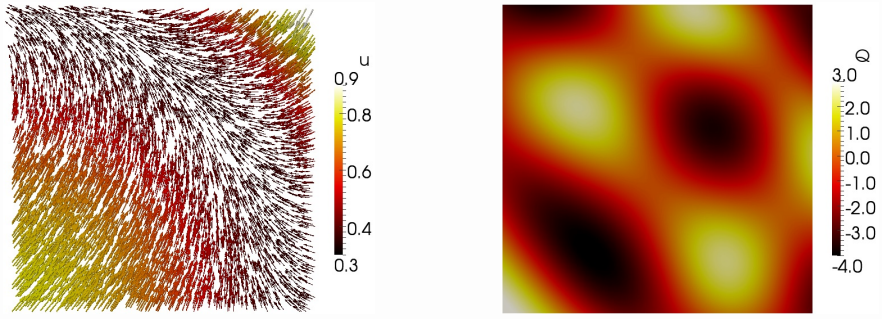
$$\frac{\partial G}{\partial t} + \nabla \cdot (G\boldsymbol{u}) = Q(\boldsymbol{x}, t)\,. \tag{6.2}$$

The source term is evaluated from the exact solution and prescribed velocity field:

$$Q(\boldsymbol{x}, t) = \frac{\partial G_{\text{ex}}(\boldsymbol{x}, t)}{\partial t} + \nabla \cdot \left(G_{\text{ex}}(\boldsymbol{x}, t)\, \boldsymbol{u}_{\text{ex}}(\boldsymbol{x}, t) \right). \tag{6.3}$$

Finally, RKDG scheme and code are tested on the unit-sized domain $[0, 1]^2$ with the following exact solution, prescribed velocity, and resulting source term:

$$\begin{aligned}
G_{\text{ex}}(x, y) &= \frac{1}{2} + \sin(2\pi x)\cos(2\pi y) \\
\boldsymbol{u}_{\text{ex}}(x, y) &= \left(\frac{1}{2} - \sin\left(x^2 + y^2\right) \right) \hat{\boldsymbol{x}} + \left(\cos\left(x^2 + y^2\right) - \frac{2}{5} \right) \hat{\boldsymbol{y}} \\
\implies Q(x, y) &= -2\cos\left(x^2 + y^2\right) x \left(1/2 + \sin\left(2\pi x\right)\cos\left(2\pi y\right)\right) \\
&\quad + 2\left(0.5 - \sin\left(x^2 + y^2\right)\right)\cos\left(2\pi x\right)\pi\cos\left(2\pi y\right) \\
&\quad - 2\sin\left(x^2 + y^2\right) y \left(1/2 + \sin(2\pi x)\cos(2\pi y)\right) \\
&\quad - 2\left(\cos\left(x^2 + y^2\right) - 0.4\right)\sin(2\pi x)\sin(2\pi y)\,\pi
\end{aligned} \tag{6.4}$$



(a) MMS velocity field $\boldsymbol{u}$ (b) MMS source term Q

Figure 6.1: MMS Advection Test

The solution as it evolves through time via RKDG with $k = 4$ polynomials is shown in Fig. 6.2 along with the error at the final converged state at $t = 4.3$. Comparing the final error to the velocity field in Fig. 6.1, it is found that the error predominantly lies in compressive regions where velocity vectors converge.

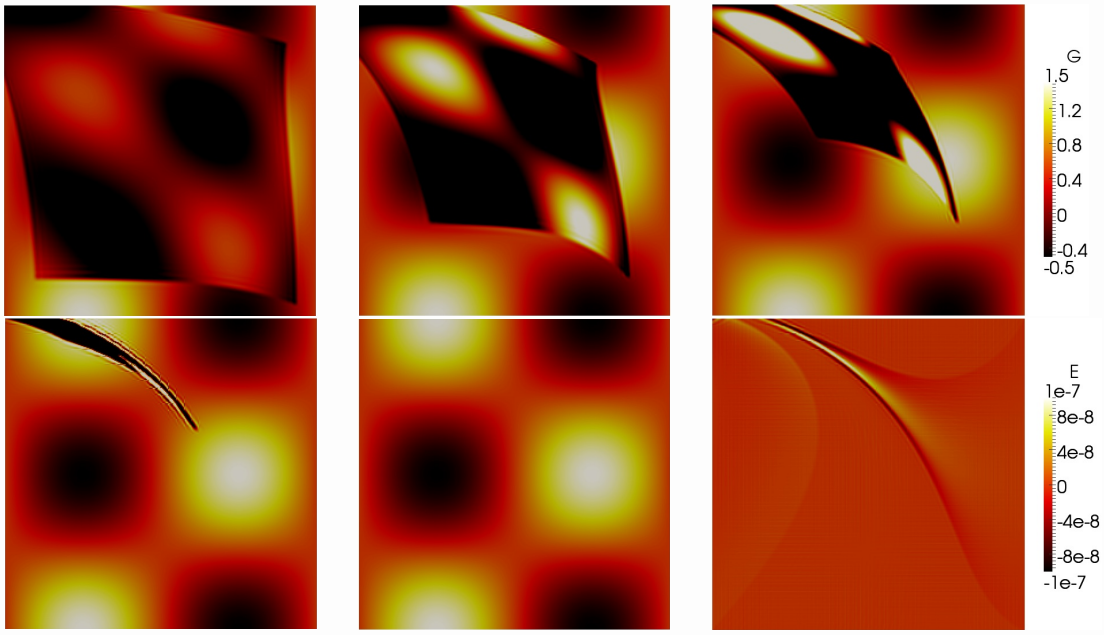


Figure 6.2: Solution G of MMS test case for $\Delta x = 1/40$, RKDG-4 for $t = 0.2, 0.5, 1.0, 2.0, 4.3$ time units, and error E at steady state (from top left to bottom right).

Table 6.1: Error norms of advection MMS test case and their order of convergence under grid refinement for RKDG-4.

Δx	L_∞	order	L_1	order
1/10	2.65e-5	-	3.37e-6	-
1/20	2.13e-6	3.6	1.03e-7	5.0
1/40	1.16e-7	4.2	3.32e-9	5.0
1/80	6.32e-9	4.2	1.07e-10	5.0
1/160	3.95e-10	4.0	3.43e-12	5.0

The errors produced for $k = 4$ are listed in Table 6.1. The results show a $k + 1$ order convergence rate in the L_1 norm with only a k^{th} order convergence rate in the L_∞ norm.

The reduced convergence for L_∞ requires further study, but can possibly be attributed to the upwind scheme or convergent velocity field.

