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

by

Zechariah J. Jibben

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

Approved July 2015 by the
Graduate Supervisory Committee:

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

ARIZONA STATE UNIVERSITY

August 2015

Abstract

This dissertation describes a process for interface capturing via an arbitrary-order, nearly quadrature free, discontinuous Galerkin (DG) scheme for the conservative level set method (Olsson et al., 2005, 2008). The DG numerical method is utilized to solve both advection and reinitialization, and executed on a refined level set grid (Herrmann, 2008) for effective use of processing power. Computation is executed in parallel utilizing both 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.

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. Other standard test cases, such as Zalesak's disk and deformation of columns and spheres in periodic vortices are also performed, showing several orders of magnitude improvement over traditional WENO level set methods. These tests also show the impact of reinitialization, which often increases shape and volume errors as a result of level set scalar trapping by normal vectors calculated from the local level set field.

Accelerating advection via GPU hardware is found to provide a 30x speedup factor comparing a 2.0GHz Intel Xeon E5-2620 CPU in serial vs. a Nvidia Tesla K20 GPU, with speedup factors increasing with polynomial degree until shared memory is filled. A similar algorithm is implemented for reinitialization, which relies on heavier use of shared and global memory and as a result fills them more quickly and produces smaller speedups of 18x.

Acknowledgements

This work was supported by the National Science Foundation grant CBET-1054272 and the Los Alamos National Laboratory Computational Physics Summer Workshop 2013. Calculations were performed using the ASU Advanced Computing Center Saguaro and Ocotillo clusters and the Los Alamos National Laboratory Moonlight ASC and Darwin CCS-7 clusters. Calculations were also performed on a Tesla K20 awarded by the Nvidia Academic Hardware Donation program.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

A crucial 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 exploring the fundamental physics of multiphase flow and mixing 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 the interesting dynamics. Approaching the problem analytically yields a system of coupled nonlinear partial differential equations, for which there are no known exact solutions or solution methods for the vast majority of engineering applications. On the other hand,

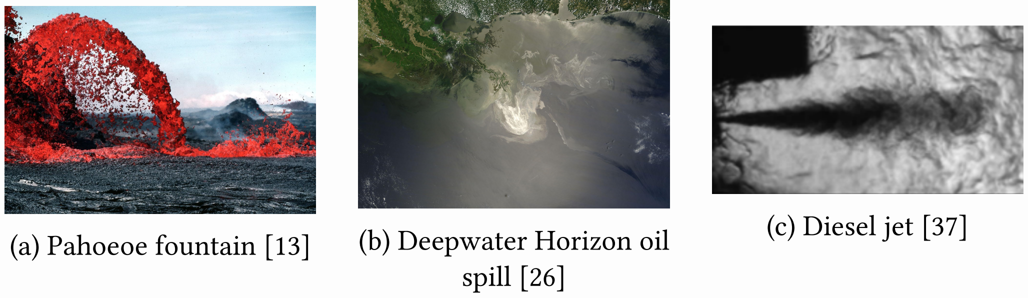

(a) Pahoeoe fountain [13] (b) Deepwater Horizon oil spill [26] (c) Diesel jet [37]

Figure 1.1: Multiphase Flow Examples

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 simulations that predict multiphase flows are vital to a multitude of engineering applications and our understanding of the underlying 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. Since experiments at operating conditions are typically unfeasible or impossible, the inference these simulations rely on leaves room for doubt. Therefore, developing a simulation which solves the governing nonlinear partial differential equations directly, from first principles, 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 with high order accuracy, as many methods suffer from either producing non-negligible mass errors or being limited to low order. The conservative level set (CLS) method [28, 29] is used with a discontinuous Galerkin discretization to achieve a high order representation of the phase interface while also offering a much improved mass conservation.

The governing equations have historically been solved by a variety of numerical approaches, with finite difference, finite volume, and spectral methods being popular choices. To avoid the need for extremely fine meshes to achieve a chosen accuracy, and therefore reduce the necessary computing power, high order solvers are essential. Although they are more computationally expensive on the same mesh size compared to low order methods, they converge on far coarser meshes, meaning accurate results can be achieved with less computing power. 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 runtime and programmer care be spent on communication. Often the overhead cost of inter-processor communication is significant 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 convergence rates while keeping a compact stencil, only dependent on immediate neighbors. DG methods can, however, be quite computationally expensive for each cell update. 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 well suited to 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.

GPUs have rose in popularity in high performance computing in recent years for their ability to perform more operations per second with the same or less power, effectively reducing the cost of a calculation. This is done through fine-grained parallelism, which invokes many low clock-rate cores as opposed to fewer high clock-rate cores, as is the case with CPUs. The fine-grained approach to parallelism requires a different programming methodology than is traditional for course-grained parallelism, in particular typically requiring greater care with memory access operations.

Since interface capturing indicates to the flowsolver regions subtended by each fluid and supplies surface tension forces, it is essential to solve accurately. In fact, small errors in interface position can result in large momentum errors, particularly when density ratios are large. This dissertation details the methods and algorithms developed for the interface capturing components of a predictive numerical laboratory for fluid flow fields involving immiscible interfaces. This includes descriptions of the CLS method, the accurate CLS

(ACLS) method [9], 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. The work follows a similar approach to Czajkowski and Desjardins [8] and Owkes and Desjardins [30], but extends their work in three ways. First, all spatially-dependent variables are projected into the DG basis in order to achieve the full predicted convergence rate of the DG method. Second, GPUs are leveraged for additional computing power. Lastly, the method is implemented on the refined level set grid (RLSG) [15] to mitigate computational effort.

1.2 Governing Equations

Herrmann [15] 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 $\boldsymbol{x}_f$.

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

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

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 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 discretized level set methods do not share this property and often exhibit strong violations of mass conservation. On the other hand, level sets are capable of being solved with high order numerical methods and have the benefit of straightforward calculation of 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 [9, 28, 31], 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 profile, originally suggested by Chopp [5], 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. Numerical representations of 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. The inevitable result is numerical errors, reducing the accuracy of curvature and surface tension, resulting in mass losses/-gains as well as momentum errors. For numerical accuracy, and much improved mass conservation, it is necessary to periodically reinitialize ϕ to restore the original signed distance profile. Several approaches to reinitialization exist, many of which are either computationally expensive or move the interface front. A common PDE-based approach developed by Sussman et al. [38] avoids explicitly locating the interface, and 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 $\Delta x \to 0$ mass is conserved. However, the discretized form of the equation does not conserve mass, and in many cases worsens the mass errors by transporting the 0-isosurface. As a result, the numerical result of the traditional level set method can diverge greatly from the exact solution.

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. [28, 29] 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). In this form, the level set scalar $G(\boldsymbol{x}, t)$ is a conserved quantity.

The interface is implicitly defined as a 0.5-isosurface of G, with $G > 0.5$ 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 in practice. The reason for this choice of profile, as opposed to piecewise function for example, is that a conservative reinitialization equation which restores the level set scalar to this form exists. As with traditional level sets, the advection equation distorts G, in this case dissipating the scalar. Over time, these errors distort the interface and reduce accuracy. Reinitialization sharpens the level set scalar, restoring the initial profile. Olsson et al. [29] suggests a conservative PDE which invokes a compression term along the direction normal to the interface, and a diffusive term to prevent the profile from becoming too thin. These terms are balanced such that the equation converges on a hyperbolic tangent profile with a thickness parameter of ε.

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

Olsson computes the normal vector,

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

from the local level set scalar. [9] suggests the accurate conservative level set method (ACLS) which instead requires the normal vectors be constructed from the interface geometry alone rather than the local G field. With this approach, oscillations or variations

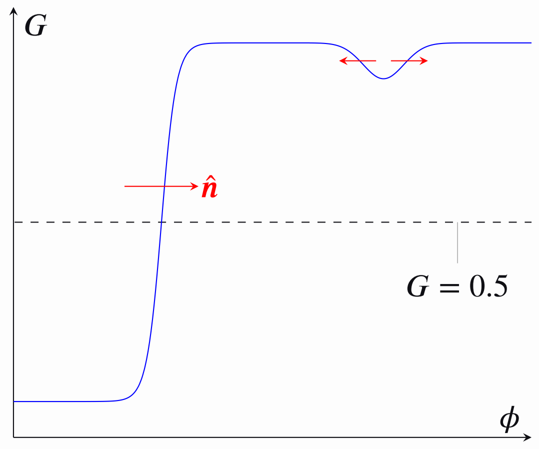


Figure 2.1: Hyperbolic Tangent Profile With Flaw Away From Interface

throughout the domain that do not cross the $G = 0.5$ threshold do not affect the normal vector field. With normal vectors unphysically varying throughout the domain (see Fig. 2.1), reinitialization gathers G away from droplets as it attempts to enforce a hyperbolic tangent profile on numerical noise. As a result, volume and shape errors increase, and new interfaces can be formed. The ACLS method computes normals via Eq. (2.6) from a signed distance function ϕ, which itself is calculated from the level set scalar field G in the vicinity of the 0.5-isosurface and marched or sweeped out. Several possible methods for calculating an arbitrary-order ϕ throughout the computational domain are discussed in Ch. 4.

The preceding reinitialization equation, then, treats G as a conserved variable and enforces the hyperbolic tangent profile with thickness ε. When needed, it is to be evaluated in pseudo-time, τ, either until convergence or for a user selected amount of pseudo-time.

Eqs. (2.3) and (2.5), together with interface normal calculation, 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

The simulation is performed on the refined level set grid (RLSG) proposed by Herrmann [15]. With the RLSG, the CLS 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 CLS equations are only solved in cells and blocks within a user defined distance from the interface. This is possible because advection and reinitialization only need to be solved inside a region surrounding the interface itself. That is, computational effort need not be wasted advecting contours that do not directly affect the 0.5 isosurface. Five important bands are generated: a) the A-band, consisting only of cells which contain the interface, b) the T-band, in which advection and reinitialization are solved, c) the W-band, which contains all ghost cells immediately neighboring the T-band, and d) the X-band, which extends beyond the T- and W- bands for volume integration, and e) the Z-band, which contains the outermost ghost cells. For details on band generation and parallelization of the refined level set grid, see [15].

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. A compact stencil is beneficial for a variety of reasons, primarily because it makes parallelization simpler (especially in the case of unstructured grids) and lowers inter-processor communication costs. DG accomplishes a high-order compact scheme by allowing sub-cell variation and storing information about derivatives locally in the form of basis function coefficients. As a result, fluxes and volume terms indicate not only changes in cell averages, as with finite volume methods, but changes for all modes calculated through coupled interactions between them. LeSaint and Raviart [17] proved DG can formally achieve a $k+1$ order convergence rate with k^{th} degree polynomials for linear problems, while Cockburn and Shu [6] found this to also be achievable for nonlinear problems in practice. This section describes the scheme construction.

3.1 Spatial Discretization

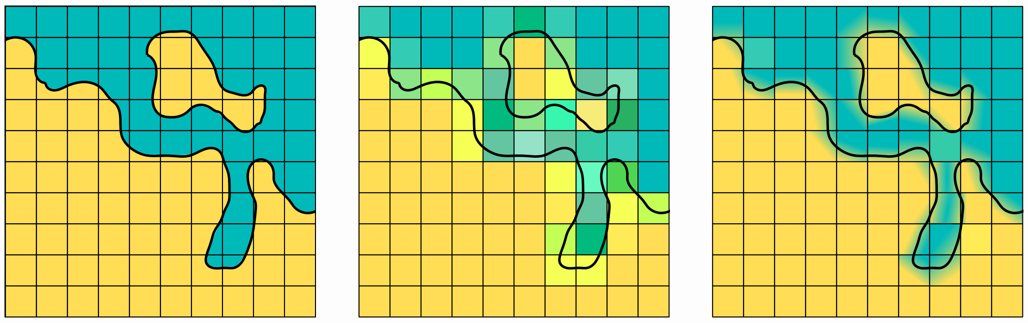

(a) Fluid Interface (b) Finite Volume (c) Discontinuous Galerkin

Figure 3.1: Interface Discretization

Originally developed by Reed and Hill [32], 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 $\mathscr{I}_\kappa$. For the purposes of this paper, the domain is mapped onto an equidistant Cartesian mesh ($\Delta x = \Delta y = \Delta z$).