Reinitialization

The reinitialization equation, Eq. (2.5), is modified with a source term as below:

$$\frac{\partial G}{\partial \tau} + \nabla \cdot (G\,(1-G)\,\hat{\boldsymbol{n}}) = \nabla \cdot (\varepsilon\,(\nabla G \cdot \hat{\boldsymbol{n}})\,\hat{\boldsymbol{n}}) + Q(\boldsymbol{x}, t)\,. \tag{6.5}$$

It should be noted that MMS does not require the normal vector to conform in any sense to the normal of any surface in this system, since it does not verify CLS. As such, there is also no need for it to be normalized. The source term is evaluated from the exact solution and prescribed normal vector field:

$$\begin{aligned} Q(\boldsymbol{x}, t) \;=\; & \frac{\partial G_{\text{ex}}(\boldsymbol{x}, \tau)}{\partial \tau} + \nabla \cdot \left(G_{\text{ex}}(\boldsymbol{x}, \tau) \left(1 - G_{\text{ex}}(\boldsymbol{x}, \tau)\right) \hat{\boldsymbol{n}}_{\text{ex}}(\boldsymbol{x}, \tau) \right) \\ & - \nabla \cdot \left(\varepsilon \left(\nabla G_{\text{ex}}(\boldsymbol{x}, \tau) \cdot \hat{\boldsymbol{n}}_{\text{ex}}(\boldsymbol{x}, \tau) \right) \hat{\boldsymbol{n}}_{\text{ex}}(\boldsymbol{x}, \tau) \right). \end{aligned} \tag{6.6}$$

The RKDG scheme for reinitialization was tested on the unit sized domain $[-0.5, 0.5]^2$ with the following exact solution and velocity:

$$\begin{aligned} G_{\text{ex}}(x, y) \;&=\; \frac{1}{2}\left(1 + \cos(2\pi x)\cos(2\pi y)\right) \\ \hat{\boldsymbol{n}}_{\text{ex}}(x, y) \;&=\; \sin(2\pi\,(x+y))\,\hat{\boldsymbol{x}} + \cos(2\pi\,(x+y))\,\hat{\boldsymbol{y}}\,. \end{aligned} \tag{6.7}$$

Here the source term is much longer and therefore omitted, although it is easily derived from Eq. (6.6). The diffusivity constant ε is chosen to be 0.08 so that the amplitudes of convective and diffusive terms are equal.

The solution as it evolves through time via RKDG with $k = 1$ polynomials is shown in Fig. 6.4 along with the error at the final converged state at $t = 2.8$. Comparing the final error to the normal field in Fig. 6.3, it is found that, similar to advection, the error predominantly lies in compressive regions where normal vectors converge.

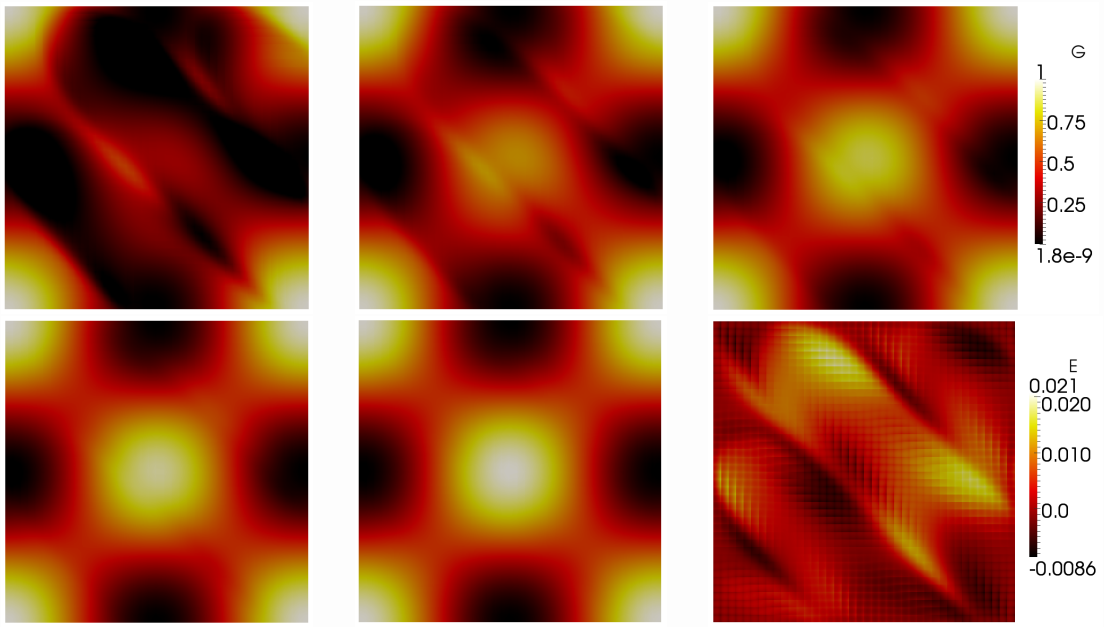
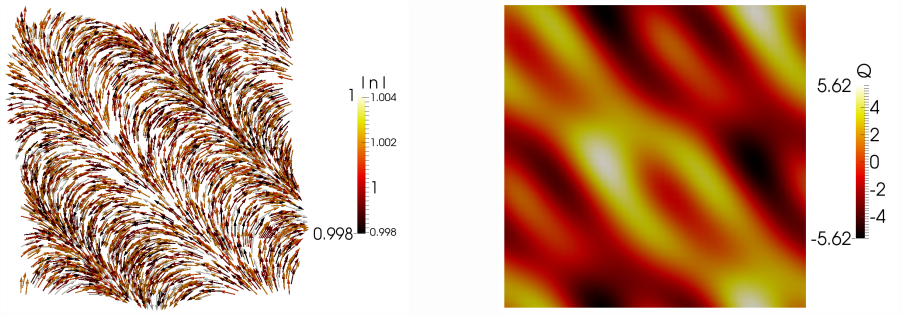


Figure 6.4: Solution G of reinitialization MMS test case for $\Delta x = 1/40$, RKDG-1 for $t = 0.1, 0.4, 0.7, 1.0, 2.8$ time units, and error E at steady state (from top left to bottom right).



(a) MMS normal field $\hat{\boldsymbol{n}}$

(b) MMS source term Q

Figure 6.3: MMS Reinitialization Test

The results for $k = 1$ are listed in Table 6.2 and show a convergence rate that approaches $k+1$ in both the L_1 and L_∞ norms. Tests for higher k and finer grids are expensive to perform because the diffusive CFL restriction, Eq. (3.24), and the requirement that ϵ remain constant for grid refinement studies causes the scheme to become prohibitively expensive. Modifying the scheme to an explicit-implicit time stepping method similar to

Table 6.2: Error norms of reinitialization MMS test case and their order of convergence under grid refinement for RKDG-1.

Δx	L_∞	order	L_1	order
1/10	1.24e-1	-	2.75e-2	-
1/20	7.73e-2	0.68	1.11e-2	1.32
1/40	3.15e-2	1.30	3.57e-3	1.63
1/80	1.02e-2	1.63	1.02e-3	1.80

the fractional-step method may be used to prevent this, however, and is left to future work. Furthermore, performing reinitialization on GPU hardware will lessen the computational cost.

6.2 Solid Body Rotations

Zalesak's disk [32], involves the solid body rotation of a notched disk. This test problem is widely used to evaluate the ability of a level set method to maintain sharp corners. A disk of radius 0.15, notch width 0.05, and notch height 0.25 is placed in a unit-sized domain at (0.5,0.75). The disk is then rotated about the origin by the velocity field

$$\boldsymbol{u}(x, y) \;=\; (0.5 - y)\,\hat{\boldsymbol{x}} + (x - 0.5)\,\hat{\boldsymbol{y}}$$

Fig. 6.5 shows the shape of the interface for WENO-5 and RKDG-CLS at $t = 2\pi$, i.e., after one full rotation. As shown, the RKDG-CLS-4 results are vastly superior, even with the same number of degrees of freedom. In fact, even after 45 rotations the RKDG-CLS-4 preserves the shape of Zalesak's disk far better than WENO-5.

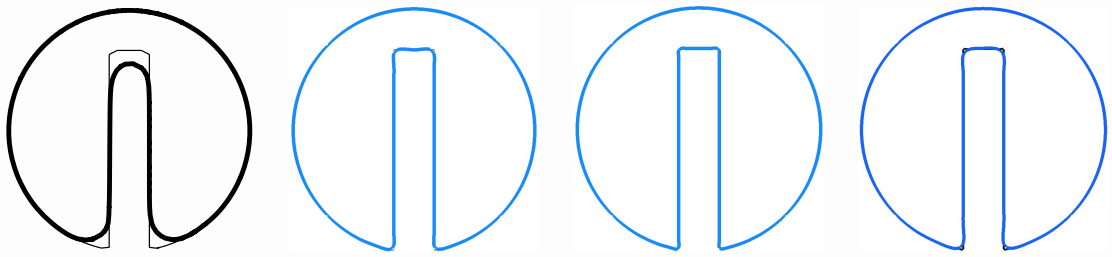


Figure 6.5: Zalesak's disk. From left to right: LS-WENO-5 ($\Delta x = 1/100$ & 1 rotation) [12]; RKDG-CLS-4 ($\Delta x = 1/50$ & 1 rotation); RKDG-CLS-4 ($\Delta x = 1/100$ & 1 rotation); RKDG-CLS-4 ($\Delta x = 1/50$ & 45 rotations). Exact solution shown as a thin line.

Table 6.3 summarizes the shape error, defined as

$$E = \frac{\int_A \left| H(G) - H\left(G_{\text{ex}}\right) \right| \, \mathrm{d}A}{\int_A H\left(G_{\text{ex}}\right) \, \mathrm{d}A} \tag{6.8}$$

and evaluated employing a recursive cell refinement algorithm using marching triangles to calculate the phase interface position, and G_{ex} denoting the exact solution, for different RKDG-CLS-k. Overall shape errors are small, however, the convergence rate in this metric appears to approach first order, independent of the order of the employed RKDG basis functions. This appears to be due to the fact that shape errors for the RKDG-CLS methods are confined to the sharp corner regions that represent a discontinuity in the solution gradients and are thus captured with the employed Legendre basis functions at best with first order. It should be noted though that even if the convergence rates appear be first order for all analyzed k, increasing k with a fixed Δx reduces errors significantly. Of course increasing k increases the numerical cost of the scheme. The computational cost C is approximated as proportional to the product of the number of degrees of freedom $N^2 (k+1)^2$, the required time steps per time unit due to the CFL restriction $2k+1$, and the number of operations per coefficient update in Eq. (3.9) k:

$$C \sim N^2 (k+1)^2 (2k+1) \, k \, . \tag{6.9}$$

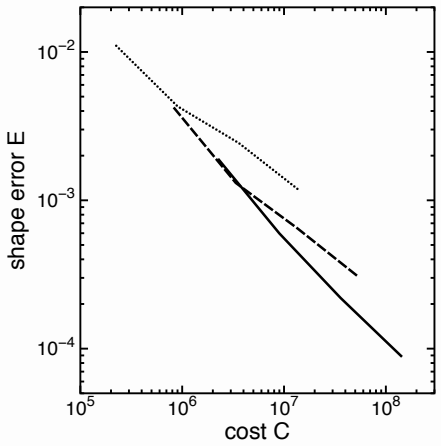


Figure 6.6: Shape error E as a function of scheme cost C; RKDG-CLS-2 (dotted line), RKDG-CLS-3 (dashed line), RKDG-CLS-4 (solid line).

Fig. 6.6 shows that increasing the order k of the scheme is preferable to increasing only the number of mesh points per spatial direction N, even in a scenario where the error is being dominated by discontinuities in the solution gradient and thus the full $k+1$ convergence rate of the RKDG-CLS scheme is not obtainable.

Table 6.3: Zalesak's disk shape errors and order of convergence for RKDG-CLS-k method after one full rotation.

Δx	$k=2$		$k=3$		$k=4$	
	E	order	E	order	E	order
1/50	1.10e-2	-	4.16e-3	-	1.91e-3	-
1/100	4.30e-3	1.4	1.31e-3	1.7	5.98e-4	1.7
1/200	2.43e-3	0.8	6.50e-4	1.0	2.19e-4	1.4
1/400	1.16e-3	1.1	3.05e-4	1.1	8.81e-5	1.3

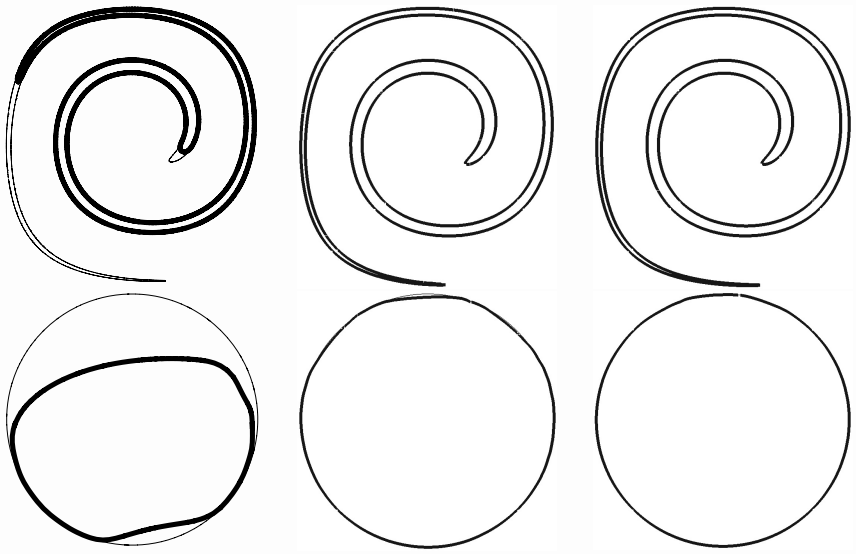


Figure 6.7: Interface shape of column in a deformation field at $t = T/2$ (top row) and $t = T$ (bottom row); from left to right: LS-WENO-5 with $\Delta x = 1/128$ [12], RKDG-CLS-4 method with $\Delta x = 1/64$ and $\Delta x = 1/128$. Thin line marks reference solution.

6.3 Reversible Velocity Fields

Column in a Deformation Field

The column or circle in a deformation field problem introduced by Bell et al. [1] and applied as a level set test problem by Enright et al. [9] tests the ability of the level set method to resolve and maintain ever thinner filaments. A column of radius $R_0 = 0.15$ and center $(0.5, 0.75)^T$ is placed inside a unit sized box. The velocity field is given by the stream function

$$\Psi(\boldsymbol{x}, t) = \frac{1}{\pi} \sin^2(\pi x) \sin^2(\pi y) \cos(\pi t/T) \tag{6.10}$$

with $T = 8$ and first stretches the column into ever thinner filaments that are wrapped around the center of the box, then slowly reverses, and pulls the filaments back into the initial circular shape.

Fig. 6.7 shows the interface shape at the moment of maximum extension $t = T/2$ and

after full flow reversal at $t = T$, for the LS-WENO-5 scheme using $\Delta x = 1/128$ [12], and RKDG-CLS-4 using $\Delta x = 1/64$ respective $\Delta x = 1/128$. The RKDG-CLS-4 method gives clearly superior results, is able to sustain the trailing filament well, and recovers the exact solution of a circle well even on a twice coarser mesh as the LS-WENO-5 method. Finally, Table 6.4 shows the shape error at $t = T$ as a function of grid spacing Δx for both the RKDG-CLS-3 and RKDG-CLS-4 methods.