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

The finite volume method then assigns an average of the solution variable to each cell. On the other hand, the discontinuous Galerkin method includes more information 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

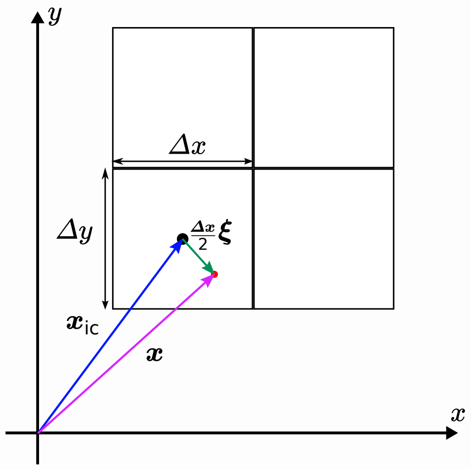
$$\boldsymbol{f}(\boldsymbol{x}, t) \; = \sum_{i=1}^{\infty} \boldsymbol{f}_i^{\kappa}(t)\, b_i(\boldsymbol{\xi}) \approx \sum_{i=1}^{N_f} \boldsymbol{f}_i^{\kappa}(t)\, b_i(\boldsymbol{\xi})\,, \tag{3.1}$$

where the series is truncated at N_f. 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 performing an inner product with the corresponding basis function (i.e., integrating the input function multiplied by the basis with respect to sub-cell coordinates over the space spanned by the cell).

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

The spatial 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.

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



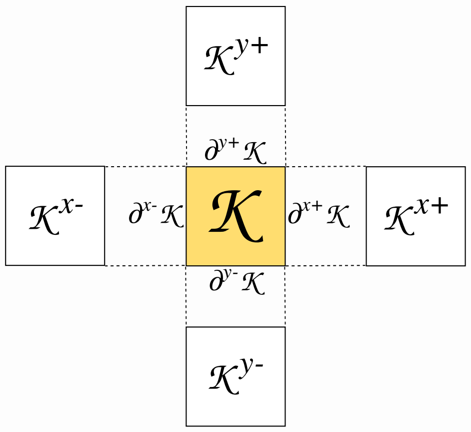
y
Δx
Δy
$\frac{\Delta x}{2}\boldsymbol{\xi}$
$\boldsymbol{x}_{\text{ic}}$
$\boldsymbol{x}$
x

Figure 3.2: Position Vectors

The normalized Legendre polynomial basis is selected for their orthonormality property, Eq. (3.4), removing the need to invert a mass matrix when solving equations.

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

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 $N_f = \left(k_f + 1\right)^3$.



$\mathcal{K}^{y+}$
$\partial^{y+}\mathcal{K}$
$\mathcal{K}^{x-}$
$\partial^{x-}\mathcal{K}$
$\mathcal{K}$
$\partial^{x+}\mathcal{K}$
$\mathcal{K}^{x+}$
$\partial^{y-}\mathcal{K}$
$\mathcal{K}^{y-}$

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). Writing derivatives 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 results in a systems 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 [8] and Owkes and Desjardins [30]. However, their method held the velocity and normal vectors constant within a cell ($N_u = N_n = 1$), expanding only the level set scalar to full order in the discontinuous basis. By doing so, they were able to easily calculate accurate normal vectors using a second order fast marching method, but the result was an overall second order method (which is shown in Sec. 6.1). Here, all variables are projected with full order into the DG basis, allowing for higher convergence rates.

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

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

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 analytically evaluated prior to running the simulation and tabulated. It is found that they form sparse arrays (discussed in detail in Sec. 3.5), and therefore present an opportunity for considerable speedup. To complete the spatial discretization, all that remains is to write the flux integrals in discrete form.

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 ought to be used in 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 $\boldsymbol{G}$, $\boldsymbol{u}$, 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 [6]. There are three criterion used here to determine which numerical fluxes to use: the approach should be (a) computationally inexpensive, (b) easily formulated in a quadrature-free sense, and (c) represent the mathematics accurately enough to achieve arbitrarily high convergence rates.

Upwinding

In the linear advection case, $\boldsymbol{f}(G) = \boldsymbol{G}\boldsymbol{u}$, the simple upwind flux is chosen–it meets all 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. [24] 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

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

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 $\hat{\boldsymbol{f}}^{\text{up}}$ becomes ambiguous, although the face center value is still used in practice. It is found through manufactured solutions tests that the error is largest in regions of converging velocity fields, and such problems reduce the convergence rate of the L_{∞} norm to k^{th} order, possibly as a result of upwinding. Quadrature-based methods can select the upwind direction independently for each quadrature point, possibly avoiding these problems.

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 no longer applicable. Instead, the local Lax-Friedrichs method is implemented, as described by Cockburn and Shu [6]. Other possible choices, such as the Godunov flux or Enquist-Osher flux are far more difficult to evaluate while avoiding quadrature. The flux is evaluated by

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

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

Inserting the nonlinear 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),

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

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 in which information is propogated, which does not hold true for diffusive flux. A quadrature-free reconstruction method was suggested by Luo et al. [22]. 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,

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

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

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

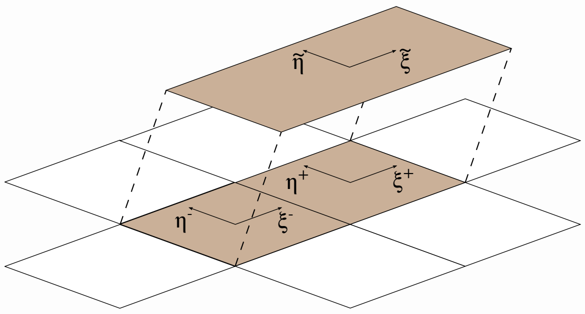


Figure 3.4: Two-Cell Projection in x-Direction

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

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

which is summed along the faces f. In the shared coordinate space, the cell face simply lies along the centerline which bisects the extended dimension. For example, a face whose normal is aligned with the x-axis is at $\tilde{\xi} = 0$ in the new shared coordinate space. 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,

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

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

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

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

One important aspect of the reconstruction flux that ought to be addressed is the number of degrees of freedom in the shared basis. In particular, is it necessary to preserve all degrees of freedom by enriching the shared basis in order to achieve predicted convergence rates? With each cell having a basis of size $(k+1)^3$, in 3D, this would imply that in order to capture all variations (plus a degree to model the jump across the cell interface) the shared basis would need to be of size $(2*k+2)^3$, 8 times larger than the original. Results shown in Sec. 6.1 indicate enrichment does reduce the error, but the original basis size is sufficient to achieve $k+1$ convergence rates. This is an important finding, since the number of arithmetic operations and amount of data storage required for integral arrays over such a large basis can become prohibitively expensive.

3.3 Temporal Discretization

The DG spatial discretization and flux equations result in a system of $2N_g\, N_{\text{cells}}$ ordinary differential equations (ODEs) describing the time evolution of the DG coefficients. Despite being a larger system of equations, ODEs are far more easily solved. At this point, a choice of temporal discretization will complete the scheme.

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 described by Cockburn and Shu [6] and Gottlieb [12].

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

2. solve for intermediate stages

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

3. set $g^{n+1}_{m,\kappa} = g^{(N_{\text{RK}})}_{m,\kappa}$

where [6, 12] provide stable values for $\alpha_{i,k}, \beta_{i,k}$, which are reprinted in Table 3.1. Higher order RK methods are described by Gottlieb [12], however the work here is limited to a maximum of 3rd order RK methods, which are found to be sufficient for $k+1$ convergence rates in all tests shown in Ch. 6.

Table 3.1: Stable Values for $\alpha_{i,k}$, $\beta_{i,k}$ Presented by Cockburn and Shu [6] and Gottlieb [12]

order	$\alpha_{i,k}$	$\beta_{i,k}$
2	1	1
	$\frac{1}{2},\frac{1}{2}$	$0,\frac{1}{2}$
3	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 [6]:

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

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 [21]:

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

where $\beta(k)$ is a function of polynomial order (several values are given in Table 3.2). Note that, in practice, $\varepsilon = \Delta x/2$ so that Δt scales with Δx rather than its square, making the method feasible. Still, reinitialization is considerably more expensive than advection, and is therefore executed as seldomly as possible. The reinitialization factor F introduced by

Owkes and Desjardins [30] is used, which relates the amount of reinitialization performed to the advection time step size:

$$\tau_f - \tau_i = F\Delta t \max\left(|\boldsymbol{u} \cdot \hat{\boldsymbol{n}}|\right) \tag{3.25}$$

We also introduce a reinitialization occurrence factor T_r, such that reinitialization is performed only between real time intervals of length T_r.

Table 3.2: Stable Values for β Presented by Lörcher et al. [21]

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 Slope Limiting

When employing polynomials of degree $k \geq 4$, reinitialization becomes unstable as a result of the nonlinear compressive term, particularly near regions with intersecting characteristics. That is, near discontinuous sign changes in the normal vector, high order representations of the normal vector exhibit oscillations as a form of Gibbs phenomenon. The oscillations in the normal vector combined with the compressive reinitialization term introduces oscillations and spikes in the G field. To mitigate this effect, a slope limiter is used. As an alternative, it may be possible to locally truncate the polynomial order of the normal vector.

We use the slope limiter introduced by van Leer [41, 42] and modified by Cockburn and Shu [6], where high order ($k > 1$) contributions in a cell are discarded if the average slope in that cell exceeds a multiple ν of the slope found through differences between the neighbor averages. In that case, the linear component is then corrected to produce the minimum of the given slopes. That is, in 1D Eq. (3.26) describes the limiter, with the minmod function defined by Eq. (3.27).

$$G_h\Big|_{I_j} = g_0 b_0 + \left(\frac{\Delta x}{2} m\left(\overline{G_{h,x}}\Big|_{I_j}, \nu\frac{\overline{G}_{j+1} - \overline{G}_j}{\Delta x}, \nu\frac{\overline{G}_j - \overline{G}_{j-1}}{\Delta x}\right)/c_1\right) b_1 + H.O.T. \tag{3.26}$$

$$m\left(a_1, a_2, a_3\right) = \begin{cases} s\min_{1\leq n\leq 3} \left|a_n\right| & \text{if } s = \operatorname{sign}\left(a_1\right) = \operatorname{sign}\left(a_2\right) = \operatorname{sign}\left(a_3\right) \\ 0 & \text{otherwise} \end{cases} \tag{3.27}$$

The average derivative, $\overline{G_{h,x}}\Big|_{I_j}$ is evaluated before limiting from Eq. (3.28). The quantity c_1 is the coefficient in the normalized Legendre polynomial $b_1 = c_1\xi$. In 2D, $c_1 = \sqrt{3}/2$ while in 3D $c_1 = \sqrt{6}/4$. If the return value of the minmod function is not equal to its first argument, the higher order coefficients are set to zero. Otherwise, they are not changed. This algorithm is repeated in each direction for a 2D or 3D simulation.

$$\overline{G_{h,x}}\Big|_{I_j} = \frac{\Delta x}{2} g_i \int_{\mathcal{K}} \frac{\partial b_i}{\partial \xi}\, \mathrm{d}V \tag{3.28}$$

The variable ν is user-set, and represents the ratio between the internally-calculated derivative (Eq. (3.28)) and derivatives calculated from neighbors allowed before limiting is enforced. van Leer originally set the quantity to $\nu = 1$, while Cockburn & Shu use a less restrictive $\nu = 2$. It is found in practice that the high curvature of the hyperbolic tangent profile near the $G = 0$ isosurface results in over-limiting in those regions unless ν is set higher. That is, $\nu = 2$ causes high curvature regions of the domain to be needlessly limited. In fact, it can be shown that for a hyperbolic tangent profile, there will always be regions where the slope limiter is active. For smooth functions, Taylor series and the cell average definition, Eq. (3.29), can be used to approximate differences as in Eq. (3.30).

$$\overline{f}_i = \int_{\mathcal{K}} f\left(\boldsymbol{x}_i + \xi\frac{\Delta x}{2}\right)\, \mathrm{d}V \tag{3.29}$$

$$\frac{\overline{G}_{j+1} - \overline{G}_j}{\Delta x} \approx \overline{G'_i} + \overline{G''_i}\frac{\Delta x}{2}, \qquad \frac{\overline{G}_j - \overline{G}_{j-1}}{\Delta x} \approx \overline{G'_i} - \overline{G''_i}\frac{\Delta x}{2} \tag{3.30}$$

This slope limiter will limit whenever $\overline{G'_j} > \nu \min\left(\left|\frac{\overline{G}_{i+1}-\overline{G}_i}{\Delta x}\right|, \left|\frac{\overline{G}_i-\overline{G}_{i-1}}{\Delta x}\right|\right)$. For regions where G is strictly increasing, this is combined with Eq. (3.30) and simplified to

$$1 > \nu\left(1 \pm \frac{\Delta x}{2}\frac{\overline{G''_i}}{\overline{G'_i}}\right) \tag{3.31}$$

Inserting the hyperbolic tangent profile Eq. (2.4) and $\varepsilon = \Delta x/2$, results in a limiter will be active whenever Eq. (3.32) is true. As a result, the limiter will apply throughout the region described by Eq. (3.33).

$$\left|\tanh\left(\frac{x}{2\varepsilon}\right)\right| > \frac{\nu - 1}{\nu} \tag{3.32}$$

$$x > \Delta x \operatorname{atanh}\left(\frac{\nu - 1}{\nu}\right) \tag{3.33}$$

The special case where $\nu = 1$, Eq. (3.33) implies that the limiter will be active everywhere. When $\nu = 2$, the limiter will be active everywhere outside a band of width $1.1\Delta x$ surrounding the $G = 0$ isosurface. In fact, Eq. (3.33) implies that for a hyperbolic tangent profile, any choice of ν will force the solution at some distance from the interface to always be limited.