Table 6.4: Shape errors and order of convergence for RKDG-CLS-k method after full flow reversal at $t = T$.

Δx	$k = 3$		$k = 4$	
	E	order	E	order
1/64	1.65e-2	-	7.98e-2	-
1/128	3.91e-3	2.1	2.01e-3	2.0
1/256	1.26e-3	1.6	6.22e-4	1.7

Sphere in a Deformation Field

To demonstrate the performance of the RKDG-CLS method in three dimensions, the sphere in a deformation field case proposed by [9] is performed. A sphere of radius $R_0 = 0.15$ is placed at $(0.35, 0.35, 0.35)^T$ inside a unit box, whose time dependent velocity field is given by

$$\begin{aligned} u &= 2\sin^2(\pi x)\sin(2\pi y)\sin(2\pi z)\cos(\pi t/T) \\ v &= -\sin(2\pi x)\sin^2(\pi y)\sin(2\pi z)\cos(\pi t/T) \\ w &= -\sin(2\pi x)\sin(2\pi y)\sin^2(\pi z)\cos(\pi t/T)\ , \end{aligned} \tag{6.11}$$

with $T = 3$. Fig. 6.8 shows the interface shape at $t = T/2$, the time of maximum deformation, and $t = T$ after full flow reversal, for the LS-WENO-5 method using $\Delta x = 1/128$ and RKDG-CLS-4 using $\Delta x = 1/32$ respective $\Delta x = 1/128$. Again, RKDG-CLS-4 yields

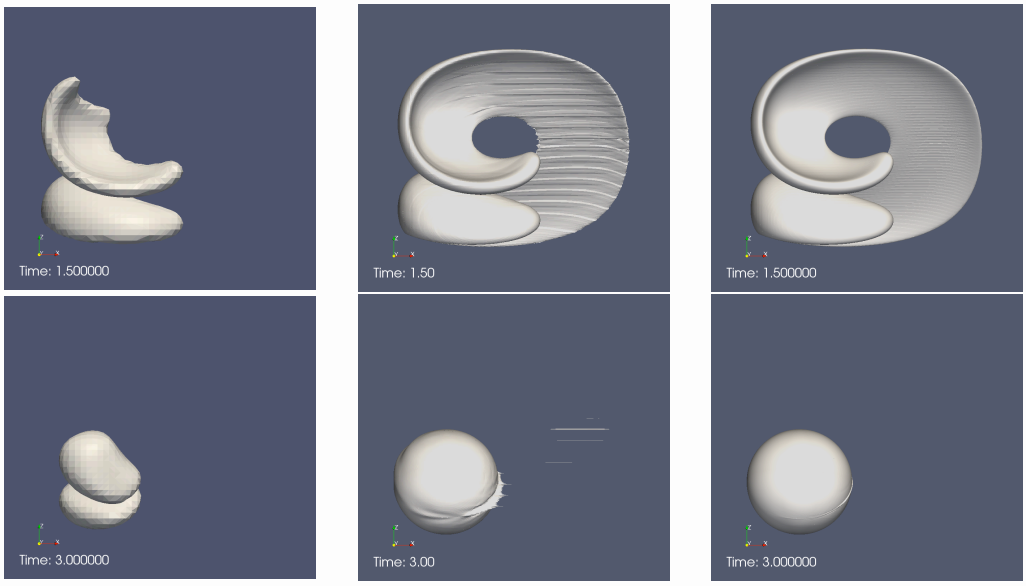


Figure 6.8: Sphere in a deformation field interface shape at $t = T/2$ (top row) and $t = T$ (bottom row); from left to right: LS-WENO-5 with $\Delta x = 1/128$, RKDG-CLS-4 with $\Delta x = 1/32$ and $\Delta x = 1/128$.

superior results, even on a four time coarser mesh compared to the LS-WENO-5 method. Even without reinitialization of the RKDG-CLS method, volume conservation is significantly improved compared to LS-WENO-5. Whereas the latter loses 27.4% of volume at $t = T$ using $\Delta x = 1/128$, the former loses only 0.27% using $\Delta x = 1/32$.

6.4 Circle Test

To assess reinitialization and the fast sweeping method, a test case was developed which involves a circle of radius R_0 placed at the origin of a unit-sized $[-0.5, 0.5]^2$ domain. The level set scalar is initialized to

$$G(\boldsymbol{x}) = \frac{1}{2}\left(\tanh\left(\frac{R_0 - \sqrt{x^2 + y^2}}{2\varepsilon_0}\right) + 1\right). \tag{6.12}$$

Reinitialization then sharpens the interface from thickness $\varepsilon_0 \to \varepsilon$, the latter of which is used in Eq. (2.5). As the circle is refined, the 0.5-isosurface is transported outward in order to conserve G. The new radius R can be computed by assuming that G is transported only

in the set normal direction and not tangent to the interface. Then, the integral over G from $r = 0, \infty$ remains unchanged under reinitialization and R can be computed in terms of R_0 , ε_0 , and ε.

$$\int_0^\infty G(r, R, \varepsilon)\ \mathrm{d}r = \int_0^\infty G\left(r, R_0, \varepsilon_0\right)\ \mathrm{d}r$$

The results of a refinement study for $k = 3$ polynomials are shown in Table 6.5, which shows the L_∞ and L_1 norms of the error at steady state together with the associated orders of convergence. Reinitialization is evaluated for a circle of initial radius $R_0 = 0.25$ and thickness $\varepsilon_0 = 0.025$ refined to $\varepsilon = 0.0125$. The final radius R is found to be 0.253063 from Eq. (6.4). As expected, the L_1 and L_∞ norms of the error converge with $k + 1$ order.

Table 6.5: Error norms of circle test and their order of convergence under grid refinement for RKDG-CLS-3.

Δx	L_∞	order	L_1	order
1/20	5.47e-2	-	3.24e-3	-
1/40	3.32e-3	4.04	1.68e-4	4.27
1/80	1.84e-4	4.17	9.40e-6	4.16

6.5 GPU Acceleration

The OpenCL algorithm for DG advection was executed on a Nvidia Tesla C2050 GPU (with a work-group size of 128) and compared to the original algorithm running in serial on a a 1.9GHz AMD Opteron 6186 CPU. Both algorithms take advantage of sparsity and are implemented on equidistant Cartesian meshes in unit sized domains. Verification of the method has been performed for the CPU algorithm via MMS, time-reversing velocity fields, and the circle test as described in previous sections of this chapter, so this test is limited to compute times and assurance that the CPU and GPU give equivalent results (within 10^5 times machine epsilon at double precision). As such, the test problem is arbi-

trary. For robustness, the solution variable coefficients are randomized, and activating or deactivating terms of the equation can be used for debugging and to further demonstrate that each term is being evaluated correctly.

These tests produce several interesting trends, shown in Table 6.6. First, low degree polynomials show little benefit from the GPU, and sometimes even slower runtimes. This is simply because of the parallelization scheme, where work for a single $g_{n,\kappa}$ coefficient is delegated to across threads within a work-group. If there are not enough terms for all of the work-items, the benefit of parallelization on the GPU is lost since some of the threads then do no work. One way to remedy this in practice is to use smaller work-group sizes when dealing with smaller polynomials. However, this solution has limitations since GPUs are most efficient when the work-group size is a multiple of 32 [20]. A similar drawback arises if sparsity is not exploited, where the GPU sees a ∼2x slow-down for 3^{rd} order polynomials. This results from parallelizing along the tensor multiplication loop, where many threads end up multiplying zeros together and appending them to a sum, again wasting effort.

Second, the data indicates the GPU is increasingly advantageous as it is given more work. With more degrees of freedom and operations, whether from refining the grid or increasing the number of polynomials, the total speedup increases. This reflects the streaming memory model on the GPU, where floating-point operations are almost free, and compliments the effectiveness of high-order DG.

To investigate the expense of different operations on the GPU, OpenCL event timers are used to rank routines by their duration. The results are shown in Table 6.7 for the degree 3 polynomial, 40x40x40 grid case.

Table 6.6: Results for Compute Time of One RK-Step

Polynomial Degree	$1/\Delta x$	2D Simulation			3D Simulation		
		CPU time (s)	GPU time (s)	Speedup	CPU time (s)	GPU time (s)	Speedup
1	10	6.37e-4	2.12e-3	0.30x	2.25e-2	6.96e-3	3.23x
	20	2.52e-3	3.11e-3	0.81x	1.79e-1	3.99e-2	4.49x
	40	9.47e-3	6.28e-3	1.51x	6.18e-1	2.83e-1	2.18x
2	10	2.45e-3	2.45e-3	1.00x	3.24e-1	2.56e-2	12.7x
	20	9.63e-3	4.57e-3	2.11x	1.18	1.41e-1	8.37x
	40	3.85e-2	1.15e-2	3.35x	8.82	1.05	8.40x
3	10	8.81e-3	2.87e-3	3.07x	1.08	8.30e-2	13.0x
	20	3.38e-2	6.52e-3	5.18x	8.34	6.20e-1	13.5x
	40	1.34e-1	1.89e-2	7.09x	6.67e+1	4.78	14.0x
4	10	3.47e-2	4.34e-3	8.00x	1.15e+1	4.16e-1	27.6x
	20	1.38e-1	1.03e-2	13.4x	1.32e+2	3.03	43.6x
	40	3.92e-1	3.06e-2	12.8x	1.36e+3	2.40e+1	56.7x

Table 6.7: GPU Event Timing

Event	Time (ms)
Kernel Create	0.1628
Data Send	336.9
Compute	4763
Data Receive	20.14
Total	4784

Note: these times may overlap, so the total compute time is not necessarily the sum of event times.

As perhaps an unexpected result, compute time overwhelmingly dominates the execution time. In other applications, the memory transfer overhead between the CPU and GPU take up a significant portion of the runtime. However, this case involves a high work to data ratio, since the scheme requires numerous arithmetic operations relative to the amount of relevant calculated data, especially at higher orders. As a result, optimizations that focus on decreasing compute time and internal memory operations are more beneficial than memory transfer optimizations, contrary to the usual case for GPU algorithms.

Chapter 7

Concluding Remarks

7.1 Summary

In order to construct a predictive numerical laboratory for multiphase flows, an arbitrary-order, nearly quadrature-free, discontinuous Galerkin, conservative level set approach is presented for modeling interface transport and topology evolution. This involves solving the advection equation, Eq. (2.3), to transport the interface and reinitialization, Eq. (2.5), to maintain the hyperbolic tangent profile, Eq. (2.4), in a mass conserving fashion. These two partial differential equations are discretized spatially by performing a spectral decomposition within cells, via Eq. (3.1). Numerical fluxes are handled using Eqs. (3.8), (3.10), and (3.14). The result is two sets of systems of ODEs, Eq. (3.9) for advection and Eq. (3.20) for reinitialization. These systems are stepped through time using a $k+1$ stage Runge-Kutta method, Eq. (3.21), which is subject to CFL constraint Eqs. (3.22), (3.23), and (3.24). The RKDG-CLS scheme for advection is executed on the GPU using OpenCL, via Alg. 1. Finally, normal vector fields for the interfaces are constructed using the fast sweeping method, which involves calculating the signed distance function by first inverting the hyperbolic tangent profile within the $\mathscr{A}$-band via Eq. (4.1). Outside the $\mathscr{A}$-band, the solution is sweeped out through alternating loop orderings by a first-order FD-based Godunov solver, Eq. (4.7), which depends on variables selected by causality via Eq. (4.6) and initializes causality flags via Eq. (4.8). This solution is feeded into an arbitrary-order DG solver, which finds the roots of Eq. (4.12) via Newton's iterative method Eq. (4.1) where the Jacobian is calculated by Eq. (4.15). The resulting signed distance function is differentiated by Eq. (4.19) and the normal vector field is finally computed via Eq. (4.2). Ap. A lists integrals that are precomputed and stored in arrays.

This work has demonstrated that taking advantage of sparsity, whenever possible, is

crucial to developing efficient algorithms, especially on the GPU. CRS contributed to an overall speedup of nearly 60x between GPU/CPU, advocating the applicability and benefit of GPUs in numerical algorithms, especially for independent segments of code that benefit from parallelism. Table 6.6 indicated that more work given to the GPU results in more speedup, especially when increasing polynomial order. This compliments the results in Fig. 6.6, showing that the discontinuous Galerkin conservative level set method is more effective at higher orders. Therefore, accelerating that method via GPUs reaps benefits from multiple angles, making it an excellent candidate for high order interface capturing and modeling atomization.

7.2 Future Work

Future work involves development of several algorithms and techniques. Normal vectors and curvature must be calculated, which involves completing the arbitrary-order DG fast sweeping method and developing a routine to compute curvature, along with verification for both. Future work will also involve improving the GPU implementation (e.g., utilizing local memory and ensuring memory alignment) and porting reinitialization, Eq. (2.5), to OpenCL. More work on the computer science side of things can be done to execute the RKDG-CLS method on a cluster of GPUs in parallel. Finally, coupling the entire scheme to a parallel flow solver, ideally a DG-based one, would present a complete fluid simulation for engineering applications involving multiphase flows.

References

[1] J. B. Bell, P. Colella, and H. M. Glaz. "A second-order projection method for the incompressible Navier Stokes equations". *J. Comput. Phys.* 85 (1989), pp. 257–283.

[2] M. Boué and P. Dupuis. "Markov chain approximations for deterministic control problems with affine dynamics and quadratic cost in the control". *SIAM J. Numer. Anal* 36 (1998), pp. 667–695.

[3] Y. Cheng and C.-W. Shu. "A discontinuous Galerkin finite element method for directly solving the Hamilton-Jacobi equations". *J. Comput. Phys.* 223 (2007), pp. 398–415.

[4] D.L. Chopp. "Computing minimal surfaces via level set curvature flow". *J. Comput. Phys.* 106 (1993), pp. 77–91.

[5] B. Cockburn and C.-W. Shu. "Runge–Kutta discontinuous Galerkin methods for convection-dominated problems". *J. Sci. Comput.* 16 (2001), pp. 173–261.

[6] M. F. Czajkowski and O. Desjardins. "A discontinuous galerkin conservative level set scheme for simulating turbulent primary atomization". *ILASS Americas 23rd Annual Conference on Liquid Atomization and Spray Systems* (2011).

[7] O. Desjardins, V. Moureau, and H. Pitsch. "An accurate conservative level set/ghost fluid method for simulating turbulent atomization". *J. Comput. Phys.* 227 (2008), pp. 8395–8416.

[8] I. Duff, R. Grimes, and J. Lewis. *User's Guide for the Harwell-Boeing Sparse Matrix Collection (Release I).* 1992.

[9] D. Enright et al. "A hybrid particle level set method for improved interface capturing". *J. Comput. Phys.* 183 (2002), pp. 83–116.