One common way to prevent the slope limiter from limiting in regions it ought not is to use the corrected minmod function, suggested by Shu [36] to prevent limiting near critical points.

$$\overline{m}\left(a_1, a_2, a_3\right) = \begin{cases} a_1 & \text{if } |a_1| \le M\Delta x^2 \\ m\left(a_1, a_2, a_3\right) & \text{otherwise} \end{cases} \tag{3.34}$$

However, since reinitialization is particularly unstable anywhere there is a discontinuity in the normal vector, critical points are precisely where the limiter is needed.

A second way to prevent the solution from being always limited near the interface is to choose a larger value for ν. In practice, it is found that $\nu = 2.5$ is large enough to retain high order accuracy near the interface while not so large as to miss the issues that ought to be corrected. It should be noted that while the limiter will not always activate near the isosurface, it will do so when there are local maxima and minima. This is particularly noticeable where the $G = 0$ isosurface has high curvature.

3.5 Integral Array Calculation and Storage

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 and stored in an array for reference later (see Ap. A). This avoids the use of quadrature, saving computation time.

A variety of software exists to analytically calculate integrals, such as Mathematica, Maple, or SymPy. However, these tools are very general and designed to handle a wide variety of integrand forms. For this task, many millions of integrals of polynomials must be performed, so speed is essential while generality is not. In fact, there are so many high-k, 3D reinitialization integrals that the above tools are simply not feasible. To address this, a specialized tool named PolyCracker was developed in C to quickly calculate only integrals of polynomials. Briefly, PolyCracker tabulates relevant integrals of polynomials using Eq. (3.35). Integrals of general polynomials, such as those in Ap. A, are calculated from products and sums of these simple integrals along with associated coefficients, as in Eq. (3.36). This approach, compared to the approach used by more general analytical tools, performs on the order of 10^4 times faster, even in serial. PolyCracker has been parallelized with MPI to further reduce calculation time.

$$\texttt{Int[k]} = \int_{-1}^{1} x^k \, \mathrm{d}x = \frac{1}{k+1}\left(1 - (-1)^{k+1}\right) \tag{3.35}$$

$$\int_{-1}^{1} P(x, y, z) \; \mathrm{d}x = \sum_{i,j,k} a_{i,j,k} \texttt{Int[i]} * \texttt{Int[j]} * \texttt{Int[k]} \tag{3.36}$$

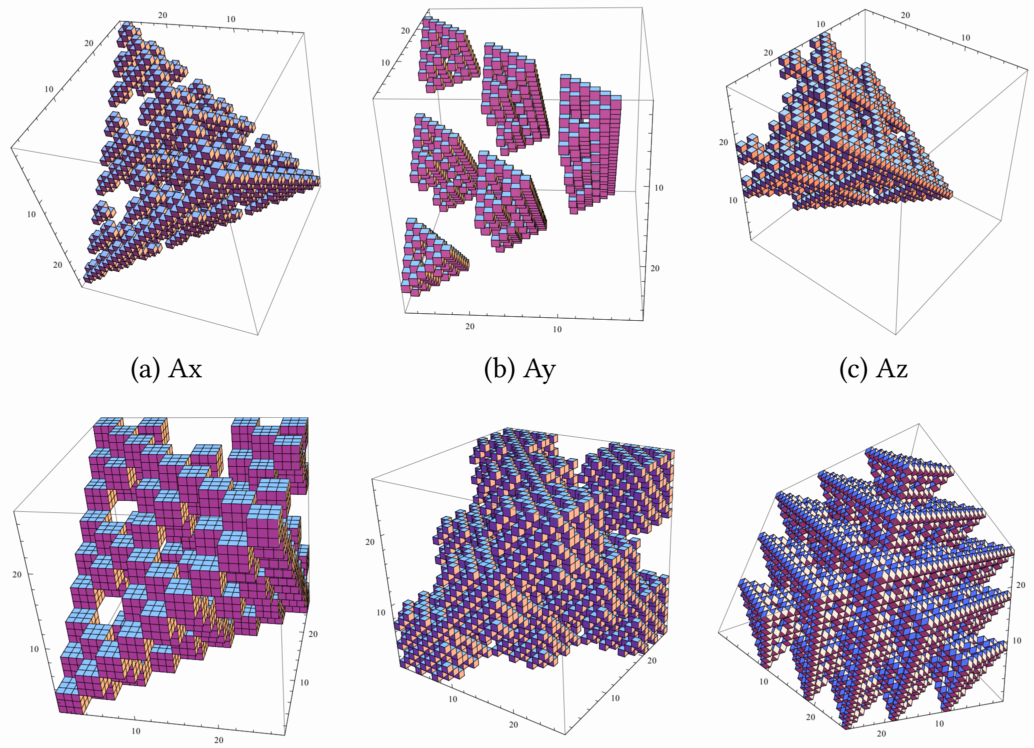
Importantly, the resulting 3D arrays are sparse, resulting from the orthogonality of the Legendre polynomial basis (see Table 3.3 and Fig. 3.5). For reference, the advection volume integrals are named Ax, Ay, and Az, and "-" face surface integrals are SAxm, SAym, and SAzm, shown explicitly along with the analogous definitions for reinitialization shown in Ap. A. 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%

Similarly, reinitialization integral arrays are quite sparse, depending on the level of enrichment. Table 3.4 and Table 3.5 show examples of matrix fill fractions for reinitialization integral arrays. Since the nonlinear and diffusive integral arrays involve 4 indicies rather than 3, they are much larger and take much more time to compute. Still, they are very sparse, especially in 3D simulations.

For computation speed and effective use of memory, it is beneficial to store these arrays in a manner that takes advantage of the fact many elements are equal to zero. For a CPU-based simulation, efficient 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 encourages a quadrature-free formulation. A quadrature-based scheme, relying on numerical integration, evaluates the


(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.

integrals without separating coefficients and basis functions. As a result, such an approach does not know a priori that much of the work can be bypassed.

The sparsity found in the quadrature-free scheme automatically takes care of this if an appropriate storage format is used. A compressed row storage (CRS) format, also called compressed sparse row, based on [10] was chosen, such that 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.

Table 3.4: Matrix Fill Fraction for 2D Reinitialization Integral Arrays

Polynomial	Integral Fill Fractions				
Degree	# of elements	B	SB	Cxx/yy	Cxy
1	256	12.5%	50.0%	6.25%	10.9%
2	6561	13.5%	45.7%	9.02%	13.7%
3	65536	14.4%	43.8%	10.9%	15.7%
4	390625	14.1%	41.3%	11.5	15.8%

Table 3.5: Matrix Fill Fraction for 3D Reinitialization Integral Arrays

Polynomial	Integral Fill Fractions				
Degree	# of elements	B	SB	Cxx/yy/zz	Cxy/yz/xz
1	4096	6.25%	25.0%	3.1%	5.5%
2	531441	5.42%	18.3%	3.3%	5.3%
3	16777216	5.66%	17.9%	4.3%	6.2%
4	244140625	6.05%	17.7%	5.0%	6.8%

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 [9]. In contrast to the CLS method proposed in [29], ACLS requires local variations in the G field that do not cross the 0.5-isosurface to not impact the normal vectors. To accomplish this, the normals are calculated from a signed distance function, which is reconstructed from the geometry implicitly described by the $G = 0.5$ isosurface. To date, no method for constructing a signed distance function at arbitrarily high order has been developed. This chapter will describe various attempts to do so, the current CLS approach for calculating normals from the local G field, and possible avenues for future research.

4.1 The Fast Sweeping Method

This approach involves calculating the signed distance function directly from the Eikonal equation, in order to calculate normal vectors by the following process:

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).

$$\phi_n = 2\varepsilon \int_{\mathscr{K}} \operatorname{atanh}\big(2g_i b_i - 1\big)\, b_n \,\mathrm{d}V \tag{4.1}$$

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).

$$n_n^k = \int_{\mathcal{K}} \frac{\mathrm{d}\phi_i^k b_i}{\sqrt{\sum_{j=1}^3 \left(\mathrm{d}\phi_k^j b_k\right)^2}} b_n \,\mathrm{d}V \tag{4.2}$$

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 2nd order DG Eikonal equation solvers [23, 45]. 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. $i = 1 \ldots N_{\text{imax}},\ j = 1 \ldots N_{\text{jmax}}$
2. $i = N_{\text{imax}} \ldots 1,\ j = 1 \ldots N_{\text{jmax}}$
3. $i = N_{\text{imax}} \ldots 1,\ j = N_{\text{jmax}} \ldots 1$
4. $i = 1 \ldots N_{\text{imax}},\ j = N_{\text{jmax}} \ldots 1$

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 [19, 40]. 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. [19] and Zhang et al. [45] using a quadratic solver.

Unfortunately, the DG form of the fast sweeping solver presented by [45] is unstable unless provided with two pieces of information: 1) a good initial guess from a finite-difference solver and 2) causality flags indicating characteristic directions. Luo [23] suggests an alternative 2nd order DG fast sweeping solver that does not require an initial guess and is far simpler to implement. However, it is not clear how to generalize his method to arbitrarily high orders. The approach described by [45] 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

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

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.

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

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

where α^f are called local causality constants by Zhang et al. [45]. His definition is modified to the following for the square of the Eikonal equation:

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

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 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 $\alpha^{x+} \ll 1$, 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.

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

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

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

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. It is possible this can be avoided through a causality flag correction routine.

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 $i = 1 \ldots N_{\text{imax}}$, the loops are executed within the local block from $i = N_{\text{block imin}} \ldots N_{\text{block imax}}$. 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 Time-Dependent Hamilton-Jacobi Solver

A classic approach to reconstructing the signed distance function for the traditional level set method is solving a time-dependent Hamilton-Jacobi equation that converges when the Eikonal equation is satisfied [38]. In this case, the PDE used for traditional reinitialization (Eq. (2.2)) could be used before every CLS reinitialization call to calculate the signed distance function, and in turn the normal vectors. Several arbitrary-order DG Hamilton-Jacobi solvers exist [3, 4, 16, 18, 20, 43] and have proven successful on several nonlinear PDEs. However, applying these methods to Sussman et al. 's PDE for reinitialization has not been successful since numerical instabilities destroy the solution starting in regions with intersecting characteristics, with similar concerns shared by Grooss and Hesthaven [14]. However, newer methods such as those by Cheng and Wang [4] and Liu and Pollack [20] have yet to be explored.

4.3 Brute-Force Method

This approach is more straightforward than the previous methods for calculating the signed distance function, but is far more expensive. Here, the signed distance function is solved on the DG quadrature points or a much finer grid via a more traditional method (e.g. finite-difference fast sweeping, brute-force search, etc.). Since it is performed on a many times finer mesh, accuracy isn't necessarily lost. However, since these methods are generally lower-order, it is likely this algorithm will not scale along with the rest of the method. As a result, this is not an attractive approach.

4.4 Gradient Calculation

Regardless of the previous algorithms used, the normal vectors are dependent on a normalized gradient operation over either the local G field or on the reconstructed ϕ field. To differentiate a field (here denoted ϕ) across the computational domain, it is possible to do so by simply evaluating Eq. (4.16) locally, with the gradient represented by the coefficients via Eq. (4.17). 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).

$$\mathrm{d}\phi_n^\alpha = \phi_i \frac{2}{\Delta x_\alpha} \int_{\mathscr{K}} \frac{\mathrm{d}b_i}{\mathrm{d}\xi_\alpha} b_n \,\mathrm{d}V \tag{4.16}$$

$$\frac{\partial \phi}{\partial x_\alpha} \approx \sum_{i=1}^{N_g} \mathrm{d}\phi_i^{\kappa,\alpha}(t)\, b_i(\boldsymbol{\xi}) \tag{4.17}$$

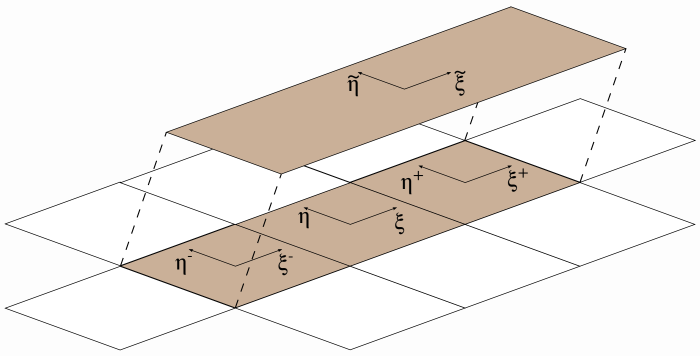


Figure 4.1: Three-Cell Projection in x-Direction

$$\begin{aligned}
\tilde{\xi} &= \begin{cases} \frac{1}{3}\left(\xi^+ + 2\right) & \text{where } -1 < \xi^+ \le 1 \\ \frac{1}{3}\xi & \text{where } -1 \le \xi \le 1 \\ \frac{1}{3}\left(\xi^- - 2\right) & \text{where } -1 \le \xi^- < 1 \end{cases} \\
\tilde{\eta} &= \begin{cases} \eta^+ & \text{where } -1 < \xi^+ \le 1 \\ \eta & \text{where } -1 \le \xi \le 1 \\ \eta^- & \text{where } -1 \le \xi^- < 1 \end{cases}
\end{aligned} \tag{4.18}$$

In the x-direction, it can easily be seen that the coordinates are related by Eq. (4.18). 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.

$$\widetilde{\phi}_n^{\alpha} = \phi_i^{\alpha-} \int_{\widetilde{\mathscr{K}}^{\alpha-}} b_i^{\alpha-} \tilde{b}_n \, \mathrm{d}\widetilde{V} + \phi_i \int_{\widetilde{\mathscr{K}}^{\alpha}} b_i \tilde{b}_n \, \mathrm{d}\widetilde{V} + \phi_i^{j+} \int_{\widetilde{\mathscr{K}}^{\alpha+}} b_i^{\alpha+} \tilde{b}_n \, \mathrm{d}\widetilde{V} \tag{4.19}$$

The resulting coefficients are then differentiated,

$$\mathrm{d}\phi_n^{\alpha} = \widetilde{\phi}_i^{\alpha} \frac{2}{\Delta x_{\alpha}} \int_{\mathscr{K}} \frac{\partial \tilde{b}_i}{\partial \xi_{\alpha}} b_n \, \mathrm{d}V \tag{4.20}$$

Eqs. (4.19) and (4.20) are combined, allowing the entire projection and differentiation procedure to be executed with a single equation:

$$\begin{aligned} \mathrm{d}\phi_n^{\alpha} \; = \; & \phi_k^{\alpha-} \frac{2}{\Delta x_{\alpha}} \int_{\widetilde{\mathscr{K}}^{\alpha-}} b_k^{\alpha-} \tilde{b}_i \, \mathrm{d}\widetilde{V} \int_{\mathscr{K}} \frac{\partial \tilde{b}_i}{\partial \xi_{\alpha}} b_n \, \mathrm{d}V \\ & + \phi_k \frac{2}{\Delta x_{\alpha}} \int_{\widetilde{\mathscr{K}}^{\alpha}} b_k \tilde{b}_i \, \mathrm{d}\widetilde{V} \int_{\mathscr{K}} \frac{\partial \tilde{b}_i}{\partial \xi_{\alpha}} b_n \, \mathrm{d}V \\ & + \phi_k^{\alpha+} \frac{2}{\Delta x_{\alpha}} \int_{\widetilde{\mathscr{K}}^{\alpha+}} b_k^{\alpha+} \tilde{b}_i \, \mathrm{d}\widetilde{V} \int_{\mathscr{K}} \frac{\partial \tilde{b}_i}{\partial \xi_{\alpha}} b_n \, \mathrm{d}V \end{aligned} \tag{4.21}$$

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

5.1 Programming Libraries

For accelerating computation with GPUs, there are a variety of libraries that follow different ideologies and practices. Two low level examples are OpenCL and CUDA, the former being open source and supported by a wide range of hardware, while the latter is developed by Nvidia for Nvidia GPUs. For large codes, this makes OpenCL the more attractive option since it is capable of being very portable to new and different architectures. However, Nvidia has excellent debugging and optimization tools for CUDA, which makes the development process quicker and easier. Since the kernel syntax is very similar between the two, the software here is developed in CUDA, which can then be easily ported to OpenCL.

It is worth noting that OpenCL and CUDA currently do not support Fortran [39]. 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 sending pointers to that data as arguments to a C function.

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 within a derived data type. 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.

5.2 CUDA and OpenCL algorithm

Using CUDA terminology, a GPU operates by executing a function called a *kernel* in parallel on a cluster of *threads*, which are organized into *blocks* with resources allocated to groups of 32 threads called *warps*. Threads and blocks then have associated integers for identification. Eq. (3.9) is solved by assigning a single cell to each block, updating all level set scalar coefficients for that cell. Then, threads within a block share the workload of tensor-vector multiplication.

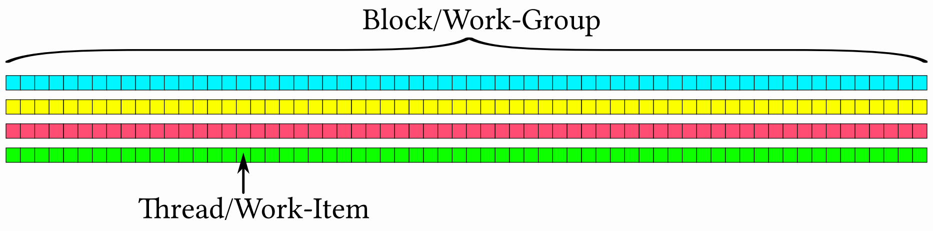


Figure 5.1: CUDA/OpenCL Execution Model

Algorithm 1 GPU 3D Array Multiplication

```
tiX ⟵ local ID of thread
ntX ⟵ block size
term ⟵ 0.0
for l ⟵ start[n]+tiX, end[n] with step size ntX do
    term += u[i2[l]] * g[i3[l]] * Z[l]          ▷ multiply u and g coeff associated with l
end for
declare shared array partialsum of length ntX
partialsum[tiX] ⟵ term                         ▷ save private result to shared array
return reduction_sum_within_tile(partialsum)
```

A second aspect of GPU programming is the use of different memory spaces for array storage: i) *global* memory, which is available to all threads but invokes an additional 400-600 clock cycles of latency when accessed [27] (for comparison, memory read/write time itself is 8 operations per clock cycle), ii) *shared* memory, which is accessible to members within the same block and may be accessed ∼100x faster than global memory [35] since the latency is significantly reduced, and iii) small *private* registers, which is not shared between threads but is slightly faster than shared memory. An excellent description of this is given by Scarpino [35]. Currently, integral arrays are stored in global memory. Since all g and u coefficients associated with a cell are accessed frequently by a block, storing solution variable coefficients in shared memory before calculating a given tensor multiplication term provides a 10-15% speedup. Unfortunately, shared memory cannot be dynamically allocated. Rather, the CPU must send a request to reserve variable sized blocks of memory before queuing kernel execution.

In Alg. 1, Eq. (3.9) is considered a series of equations of the form $\Delta g_n^\kappa \mathrel{+}= \sum_{k=1}^{N_u} u_k^\kappa \sum_{i=1}^{N_g} g_i^\kappa Z_{n,k,i}$ with coefficients for velocity `u`, level set scalar `g`, and an integral array `Z`. Each thread has its own instance of the variable `my_dg`, in which it sums together a subset of the above equation. Following the CRS format, a single integer l which corresponds to nonzero elements of the compressed array `Z[l]` is looped over. Each call of the multiplication routine evaluates a term for one row n, looping through a subset of `Z` bounded by two integers `start[n]` and `end[n]`.

In order to take advantage of memory coalescence and evenly distribute the workload, and hence reduce runtime, the local group of threads must align their access to the global array `Z` by their local id number [27]. For example, thread 7 will access the array element located immediately after the memory accessed by thread 6 and immediately before the memory accessed thread 8. To accomplish this, threads begin the loop offset by their local id and step through the loop by the local block size. Furthermore, the `0`th thread should access array elements that are multiples of 32, the warp size [7]. This optimization alone provides an additional 10-15% speedup.

Finally, the CPU is easily able to reduce the number of repeated flux calculations by, away from edges, computing only left-side fluxes. The result is then used to find the right-side flux of the left neighbor. This same optimization is not so simple on the GPU when cells are assigned single blocks, since there is no global syncronization between blocks or any guarantee of the order in which blocks are executed. As a result, multiple blocks writing to an address of global memory often causes collisions, destroying the data. To avoid repeated calculations on the GPU, left side fluxes are stored in separate global arrays, which are kept on the GPU for a second kernel which only calculates right-side flux from that information. Again, each block in this kernel is assigned a cell, and threads update each coefficient according to the provided left-side flux of the corresponding right-side neighbor.

The same process is repeated for reinitialization, with similar algorithms for the 2- and 4-index array multiplications. However, since more variables are necessary in a given calculation (three different arrays rather than two) it is far easier to run out of shared memory, which can reduce performance.

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. 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 reinitialization directly.

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 [34] at Sandia National Laboratory. An excellent overview of the method is given by Roache [33]. The motivation for MMS lies in the difficulty of obtaining analytical solutions 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 to the problem, you modify the problem to match a solution of your choosing. This means that the solution variables $\boldsymbol{G}$, $\boldsymbol{u}$, and $\boldsymbol{\hat{n}}$ become arbitrary and may be prescribed. Then, the governing PDE is modified with a source term $\boldsymbol{Q}\left(\boldsymbol{x}, t\right)$, which is computed exactly from the chosen solution variable fields. The source function is then projected into the discontinuous basis via Eq. (3.2).

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

It is important to remember that MMS tests do not evaluate the physics of the solver, only the mathematics, since the governing equation is modified. So, in our case the CLS method is not being evaluated (since the hyperbolic tangent profile plays no role), but the RKDG discretization is.

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.

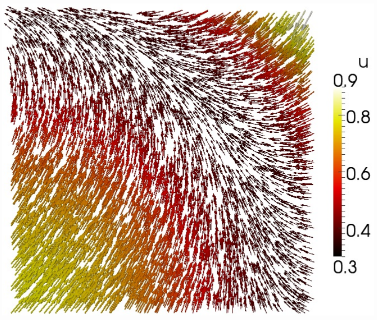
$$\frac{\partial G}{\partial t} + \nabla \cdot (G\boldsymbol{u}) = Q(\boldsymbol{x}, t)\,. \tag{6.2}$$

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

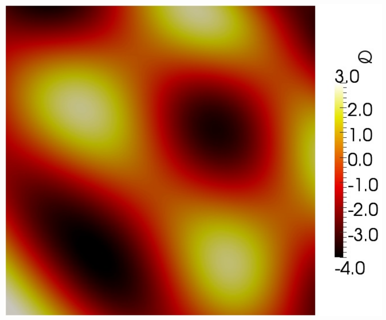
$$Q(\boldsymbol{x}, t) = \frac{\partial G_{\text{ex}}(\boldsymbol{x}, t)}{\partial t} + \nabla \cdot \left(G_{\text{ex}}(\boldsymbol{x}, t)\, \boldsymbol{u}_{\text{ex}}(\boldsymbol{x}, t)\right). \tag{6.3}$$

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:

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



(a) MMS velocity field $\boldsymbol{u}$



(b) MMS source term $\boldsymbol{Q}$

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.