[10] S. Gottlieb. "On high order strong stability preserving Runge-Kutta and multi step time discretizations". *J. Sci. Comput.* 25 (2005), pp. 105–128.

[11] J.D. Griggs. *Pahoeoe Fountain*. 2007. URL: http://commons.wikimedia.org/wiki/File:Pahoeoe_fountain_edit2.jpg.

[12] M. Herrmann. "A balanced force refined level set grid method for two-phase flows on unstructured flow solver grids". *J. Comput. Phys.* 227 (2008), pp. 2674–2706.

[13] P. LeSaint and P. A. Raviart. "On a finite element method for solving the neutron transport equation". In: *Mathematical Aspects of Finite Elements in Partial Differential Equations*. Ed. by C. de Boor. Academic Press, NY, 1974, pp. 89–123.

[14] F. Li et al. "A second order discontinuous Galerkin fast sweeping method for Eikonal equations". *J. Comput. Phys.* 227 (2008), pp. 8191–8208.

[15] F. Lörcher, G. Gassner, and C.-D. Munz. "An explicit discontinuous Galerkin scheme with local time-stepping for general unsteady diffusion equations". *J. Comput. Phys.* 227 (2008), pp. 5649–5670.

[16] H. Luo et al. "A reconstructed discontinuous Galerkin method for the compressible Navier-Stokes equations on arbitrary grids". *J. Comput. Phys.* 229 (2010), pp. 6961–6978.

[17] S. Luo. "A uniformly second order fast sweeping method for Eikonal equations". *J. Comput. Phys.* 241 (2013), pp. 104–117.

[18] E. Marchandise, J.-F. Remacle, and N. Chevaugeon. "A quadrature-free discontinuous Galerkin method for the level set equation". *J. Comput. Phys.* 212 (2006), pp. 338–357.

[19] NASA. *NASA's Terra Satellites Sees Spill on May 24*. 2010. URL: http://www.nasa.gov/topics/earth/features/oilspill/20100525_spill.html.

[20] *NVIDIA OpenCL Best Practices Guide*. ver. 1.0. NVIDIA Corporation. 2009.

[21] E. Olsson and G. Kreiss. "A conservative level set method for two phase flow". *J. Comput. Phys.* 210 (2005), pp. 225–246.

[22] E. Olsson, G. Kreiss, and S. Zahedi. "A conservative level set method for two phase flow II". *J. Comput. Phys.* 225 (2007), pp. 785–807.

[23] R.L. Panton. *Incompressible Flow*. Wiley, 2005.

[24] W. H. Reed and T. R. Hill. *Triangular mesh methods for the neutron transport equation*. Tech. rep. LA-UR-73-479. Los Alamos National Laboratory, 1973.

[25] P. J. Roache. "Code verification by the method of manufactured solutions". *J. Fluids Eng*. 124 (2002), pp. 4–10.

[26] K. Salari and P. Knupp. *Code verification by the method of manufactured solutions*. Tech. rep. SAND2000-1444. Sandia National Laboratory, 2000.

[27] M. Scarpino. *OpenCL in Action: How to Accelerate Graphics and Computation*. Shelter Island, NY: Manning Publications Co., 2012.

[28] P. Spiekermann et al. "Experimental and numerical investigation of common-rail ethanol sprays at diesel engine-like conditions". *ILASS Americas 20th Annual Conference on Liquid Atomization and Spray Systems* (2007).

[29] M. Sussman, P. Smereka, and S. Osher. "A level set approach for computing solutions to incompressible two-phase flow". *J. Comput. Phys.* 114 (1994), pp. 146–159.

[30] *The OpenCL Specification*. ver. 1.2. Khronos Group, Inc. 2011.

[31] Y.-H.R. Tsai et al. "Fast sweeping algorithms for a class of Hamilton-Jacobi equations". *SIAM J. on Numer. Anal.* 41 (2003), pp. 673–694.

[32] S. T. Zalesak. "Fully multidimensional flux-corrected transport algorithms for fluids". *J. Comput. Phys.* 31 (1979), pp. 335–362.

[33] Y.-T. Zhang et al. “Uniformly accurate discontinuous Galerkin fast sweeping methods for Eikonal equations”. *SIAM J. Sci. Comput.* 33 (2011), pp. 1873–1896.

Appendix A

Precomputed Integrals

A.1 Advection

For advection, the precomputed integrals are:

$$\mathrm{Ax}_{i,k,n} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} b_i b_k \frac{\partial b_n}{\partial \xi}\, \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.1}$$

$$\mathrm{Ay}_{i,k,n} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} b_i b_k \frac{\partial b_n}{\partial \eta}\, \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.2}$$

$$\mathrm{Az}_{i,k,n} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} b_i b_k \frac{\partial b_n}{\partial \zeta}\, \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.3}$$

$$\mathrm{SAxm}_{i,k,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(+1,\eta,\zeta)\, b_k(+1,\eta,\zeta)\, b_n(-1,\eta,\zeta)\ \mathrm{d}\eta\mathrm{d}\zeta \tag{A.4}$$

$$\mathrm{SAxp}_{i,k,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(-1,\eta,\zeta)\, b_k(-1,\eta,\zeta)\, b_n(-1,\eta,\zeta)\ \mathrm{d}\eta\mathrm{d}\zeta \tag{A.5}$$

$$\mathrm{SAym}_{i,k,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(\xi,+1,\zeta)\, b_k(\xi,+1,\zeta)\, b_n(\xi,-1,\zeta)\ \mathrm{d}\xi\mathrm{d}\zeta \tag{A.6}$$

$$\mathrm{SAyp}_{i,k,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(\xi,-1,\zeta)\, b_k(\xi,-1,\zeta)\, b_n(\xi,-1,\zeta)\ \mathrm{d}\xi\mathrm{d}\zeta \tag{A.7}$$

$$\mathrm{SAzm}_{i,k,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(\xi,\eta,+1)\, b_k(\xi,\eta,+1)\, b_n(\xi,\eta,-1)\ \mathrm{d}\xi\mathrm{d}\eta \tag{A.8}$$

$$\mathrm{SAzp}_{i,k,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(\xi,\eta,-1)\, b_k(\xi,\eta,-1)\, b_n(\xi,\eta,-1)\ \mathrm{d}\xi\mathrm{d}\eta \tag{A.9}$$

A.2 Reinitialization

Reinitialization requires the integrals precomputed for advection, plus:

$$\mathrm{Bx}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} b_i b_j b_k \frac{\partial b_n}{\partial \xi}\, \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.10}$$

$$\mathrm{By}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} b_i b_j b_k \frac{\partial b_n}{\partial \eta}\, \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.11}$$

$$\mathrm{Bz}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} b_i b_j b_k \frac{\partial b_n}{\partial \zeta}\, \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.12}$$

$$\mathrm{Cxx}_{i,k,l,n} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \frac{\partial b_i}{\partial \xi} b_k b_l \frac{\partial b_n}{\partial \xi}\, \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.13}$$

$$\mathrm{Cyy}_{i,k,l,n} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \frac{\partial b_i}{\partial \eta} b_k b_l \frac{\partial b_n}{\partial \eta}\, \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.14}$$

$$\mathrm{Czz}_{i,k,l,n} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \frac{\partial b_i}{\partial \zeta} b_k b_l \frac{\partial b_n}{\partial \zeta}\, \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.15}$$