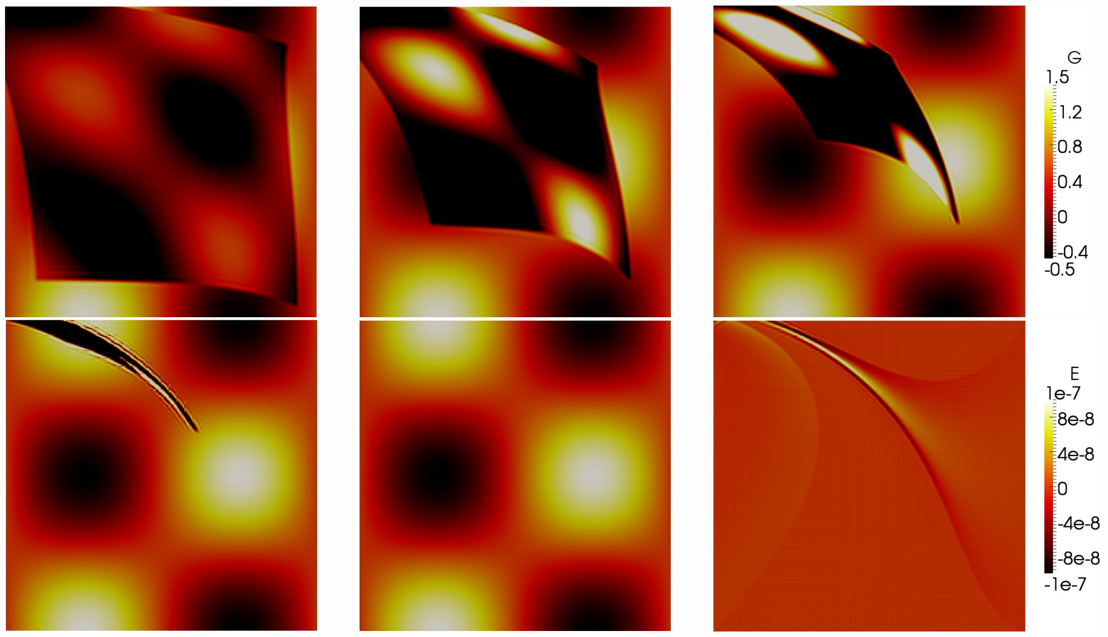


Figure 6.2: Solution G of MMS test case for $\Delta x = 1/40$, RKDG-4 for $t = 0.2, 0.5, 1.0, 2.0, 4.3$ time units, and error E at steady state (from top left to bottom right).

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.

Table 6.2: Error Norms of Advection MMS Test Case and Their Order of Convergence Under Grid Refinement for Various Polynomial Orders

k_g	k_u	Δx	L_∞	order	L_1	order
1	0	1/10	7.96e-1	-	6.71e-2	-
1	0	1/20	3.00e-1	1.4	3.00e-2	1.1
1	0	1/40	4.03e-1	-0.4	1.78e-2	0.4
1	0	1/80	2.58e-1	0.6	9.26e-3	0.7
1	1	1/10	1.08e-1	-	2.38e-2	-
1	1	1/20	5.72e-2	0.9	7.32e-3	1.7
1	1	1/40	2.18e-2	1.4	2.09e-3	1.8
1	1	1/80	9.96e-3	1.1	5.70e-4	1.9
1	3	1/10	1.38e-1	-	2.51e-2	-
1	3	1/20	6.83e-2	1.0	7.44e-3	1.8
1	3	1/40	2.89e-2	1.2	2.06e-3	1.9
1	3	1/80	1.10e-2	1.4	5.45e-4	1.9
3	0	1/10	9.28e-1	-	8.01e-2	-
3	0	1/20	1.20	-0.4	4.30e-2	0.9
3	0	1/40	6.21e-1	1.0	2.29e-2	0.9
3	0	1/80	4.59e-1	0.4	1.13e-2	1.0
3	1	1/10	5.04e-2	-	5.34e-3	-
3	1	1/20	2.40e-2	1.1	1.31e-3	2.0
3	1	1/40	1.05e-2	1.2	3.28e-4	2.0
3	1	1/80	4.39e-3	1.3	8.31e-5	2.0
3	3	1/10	4.49e-4	-	7.76e-5	-
3	3	1/20	4.88e-5	3.2	5.69e-6	3.8
3	3	1/40	6.35e-6	2.9	3.79e-7	3.9
3	3	1/80	7.87e-7	3.0	2.47e-8	3.9

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

Table 6.2 shows the resulting errors and convergence rates when $\boldsymbol{G}$ and $\boldsymbol{u}$ are given different polynomial degrees. It is found that the convergence rate of the solution is limited by the smallest number of degrees of freedom given to a particular variable. That is, the RKDG method only converges at a rate of $\min(k_g, k_u) + 1$. This particular finding motivates the need for treating both velocity and normal vectors as DG variables.

The reinitialization equation is modified with a source term:

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

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

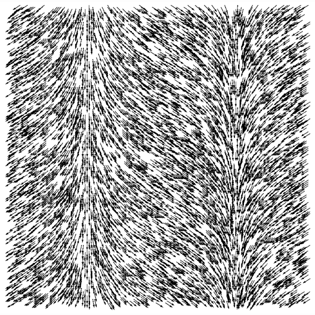
It should be noted that MMS does not require the normal vector to conform in any sense to the normal of any surface in this system, since it does not verify CLS. As such, there is also no need for it to be normalized. However, for evaluating the source term, the normal is calculated from the exact solution chosen for G, using direct differentiation as described in Ch. 4. This is done to include error from that algorithm in our testing. In this case, the exact solution to G is chosen to be

$$G_{\mathrm{ex}}(x, y) = \frac{\exp\left(\frac{1}{4}\sin(2\pi x) - y\right) - e^{-3/4}}{e^{3/4} - e^{-3/4}}. \tag{6.7}$$

This function is chosen for its nonlinear behavior, as well as a nicely behaving source term (not too sharp anywhere) and normal that is defined everywhere in the domain $[-0.5, 0.5]^2$. The resulting source term is too long to write here, but a color plot is shown in Fig. 6.3.

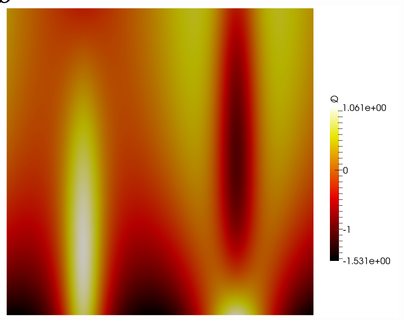
One important detail is that the scalar ε must be held constant rather than modified with the grid spacing (as it would in flow simulations), since it appears in the reinitialization equation itself. In practice, it represents the thickness of the hyperbolic tangent profile and its value is changed along with the grid. However, changing it also changes the governing equation, thereby invalidating convergence rates. In these tests, a value of $\varepsilon = 0.2$ is used since it approximately balances the magnitudes of the convective and diffusive terms. Unfortunately, a constant ε also results in timestep sizes proportional to Δx^2, as shown in Eq. (3.24). As a result, these tests can become expensive quickly. Therefore, only $k = 1$ results are presented here (circle tests go higher, however).

The results for this test are shown in Table 6.3. Convergence rates start low, but approach $k+1$ in both norms as the grid is refined. The results for unenriched diffusive flux are shown in Table 6.4. The errors are much higher than those for the enriched case, but the convergence rates are very similar.



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

b



(b) MMS source term $\boldsymbol{Q}$

Figure 6.3: MMS Reinitialization Test

Table 6.3: 2D RKDG-1 Reinitialization MMS Error Norms and Their Order of Convergence Under Grid Refinement

Δx	L_∞	order	L_1	order
1/10	2.54e-2	-	4.73e-3	-
1/20	1.30e-2	1.0	2.24e-3	1.1
1/40	4.88e-3	1.4	6.87e-4	1.7
1/80	1.55e-3	1.7	1.94e-4	1.8

Table 6.4: Unenriched 2D RKDG-1 Reinitialization MMS Error Norms and Their Order of Convergence Under Grid Refinement

Δx	L_∞	order	L_1	order
1/10	1.27e-1	-	2.42e-2	-
1/20	4.17e-2	1.6	8.80e-3	1.5
1/40	1.71e-2	1.3	3.03e-3	1.5
1/80	5.82e-3	1.6	9.50e-4	1.7

Reinitialization's 3D capabilities are evaluated with Eq. (6.8). The results, shown in Table 6.5, show similar results approaching $k + 1$ order convergence rates.

$$G_{\text{ex}}(x, y, z) = \frac{\exp\left(\frac{1}{4}\sin\left(\sqrt{2}\pi\,(x+z)\right) - y\right) - e^{-3/4}}{e^{3/4} - e^{-3/4}}. \tag{6.8}$$

Table 6.5: 3D RKDG-1 Reinitialization MMS Error Norms and Their Order of Convergence Under Grid Refinement

Δx	L_∞	order	L_1	order
1/10	2.24e-2	-	4.52e-3	-
1/20	1.39e-2	0.7	1.51e-3	1.6
1/40	5.02e-3	1.5	4.81e-4	1.7

6.2 Circle Test

To assess reinitialization and the normal calculation algorithm, 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

$$G(\boldsymbol{x}) = \frac{1}{2}\left(\tanh\left(\frac{R_0 - \sqrt{x^2 + y^2}}{2\varepsilon_0}\right) + 1\right). \tag{6.9}$$

Here, ε_0 refers to the initial thickness of the level set profile, here set larger than the thickness ε after reinitialization to simulate correcting dissipative errors normally generated by numerically solving the advection equation. Reinitialization then sharpens the interface from thickness $\varepsilon_0 \to \varepsilon$, the latter used in Eq. (2.5). The conservative reinitialization equation is designed to conserve G, which comes at the cost of not conserving the volume enclosed by the $G = 0.5$ isosurface. In the context of our circle test, this means that the profile thickness is sharpened, the 0.5-isosurface is transported outward in order to conserve G. The new radius R can be calculated 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. Fig. 6.4 shows the resulting relationship between change in radius and change in profile thickness. This advection of the interface performed by reinitialization is also demonstrated by McCaslin and Desjardins [25].

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

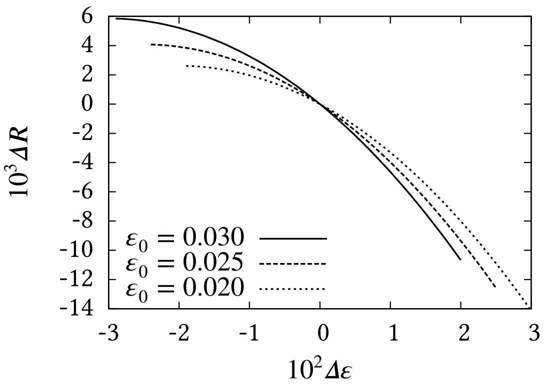


Figure 6.4: Change in Radius Against Change in Profile Thickness

$$G_{\mathrm{ex}}(\boldsymbol{x}) = \frac{1}{2}\left(\tanh\left(\frac{R - \sqrt{x^2 + y^2}}{2\varepsilon}\right) + 1\right) \tag{6.10}$$

$$E = \left|G_{\mathrm{ex}}(\boldsymbol{x}) - G(\boldsymbol{x})\right| \tag{6.11}$$

The results of a refinement study for $k = 3$ polynomials are shown in Table 6.6, which shows the L_∞ and L_1 norms of the level set scalar error at steady state together with the associated convergence rates. The error is defined as the absolute value of the difference between the exact solution and final state of the level set scalar (ref. Eqs. (6.10) and (6.11)). 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 0.2530653 from Eq. (6.2). The L_1 and L_∞ norms of the error appear to converge with $k + 1$ order, with the coarsest mesh exhibiting high error from under-resolution.

Table 6.6: 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	2.16e-2	-	2.01e-3	-
1/40	1.04e-3	4.4	2.84e-5	6.1
1/80	6.19e-5	4.1	1.65e-6	4.1
1/160	3.77e-6	4.0	9.84e-8	4.1

6.3 Solid Body Rotations

Zalesak's disk [44], 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

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

The T-band is set to 8 cells, the W-band to 1 cell, and the X-band is set large enough to fill the entirety of the disk (for straightforward volume and shape error calculation). These parameters are shared by deforming column and sphere test cases. When the mesh is updated after the interface is advected, new X-band cells are given the value of $G = 0$. If any new X-band cells were generated inside the disk, they would be given a value of $G = 1$. In general, it is found that varying the X-band size has very little affect on the simulation, while changing the T-band size (especially if it is too small) can have a moderate impact.

The impact of banding on conservation of G is examined by calculating the fraction of G lost during the simulation via Eq. (6.12). This is done for the usual banding settings, just described, and for the case where the T-band fills the entire domain. For the case $k = 2, \Delta x = 1/100, T_r = 0.0, F = 0.0$, the default band settings result in a G loss fraction of

8.87e-7. With the T-band filling the entire domain, the conservation of G is demonstrated by the total G loss fraction of 9.98e-14.

$$\text{Fraction of } G \text{ lost} = \left(\int_{\Omega} G_f \, \mathrm{d}V - \int_{\Omega} G_0 \, \mathrm{d}V\right)\left(\int_{\Omega} G_0 \, \mathrm{d}V\right)^{-1} \tag{6.12}$$

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.

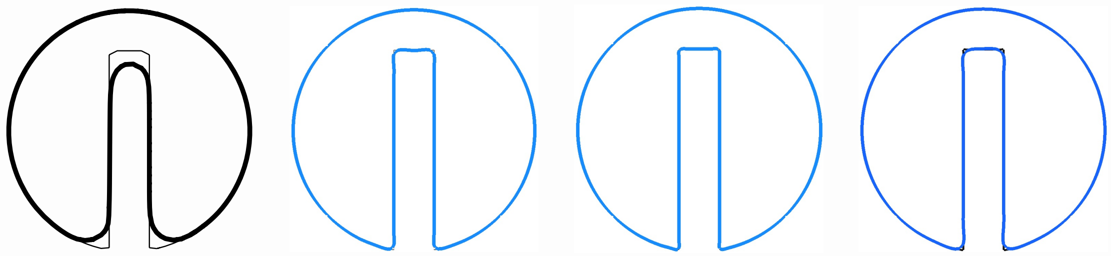


Figure 6.5: Zalesak's disk. From left to right: LS-WENO-5 ($\varDelta x = 1/100$ & 1 rotation) [15]; RKDG-CLS-4 ($\varDelta x = 1/50$ & 1 rotation); RKDG-CLS-4 ($\varDelta x = 1/100$ & 1 rotation); RKDG-CLS-4 ($\varDelta x = 1/50$ & 45 rotations). Exact solution shown as a thin line.

Table 6.7 summarizes the shape error, defined as

$$E = \frac{\int_A \left| H(G) - H\left(G_{\text{ex}}\right) \right| \, \mathrm{d}A}{\int_A H\left(G_{\text{ex}}\right) \, \mathrm{d}A} \tag{6.13}$$

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

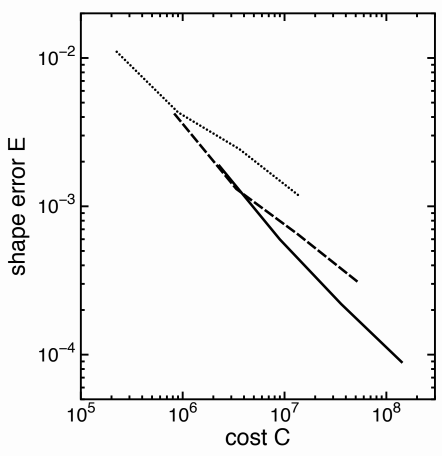


Figure 6.6: Shape error E as a function of scheme cost C; RKDG-CLS-2 (dotted line), RKDG-CLS-3 (dashed line), RKDG-CLS-4 (solid line).

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:

$$C \sim N^2\,(k+1)^2\,(2k+1)\,k\,. \tag{6.14}$$

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.

Reinitialization's affect on Zalesak's disk is analyzed with three amounts of reinitialization shown in Fig. 6.7, Table 6.7, and Table 6.8. In general, it is found that reinitialization increases shape errors. This occurs because reinitialization smooths out the corners of the notched disk, as Fig. 6.7 shows.

It is also found that some reinitialization improves volume conservation, but increasing the amount of reinitialization further will increase the volume error. Reinitialization

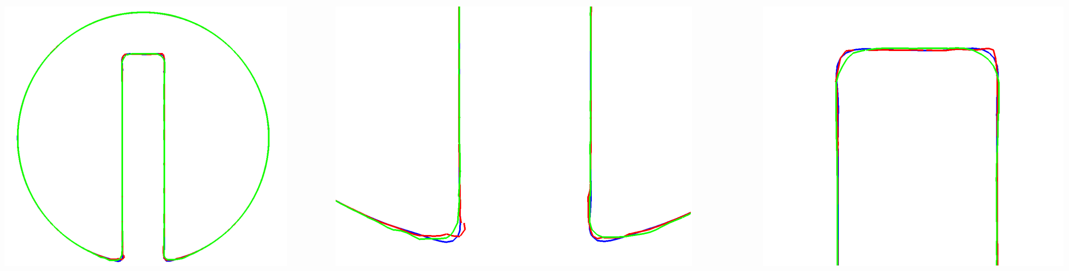


Figure 6.7: Interface shape of Zalesak's disk after 1 rotation, $k = 3$, $\Delta x = 1/100$: no reinitialization (blue); $T_r = 1.57$, $F = 2.0$ (red); $T_r = 0.79$, $F = 4.0$ (green)

is expected to improve volume conservation, but unphysically varying normal vectors away from the interface cause some G to be gathered outside the disk, slightly reducing the overall mass. Normal vectors vary because they are dependent on local changes in the G field, rather than on the interface geometry alone. As a result, more frequent reinitialization will increase volume errors. This trend isn't shared by the $k = 4$ case, since it employs limiters during reinitialization that degrade the accuracy in the sharp corners.

To address the issue of G-capturing by reinitialization, an arbitrarily high-order implementation of the accurate conservative level set method [9] would depend on a method for calculating high order DG normal vectors from only the interface geometry.

Table 6.7: Zalesak's Disk Shape Errors and Convergence Rates for RKDG-CLS-k After 1 Rotation

T_r, F	Δx	$k = 2$		$k = 3$		$k = 4$	
		E	order	E	order	E	order
0.0, 0.0	1/50	1.08e-2	-	3.09e-3	-	1.39e-3	-
	1/100	4.07e-3	1.4	9.95e-4	1.6	4.79e-4	1.5
	1/200	2.38e-3	0.8	4.25e-4	1.2	1.58e-4	1.6
	1/400	1.13e-3	1.1	1.76e-4	1.3	5.14e-5	1.6
1.57, 2.0	1/50	7.16e-3	-	2.97e-2	-	2.25e-2	-
	1/100	2.67e-3	1.4	2.43e-3	3.6	3.91e-3	2.5
	1/200	1.69e-3	0.7	6.86e-4	1.8	7.30e-4	2.4
	1/400	8.78e-4	0.9	2.11e-4	1.7	1.70e-4	2.1
0.79, 4.0	1/50	1.29e-2	-	3.70e-2	-	3.73e-2	-
	1/100	2.77e-3	2.2	4.07e-3	3.2	4.72e-3	3.0
	1/200	1.46e-3	0.9	1.31e-3	1.6	9.17e-4	2.4
	1/400	5.88e-4	1.3	7.42e-4	0.8	6.82e-4	0.4

Table 6.8: Zalesak's Disk Volume Errors and Convergence Rates for RKDG-CLS-k After 1 Rotation.

T_r, F	Δx	$k = 2$		$k = 3$		$k = 4$	
		E	order	E	order	E	order
0.0, 0.0	1/50	3.08e-3	-	4.43e-4	-	1.06e-5	-
	1/100	4.87e-4	2.7	8.29e-5	2.4	9.23e-5	-3.1
	1/200	1.43e-4	1.8	2.42e-5	1.8	4.95e-6	4.2
	1/400	6.65e-5	1.1	1.30e-5	0.9	8.97e-7	2.5
1.57, 2.0	1/50	6.00e-4	-	1.87e-2	-	6.46e-3	-
	1/100	6.79e-5	3.1	8.16e-4	4.5	2.35e-4	4.8
	1/200	3.49e-5	1.0	2.13e-5	5.3	4.67e-5	2.3
	1/400	2.62e-5	0.4	1.19e-6	4.2	5.74e-6	3.0
0.79, 4.0	1/50	4.22e-4	-	3.13e-3	-	1.16e-2	-
	1/100	4.24e-4	0.9	3.10e-4	3.3	1.72e-4	6.1
	1/200	1.52e-4	0.6	4.54e-5	2.8	1.52e-5	3.5
	1/400	9.12e-5	0.7	2.71e-5	0.8	2.32e-5	-0.6

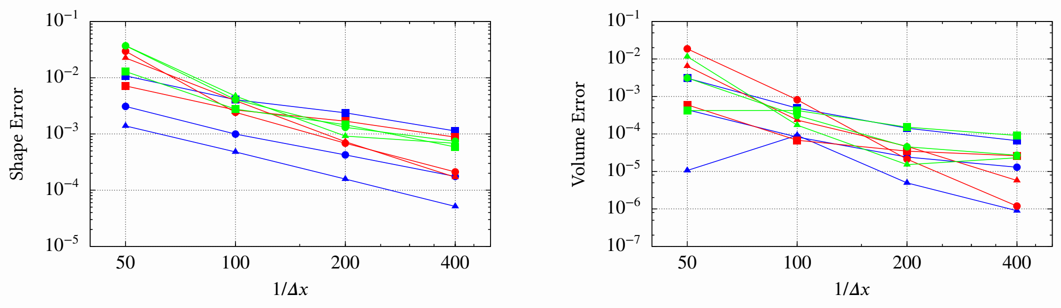


Figure 6.8: Shape and volume errors for Zalesak's disk. No reinitialization (blue), $T_r = 1.57, F = 2.0$ (red), $T_r = 0.79, F = 4.0$ (green). $k = 2$ (squares), $k = 3$ (circles), $k = 4$ (triangles).

For comparison, Herrmann [15] reports volume errors for a banded fifth order WENO method as 4.6e-3, 1.0e-3, 1.3e-4, and 7e-5 for 100^2, 200^2, 400^2, and 800^2 grids, respectively. This is most closely comparable to the RKDG $k = 2$ case with no reinitialization on meshes with 4x fewer cells.

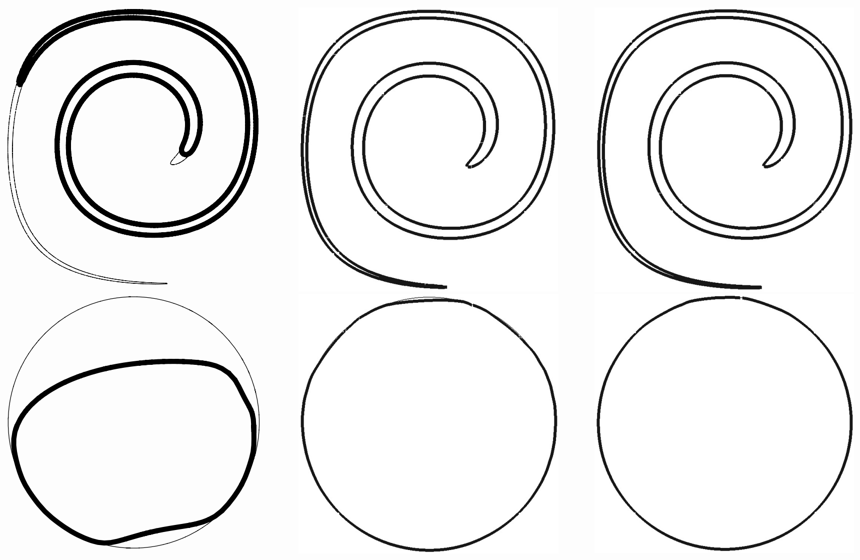


Figure 6.9: Interface shape of column in a deformation field at $t = T/2$ (top row) and $t = T$ (bottom row); from left to right: LS-WENO-5 with $\Delta x = 1/128$ [15], RKDG-CLS-4 method with $\Delta x = 1/64$ and $\Delta x = 1/128$. Thin line marks reference solution.

6.4 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. [11] 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

$$\Psi(\boldsymbol{x}, t) = \frac{1}{\pi} \sin^2(\pi x) \sin^2(\pi y) \cos(\pi t/T) \tag{6.15}$$

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. 6.9 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 $\Delta x = 1/128$ [15], 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 6.9 shows the shape error at $t = T$ as a function of grid spacing Δx for RKDG-CLS-2, RKDG-CLS-3, and RKDG-CLS-4 methods.

The impact of reinitialization is examined by changing the frequency with which reinitialization is performed, as well as the number of reinitialization iterations performed with each call. A comparison of several schemes is shown in Fig. 6.10, Table 6.9, and Table 6.10. In this case, reinitialization increases both volume and shape errors. In contrast to Zalesak's disk, the column is deformed, which causes more G to be left behind in a trail behind the droplet. As with before, local variations in the level set scalar cause reinitialization to gather G away from the interface. This often results in what is here called streaking (see Fig. 6.12), and with the added potential for lost G and streaks that deform the interface, both volume errors and shape errors are increased. This result further demonstrates the need for accurately calculated normals.