$$\mathrm{Cxy}_{i,k,l,n} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} b_k b_l \left(\frac{\partial b_i}{\partial \xi} \frac{\partial b_n}{\partial \eta} + \frac{\partial b_i}{\partial \eta} \frac{\partial b_n}{\partial \xi} \right) \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.16}$$

$$\mathrm{Cxz}_{i,k,l,n} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} b_k b_l \left(\frac{\partial b_i}{\partial \xi} \frac{\partial b_n}{\partial \zeta} + \frac{\partial b_i}{\partial \zeta} \frac{\partial b_n}{\partial \xi} \right) \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.17}$$

$$\mathrm{Cyz}_{i,k,l,n} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} b_k b_l \left(\frac{\partial b_i}{\partial \xi} \frac{\partial b_n}{\partial \eta} + \frac{\partial b_i}{\partial \eta} \frac{\partial b_n}{\partial \xi} \right) \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.18}$$

$$\mathrm{Sxm}_{i,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(+1,\eta,\zeta)\, b_n(-1,\eta,\zeta)\ \mathrm{d}\eta\mathrm{d}\zeta \tag{A.19}$$

$$\mathrm{Sxp}_{i,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(-1,\eta,\zeta)\, b_n(-1,\eta,\zeta)\ \mathrm{d}\eta\mathrm{d}\zeta \tag{A.20}$$

$$\mathrm{Sym}_{i,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(\xi,+1,\zeta)\, b_n(\xi,-1,\zeta)\ \mathrm{d}\xi\mathrm{d}\zeta \tag{A.21}$$

$$\mathrm{Syp}_{i,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(\xi,-1,\zeta)\, b_n(\xi,-1,\zeta)\ \mathrm{d}\xi\mathrm{d}\zeta \tag{A.22}$$

$$\mathrm{Szm}_{i,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(\xi,\eta,+1)\, b_n(\xi,\eta,-1)\ \mathrm{d}\xi\mathrm{d}\eta \tag{A.23}$$

$$\mathrm{Szp}_{i,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(\xi,\eta,-1)\, b_n(\xi,\eta,-1)\ \mathrm{d}\xi\mathrm{d}\eta \tag{A.24}$$

$$\mathrm{SBxm}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(+1,\eta,\zeta)\, b_j(+1,\eta,\zeta)\, b_k(+1,\eta,\zeta)\, b_n(-1,\eta,\zeta)\ \mathrm{d}\eta\mathrm{d}\zeta \tag{A.25}$$

$$\mathrm{SBxp}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(-1,\eta,\zeta)\, b_j(-1,\eta,\zeta)\, b_k(-1,\eta,\zeta)\, b_n(-1,\eta,\zeta)\ \mathrm{d}\eta\mathrm{d}\zeta \tag{A.26}$$

$$\mathrm{SBym}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(\xi,+1,\zeta)\, b_j(\xi,+1,\zeta)\, b_k(\xi,+1,\zeta)\, b_n(\xi,-1,\zeta)\ \mathrm{d}\xi\mathrm{d}\zeta \tag{A.27}$$

$$\mathrm{SByp}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(\xi,-1,\zeta)\, b_j(\xi,-1,\zeta)\, b_k(\xi,-1,\zeta)\, b_n(\xi,-1,\zeta)\ \mathrm{d}\xi\mathrm{d}\zeta \tag{A.28}$$

$$\mathrm{SBzm}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(\xi,\eta,+1)\, b_j(\xi,\eta,+1)\, b_k(\xi,\eta,+1)\, b_n(\xi,\eta,-1)\ \mathrm{d}\xi\mathrm{d}\eta \tag{A.29}$$

$$\mathrm{SBzp}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1} b_i(\xi,\eta,-1)\, b_j(\xi,\eta,-1)\, b_k(\xi,\eta,-1)\, b_n(\xi,\eta,-1)\ \mathrm{d}\xi\mathrm{d}\eta \tag{A.30}$$

$$\mathrm{SCxX}_{i,j,k,n} = \frac{1}{2}\int_{-1}^{1}\int_{-1}^{1}\left.\frac{\partial b_i}{\partial \xi}\right|_{\xi=0} b_j(0,\eta,\zeta)\, b_k(0,\eta,\zeta)\, b_n(-1,\eta,\zeta)\ \mathrm{d}\eta\mathrm{d}\zeta \tag{A.31}$$

$$\mathrm{SCxY}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1}\left.\frac{\partial b_i}{\partial \eta}\right|_{\xi=0} b_j(0,\eta,\zeta)\, b_k(0,\eta,\zeta)\, b_n(-1,\eta,\zeta)\ \mathrm{d}\eta\mathrm{d}\zeta \tag{A.32}$$

$$\mathrm{SCxZ}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1}\left.\frac{\partial b_i}{\partial \zeta}\right|_{\xi=0} b_j(0,\eta,\zeta)\, b_k(0,\eta,\zeta)\, b_n(-1,\eta,\zeta)\ \mathrm{d}\eta\mathrm{d}\zeta \tag{A.33}$$

$$\mathrm{SCyX}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1}\left.\frac{\partial b_i}{\partial \xi}\right|_{\eta=0} b_j(\xi,0,\zeta)\, b_k(\xi,0,\zeta)\, b_n(\xi,-1,\zeta)\ \mathrm{d}\xi\mathrm{d}\zeta \tag{A.34}$$

$$\mathrm{SCyY}_{i,j,k,n} = \frac{1}{2}\int_{-1}^{1}\int_{-1}^{1}\left.\frac{\partial b_i}{\partial \eta}\right|_{\eta=0} b_j(\xi,0,\zeta)\, b_k(\xi,0,\zeta)\, b_n(\xi,-1,\zeta)\ \mathrm{d}\xi\mathrm{d}\zeta \tag{A.35}$$

$$\mathrm{SCyZ}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1}\left.\frac{\partial b_i}{\partial \zeta}\right|_{\eta=0} b_j(\xi,0,\zeta)\, b_k(\xi,0,\zeta)\, b_n(\xi,-1,\zeta)\ \mathrm{d}\xi\mathrm{d}\zeta \tag{A.36}$$

$$\mathrm{SCzX}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1}\left.\frac{\partial b_i}{\partial \xi}\right|_{\zeta=0} b_j(\xi,\eta,0)\, b_k(\xi,\eta,0)\, b_n(\xi,\eta,-1)\ \mathrm{d}\xi\mathrm{d}\eta \tag{A.37}$$

$$\mathrm{SCzY}_{i,j,k,n} = \int_{-1}^{1}\int_{-1}^{1}\left.\frac{\partial b_i}{\partial \eta}\right|_{\zeta=0} b_j(\xi,\eta,0)\, b_k(\xi,\eta,0)\, b_n(\xi,\eta,-1)\ \mathrm{d}\xi\mathrm{d}\eta \tag{A.38}$$

$$\mathrm{SCzZ}_{i,j,k,n} = \frac{1}{2}\int_{-1}^{1}\int_{-1}^{1}\left.\frac{\partial b_i}{\partial \zeta}\right|_{\zeta=0} b_j(\xi,\eta,0)\, b_k(\xi,\eta,0)\, b_n(\xi,\eta,-1)\ \mathrm{d}\xi\mathrm{d}\eta \tag{A.39}$$

$$\mathrm{Pr2to1Xm}_{i,n} = \int_{-1}^{0}\int_{-1}^{1}\int_{-1}^{1} b_i(2\xi+1,\eta,\zeta)\, b_n(\xi,\eta,\zeta)\ \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.40}$$