For comparison, Owkes and Desjardins [30] perform this test, showing $k = 2, T_r = 0.0, F = 0.5$ volume errors of $E = 8.9e-3, 4.1e-3, 4.6e-3$ for $\Delta x = 1/64, 1/128, 1/256$, respectively. Our results, in Table 6.10, show similar volume errors even with no reinitialization. Similarly, they show volume errors for $T_r = 0.0, F = 0.5, \Delta x = 1/128$ against polynomial orders $k = 1, 2, 3$ as $E = 9.4e-3, 4.1e-3, 5.4e-3$, respectively. Again, very similar results are found in the tests lacking reinitialization shown here. They show tremendous improvement comparing cases with reinitialization against without, showing for $k = 2, \Delta x = 1/64, T_r = 0.0$ volume errors of $E = 2.12e-2, 8.9e-3, 6.2e-3$ for $F = 0.0, 0.5, 1.0$, respectively. This further motivates the projection of all variables to the

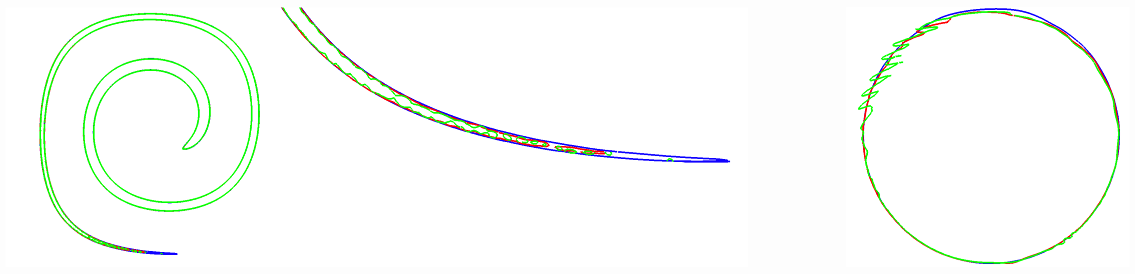


Figure 6.10: Interface shape of column in a deformation field for $k = 3$, $\Delta x = 1/128$ at $t = T/2$ (left, middle zoomed) and $t = T$ (right); no reinitialization (blue); $T_r = 0.5$, $F = 2.0$ (red); $T_r = 0.25$, $F = 4.0$ (green)

same polynomial order, since it drastically improves volume conservation even before reinitialization is performed. However, our results also show volume errors increasing with reinitialization rather than decreasing, as a result of the normal vector dependency on the local level set scalar.

Herrmann [15] reports volume errors for a banded fifth order WENO method as 3.1e-1, 4.6e-2, 1.0e-2, and 2.8e-3 for $128^2, 256^2, 512^2$, and 1024^2 grids, respectively. The performance of the RKDG method is far superior, as even the $k = 2$ case with no reinitialization on a 256^2 mesh produces a smaller error than any of the WENO tests performed.

This test also shows both shape and volume errors increasing with reinitialization, with a jump in error for $k = 4$ since the slope limiter is active for these tests. When reinitialization is active, the error diminishes with grid refinement, but increasing the polynomial degree has little affect in most cases. On the other hand, when reinitialization is not active, increasing the polynomial degree does reduce the error. This is likely because small discrepancies in the high order polynomials introduce greater opportunity for error in the normal vectors, and in turn the level set field. The $k = 4$ tests do not necessarily follow this trend since the active slope limiter often truncates cells to a piecewise linear solution.

Streaking is also affected by the band sizes chosen for the RLSG. Presently, new X-band cells are given values of either 0 or 1, depending on which side of the interface

Table 6.9: Deforming Column Shape Errors and Convergence Rates for RKDG-CLS-k After Full Flow Reversal at $t = T$

T_r, F	Δx	$k = 2$		$k = 3$		$k = 4$	
		E	order	E	order	E	order
0.0,0.0	1/64	5.12e-2	-	1.47e-2	-	6.46e-3	-
	1/128	1.05e-2	2.3	2.76e-3	2.4	1.33e-3	2.3
	1/256	1.88e-3	2.5	6.21e-4	2.2	2.54e-4	2.4
0.5,2.0	1/64	9.07e-2	-	9.76e-2	-	2.37e-1	-
	1/128	2.11e-2	2.1	1.91e-2	2.4	5.66e-2	2.1
	1/256	4.62e-3	2.2	4.59e-3	2.1	1.00e-2	2.5
0.25,4.0	1/64	8.36e-2	-	1.14e-1	-	4.53e-1	-
	1/128	2.61e-2	1.7	2.83e-2	2.0	1.32e-1	1.8
	1/256	4.41e-3	2.6	5.51e-3	2.4	2.29e-2	2.5

Table 6.10: Deforming Column Volume Errors and Convergence Rates for RKDG-CLS-k After Full Flow Reversal at $t = T$

T_r, F	Δx	$k = 2$		$k = 3$		$k = 4$	
		E	order	E	order	E	order
0.0,0.0	1/64	6.00e-3	-	8.25e-3	-	5.36e-3	-
	1/128	4.81e-3	0.3	2.58e-3	1.7	1.65e-3	1.7
	1/256	1.43e-3	1.8	7.91e-4	1.7	4.71e-4	1.8
0.5,2.0	1/64	8.60e-2	-	9.23e-2	-	1.06e-1	-
	1/128	1.82e-2	2.2	1.51e-2	2.6	2.75e-3	5.3
	1/256	3.46e-3	2.4	3.29e-3	2.2	2.08e-3	0.4
0.25,4.0	1/64	6.40e-2	-	8.57e-2	-	2.24e-1	-
	1/128	1.54e-2	2.1	1.67e-2	2.4	1.84e-2	3.6
	1/256	2.51e-3	2.6	3.47e-3	2.3	1.48e-3	3.6

Table 6.11: Deforming Column Volume and Shape Errors for $T_r = 0.5, F = 2.0, k = 2, \Delta x = 1/128$ at Flow Reversal at $t = T$ Against T-band Size

T-band size	volume error	shape error
8	1.74E-02	2.11E-02
15	1.59E-02	1.94E-02
128	1.55E-02	1.91E-02

they lie. Ideally, the transport band (T-band) is made large enough for this assumption to take place sufficiently far away along the hyperbolic tangent profile that the jump has little effect. However, since locally-dependent normals gather G away from the interface, often destroying the hyperbolic tangent profile, this becomes much more difficult. Fig. 6.12

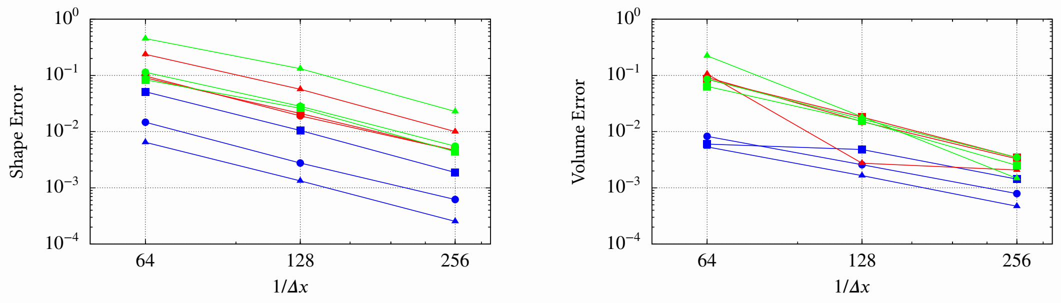


Figure 6.11: Shape and volume errors for deforming column. No reinitialization (blue), $T_r = 0.5$, $F = 2.0$ (red), $T_r = 0.25$, $F = 4.0$ (green). $k = 2$ (squares), $k = 3$ (circles), $k = 4$ (triangles).

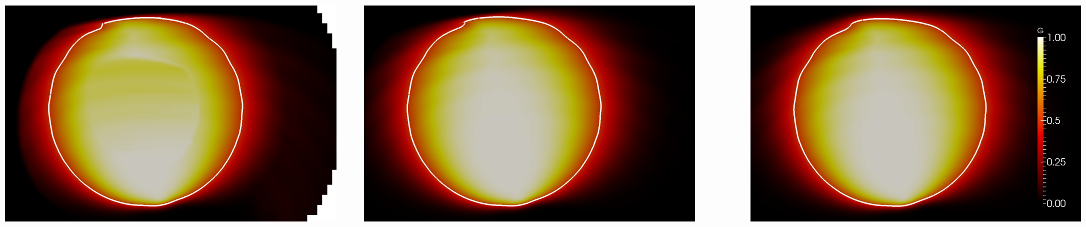


Figure 6.12: Streaking exhibited after full flow reversal for $T_r = 0.5$, $F = 2.0$, $k = 2$, $\Delta x = 1/128$. $G = 0.5$ isosurface indicated with a white line. T-band sizes of 8, 15, and 128 (full domain), respectively.

shows a comparison between two different band sizes and the impact on streaking and the resulting interface. Both volume and shape errors decrease as the T-band size is increased, shown in Table 6.11. This is the result of G being smeared out by advection, making it necessary for advection and reinitialization to handle G fluxes farther and farther from the interface. This can be remedied by more reinitialization, which would pull G closer to the vicinity of the interface by restoring the hyperbolic tangent profile. However, as Table 6.9 and Table 6.10 indicate, more reinitialization often increases errors if the normal vectors are calculated from the local level set scalar. This is because small variations in G away from the interface result in large variations in the normals, which in turn causes reinitialization to gather G away from the interface. With an ACLS approach, it is expected that reinitialization will properly restore the hyperbolic tangent profile and in turn allow for smaller T-band sizes without sacrificing accuracy.

Sphere in a Deformation Field

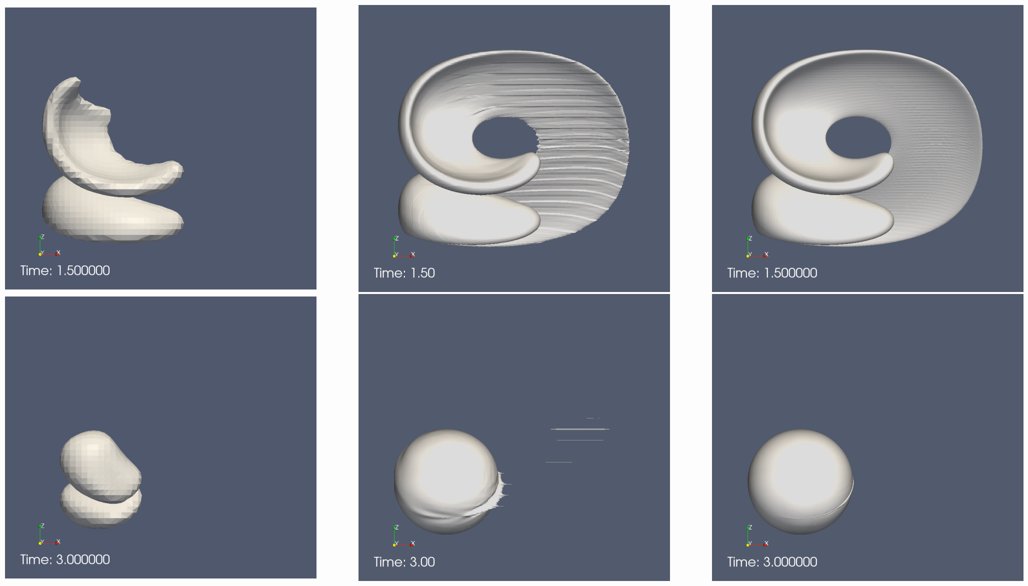


Figure 6.13: Sphere in a deformation field interface shape at $t = T/2$ (top row) and $t = T$ (bottom row); from left to right: LS-WENO-5 with $\Delta x = 1/128$, RKDG-CLS-4 with $\Delta x = 1/32$ and $\Delta x = 1/128$.

To demonstrate the performance of the RKDG-CLS method in three dimensions, the sphere in a deformation field case proposed by [11] 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

$$\begin{aligned} u &= 2\sin^2(\pi x)\sin(2\pi y)\sin(2\pi z)\cos(\pi t/T) \\ v &= -\sin(2\pi x)\sin^2(\pi y)\sin(2\pi z)\cos(\pi t/T) \\ w &= -\sin(2\pi x)\sin(2\pi y)\sin^2(\pi z)\cos(\pi t/T)\ , \end{aligned} \tag{6.16}$$

with $T = 3$. Fig. 6.13 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 superior results, even on a four time coarser mesh compared to the LS-WENO-5 method.

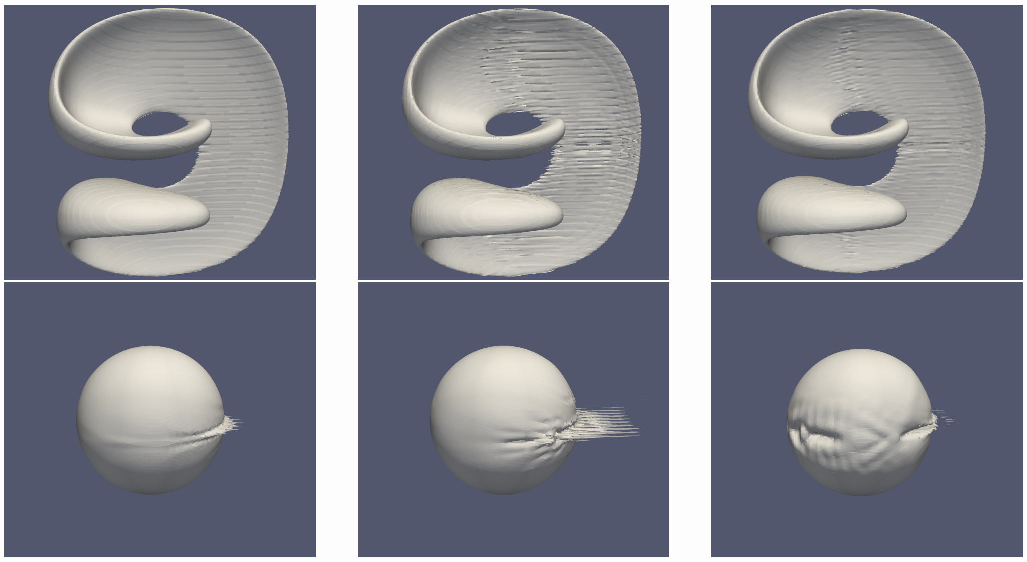


Figure 6.14: Sphere in a deformation field interface shape at $t = T/2$ (top row) and $t = T$ (bottom row), all RKDG-CLS-3 with $\Delta x = 1/64$; from left to right: No reinitialization; $T_r = 1.0, F = 1.0$; $T_r = 0.5, F = 1.0$.