$$\mathrm{Pr2to1Xp}_{i,n} = \int_{0}^{1}\int_{-1}^{1}\int_{-1}^{1} b_i(2\xi-1,\eta,\zeta)\, b_n(\xi,\eta,\zeta)\ \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.41}$$

$$\mathrm{Pr2to1Ym}_{i,n} = \int_{-1}^{1}\int_{-1}^{0}\int_{-1}^{1} b_i(\xi,2\eta+1,\zeta)\, b_n(\xi,\eta,\zeta)\ \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.42}$$

$$\mathrm{Pr2to1Yp}_{i,n} = \int_{-1}^{1}\int_{0}^{1}\int_{-1}^{1} b_i(\xi,2\eta-1,\zeta)\, b_n(\xi,\eta,\zeta)\ \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.43}$$

$$\mathrm{Pr2to1Zm}_{i,n} = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{0} b_i(\xi,\eta,2\zeta+1)\, b_n(\xi,\eta,\zeta)\ \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.44}$$

$$\mathrm{Pr2to1Zp}_{i,n} = \int_{-1}^{1}\int_{-1}^{1}\int_{0}^{1} b_i(\xi,\eta,2\zeta-1)\, b_n(\xi,\eta,\zeta)\ \mathrm{d}\xi\mathrm{d}\eta\mathrm{d}\zeta \tag{A.45}$$

A.3 Normal Calculation

Fast Sweeping Method

For the fast sweeping method, the precomputed integrals are:

$$\mathrm{AveX}_i = \frac{1}{4}\int_{-1}^{1}\int_{-1}^{1}\frac{\partial b_i}{\partial \xi}\,\mathrm{d}\xi\mathrm{d}\eta \tag{A.46}$$

$$\mathrm{AveY}_i = \frac{1}{4}\int_{-1}^{1}\int_{-1}^{1}\frac{\partial b_i}{\partial \eta}\,\mathrm{d}\xi\mathrm{d}\eta \tag{A.47}$$

$$\mathrm{fsm\_Va}_{n,i,j} = \int_{-1}^{1}\int_{-1}^{1}\left(\frac{\partial b_i}{\partial \xi}\frac{\partial b_j}{\partial \xi}+\frac{\partial b_i}{\partial \eta}\frac{\partial b_j}{\partial \eta}\right) b_n\,\mathrm{d}\xi\mathrm{d}\eta \tag{A.48}$$

$$\mathrm{fsm\_SLa}_{n,i} = \int_{-1}^{1} b_i(-1,\eta)\, b_n(-1,\eta)\ \mathrm{d}\eta \tag{A.49}$$

$$\mathrm{fsm\_SLb}_{n,i} = \int_{-1}^{1} b_i(+1,\eta)\, b_n(-1,\eta)\ \mathrm{d}\eta \tag{A.50}$$

$$\mathrm{fsm\_SRa}_{n,i} = \int_{-1}^{1} b_i(+1,\eta)\, b_n(+1,\eta)\ \mathrm{d}\eta \tag{A.51}$$

$$\mathrm{fsm\_SRb}_{n,i} = \int_{-1}^{1} b_i(-1,\eta)\, b_n(+1,\eta)\ \mathrm{d}\eta \tag{A.52}$$

$$\mathrm{fsm\_SLa}_{n,i} = \int_{-1}^{1} b_i(\xi,-1)\, b_n(\xi,-1)\ \mathrm{d}\xi \tag{A.53}$$

$$\mathrm{fsm\_SLb}_{n,i} = \int_{-1}^{1} b_i(\xi,+1)\, b_n(\xi,-1)\ \mathrm{d}\xi \tag{A.54}$$

$$\mathrm{fsm\_SRa}_{n,i} = \int_{-1}^{1} b_i(\xi,+1)\, b_n(\xi,+1)\ \mathrm{d}\xi \tag{A.55}$$

$$\mathrm{fsm\_SRb}_{n,i} = \int_{-1}^{1} b_i(\xi,-1)\, b_n(\xi,+1)\ \mathrm{d}\xi \tag{A.56}$$

Projection

For 3-cell projection, the precomputed integrals are:

$$\text{fsm\_DXm}_{k,n} = \sum_{i=1}^{N_{gex}} \left(\int_{-1}^{1} \int_{-1}^{-1/3} b_k(3\xi + 2, \eta)\, b_i(\xi, \eta)\ \mathrm{d}\xi\mathrm{d}\eta \right) \left(\int_{-1}^{1} \int_{-1}^{1} \frac{\partial b_i(\xi/3, \eta)}{\partial \xi} b_n(\xi, \eta)\ \mathrm{d}\xi\mathrm{d}\eta \right) \tag{A.57}$$

$$\text{fsm\_DXc}_{k,n} = \sum_{i=1}^{N_{gex}} \left(\int_{-1}^{1} \int_{-1/3}^{1/3} b_k(3\xi, \eta)\, b_i(\xi, \eta)\ \mathrm{d}\xi\mathrm{d}\eta \right) \left(\int_{-1}^{1} \int_{-1}^{1} \frac{\partial b_i(\xi/3, \eta)}{\partial \xi} b_n(\xi, \eta)\ \mathrm{d}\xi\mathrm{d}\eta \right) \tag{A.58}$$

$$\text{fsm\_DXp}_{k,n} = \sum_{i=1}^{N_{gex}} \left(\int_{-1}^{1} \int_{1/3}^{1} b_k(3\xi - 2, \eta)\, b_i(\xi, \eta)\ \mathrm{d}\xi\mathrm{d}\eta \right) \left(\int_{-1}^{1} \int_{-1}^{1} \frac{\partial b_i(\xi/3, \eta)}{\partial \xi} b_n(\xi, \eta)\ \mathrm{d}\xi\mathrm{d}\eta \right) \tag{A.59}$$

$$\text{fsm\_DYm}_{k,n} = \sum_{i=1}^{N_{gex}} \left(\int_{-1}^{-1/3} \int_{-1}^{1} b_k(\xi, 3\eta + 2)\, b_i(\xi, \eta)\ \mathrm{d}\xi\mathrm{d}\eta \right) \left(\int_{-1}^{1} \int_{-1}^{1} \frac{\partial b_i(\xi, \eta/3)}{\partial \eta} b_n(\xi, \eta)\ \mathrm{d}\xi\mathrm{d}\eta \right) \tag{A.60}$$

$$\text{fsm\_DYc}_{k,n} = \sum_{i=1}^{N_{gex}} \left(\int_{-1/3}^{1/3} \int_{-1}^{1} b_k(\xi, 3\eta)\, b_i(\xi, \eta)\ \mathrm{d}\xi\mathrm{d}\eta \right) \left(\int_{-1}^{1} \int_{-1}^{1} \frac{\partial b_i(\xi, \eta/3)}{\partial \eta} b_n(\xi, \eta)\ \mathrm{d}\xi\mathrm{d}\eta \right) \tag{A.61}$$

$$\text{fsm\_DYp}_{k,n} = \sum_{i=1}^{N_{gex}} \left(\int_{1/3}^{1} \int_{-1}^{1} b_k(\xi, 3\eta - 2)\, b_i(\xi, \eta)\ \mathrm{d}\xi\mathrm{d}\eta \right) \left(\int_{-1}^{1} \int_{-1}^{1} \frac{\partial b_i(\xi, \eta/3)}{\partial \eta} b_n(\xi, \eta)\ \mathrm{d}\xi\mathrm{d}\eta \right) \tag{A.62}$$