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 $\Delta x = 1/32$. Results are shown in Table 6.12, Table 6.13, Table 6.14, Table 6.15.

Table 6.12: Deforming Sphere Shape Errors and Convergence Rates for RKDG-CLS-k After Full Flow Reversal at $t = T$

(a) Varying reinit factor 64^3 mesh, $k = 2$		(b) Varying mesh $T_r = 1.0, F = 1.0, k = 2$			(c) Varying DG order $T_r = 1.0, F = 1.0, 64^3$	
T_r, F	E	Δx	E	order	k	E
0.0,0.0	4.17e-2	1/32	6.76e-2	-	1	1.06e-1
1.0,1.0	7.14e-2	1/64	7.14e-2	-0.1	2	7.14e-2
0.5,1.0	4.86e-2	1/128	5.72e-2	0.3	3	3.19e-2

Similar to previous tests, it is found that without reinitialization, error is reduced both by grid refinement and increasing polynomial degree. Adding reinitialization accentuates variations in G, resulting in deformations and streaking in the interface, as shown in Fig. 6.14. This causes both shape errors and volume errors to increase with reinitialization.

Table 6.13: Deforming Sphere Volume Errors and Convergence Rates for RKDG-CLS-k After Full Flow Reversal at $t = T$

(a) Varying reinit factor 64^3 mesh, $k = 2$		(b) Varying mesh $T_r = 1.0, F = 1.0, k = 2$			(c) Varying DG order $T_r = 1.0, F = 1.0, 64^3$	
T_r, F	E	Δx	E	order	k	E
0.0,0.0	3.12e-3	1/32	1.35e-1	-	1	1.23e-1
1.0,1.0	3.17e-2	1/64	3.17e-2	2.1	2	3.17e-2
0.5,1.0	3.63e-2	1/128	2.50e-3	3.7	3	1.40e-2

Table 6.14: Deforming Sphere Shape Errors and Convergence Rates for RKDG-CLS-k with no Reinit. After Full Flow Reversal at $t = T$

Δx	$k = 2$		$k = 3$	
	E	order	E	order
1/32	6.70e-2	-	3.41e-2	-
1/64	4.17e-2	0.7	1.78e-2	0.9
1/128	3.45e-2	0.3	1.14e-2	0.6

Table 6.15: Deforming Sphere Volume Errors and Convergence Rates for RKDG-CLS-k with no Reinit. After Full Flow Reversal at $t = T$

Δx	$k = 2$		$k = 3$	
	E	order	E	order
1/32	4.91e-2	-	3.83e-3	-
1/64	3.12e-3	4.0	2.12e-3	0.9
1/128	1.94e-3	0.7	1.61e-3	0.4

With reinitialization present, shape errors are not necessarily reduced by grid refinement, although even in these cases increasing the polynomial degree reduces the error.

6.5 GPU Acceleration

The CUDA algorithm for DG advection was executed on a Nvidia Tesla K20 and compared to the original algorithm running in serial on an Intel Xeon E5-2620. 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, Zalesak's disk, time-reversing velocity fields, and the circle test as described in previous sections of this chapter, so this test is limited to computation speed and assurance that the CPU and GPU give equivalent results (within 10^5 times machine epsilon at double precision). This is done by randomizing the level set scalar and velocity coefficients, then performing a single RK advection step and comparing the CPU and GPU runtime and output.

Runtimes are calculated for the algorithms of interest alone, and speedup factors calculated for how much a given algorithm is accelerated by being executed on GPU hardware rather than the CPU. This is done to more easily isolate the performance boost provided by the GPU, noting that other routines outside the main solver may also be adapted to GPU hardware. This is particularly important for the random coefficients test shown here. Since the test only performs a single Runge-Kutta step, the bulk of the runtime is spent on initialization and does not well represent the total runtime found in practice.

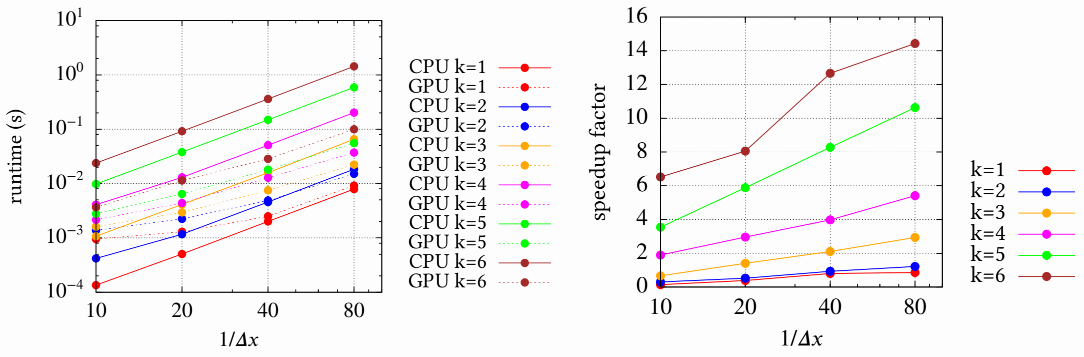


Figure 6.15: Compute Time and Speedup of One 2D Advection RK Step

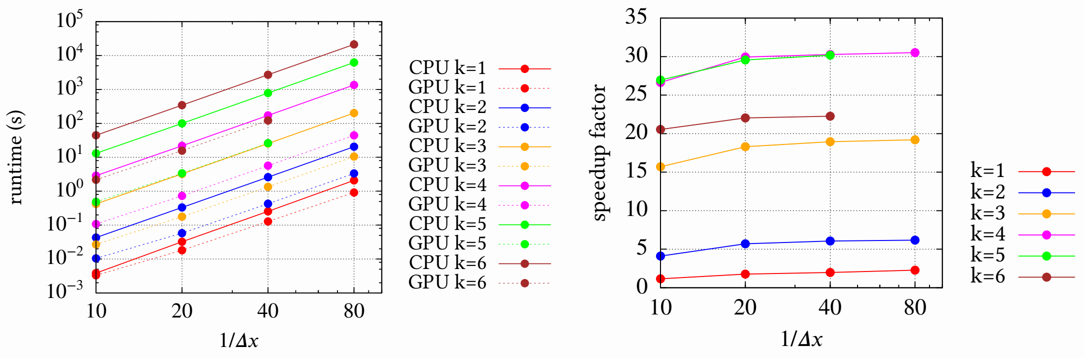


Figure 6.16: Compute Time and Speedup of One 3D Advection RK Step

These tests produce several interesting trends, shown in Table B.1, Table B.2, Fig. 6.15, and Fig. 6.16. First, low degree polynomials show little benefit from the GPU, and sometimes even result in slower runtimes, simply because the integral arrays are not large enough for all threads to be operating simultaneously. A similar drawback arises if sparsity is not exploited, where the GPU instead slows down computation by a factor of 42x for $k = 4$ and $\Delta x = 80$ in 3D since many threads end up multiplying by zero, wasting significant FLOPs. One way to remedy this in practice is to use smaller work-group sizes when dealing with low-degree polynomials. However, this solution has limitations since GPUs are most efficient when the block size is a multiple of 32 (the warp size) [27].

Second, the data indicates the GPU is increasingly advantageous as it is given more work. This is demonstrated in Fig. 6.15 and Fig. 6.16 since speedup increases with grid size. With more degrees of freedom and operations, whether from refining the grid or increasing the number of polynomials, the total speedup increases until a certain point where it levels off at an approximately constant speedup factor. This implies that at $\Delta x = 1/10$ there are not enough blocks for warps on the GPU to effectively mask memory latencies with massive parallelism.

Finally, higher k in general produces higher speedup factors. This compliments the effectiveness of high-order DG and reflects the streaming memory model on the GPU, where one warp's memory latencies are masked by another's floating-point operations. However, $k = 4$ and $k = 5$ produce similar speedup factors, indicating that at that point the integral arrays are large enough that parallelization along the array elements is sufficient to fully occupy the GPU. At $k = 6$, the speedup factors drop since the level set scalar and velocity arrays become too large to fit entirely in shared memory. And finally, the $k = 5, 6$, 80^3 grid data points are not calculated since the total global memory required for all variables is too large to fit on the Nvidia Tesla K20 simultaneously. In practice, this would be avoided by either executing the main kernel multiple times in sequence on different subsets of the mesh, or by performing the calculations on multiple GPUs.

Table 6.16: GPU Event Timing

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

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

To investigate the expense of different operations on the GPU, CUDA event timers are used to calculate the duration of various routines. The results are shown in Table 6.16 for the degree 3 polynomial, 40x40x40 grid advection case. 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.

Finally, it is worth noting that this test produces a lower bound for speedup factors compared to full simulations. This is because this test only performs a single Runge-Kutta step, rather than a full time step. As a result, the CPU-GPU communication overhead is emphasized. For example, the sphere in deformation field under advection alone with

$k = 4$ and $\Delta x = 1/32$ produces a speedup factor of 33.1x, compared to approximately 30x expected from the results in Table B.2.

This test is repeated for reinitialization, with the results shown in Table B.3, Fig. 6.17, and Fig. 6.18. In this case, the speedup factors are not as high for multiple reasons. First, the tests stop at 3rd order here because the integral arrays are much larger. For $k = 4$ 3D, even unenriched flux arrays take up more global memory than is available on the Tesla K20. The speedup up to $k = 3$ shown in the tables is very similar to that of the speedups for advection at the same order, and it may continue to scale alongside advection. In practice, individual integral terms could be evaluated in sequence with the integral arrays copied every time step, but this would greatly slow the process. Another possibility is that the reinitialization kernel does not continue to speed up and match the factors found in advection because the parallelization implementation results in reinitialization relying on more shared memory than is available. This is because reinitialization's nonlinear and diffusive terms depend on the product of two variables rather than one, both of which are brought to local memory. At higher orders, these arrays are too large to fit in shared memory, resulting in some elements being automatically placed in global memory. If the added latency is not sufficiently hidden by parallel operations, the GPU will lose some efficiency. In this case, the operation could be improved by modifying the compressed data structure with an additional variable, indicating where the second-outermost index changes values. Then, the corresponding variable would be stored in a local register one coefficient at a time, rather than attempting to store all coefficients in shared memory. However, since this reduces the length of the loop being parallelized, it increases the risk of reducing occupancy of the GPU.

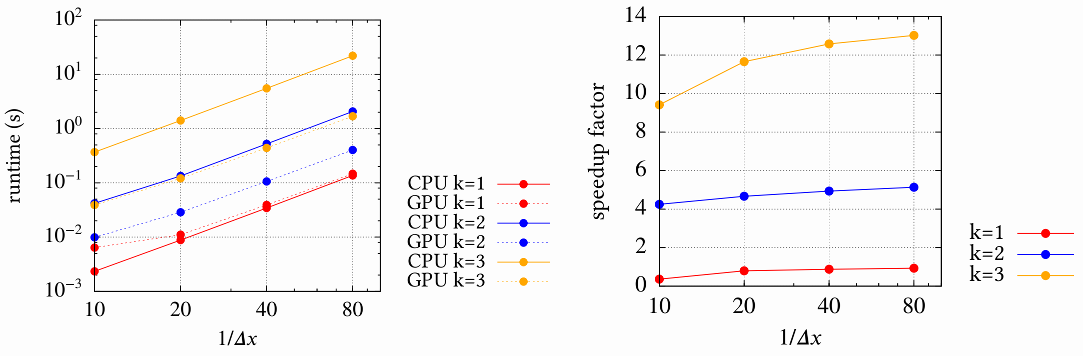


Figure 6.17: Compute Time and Speedup of One 2D Reinitialization RK Step

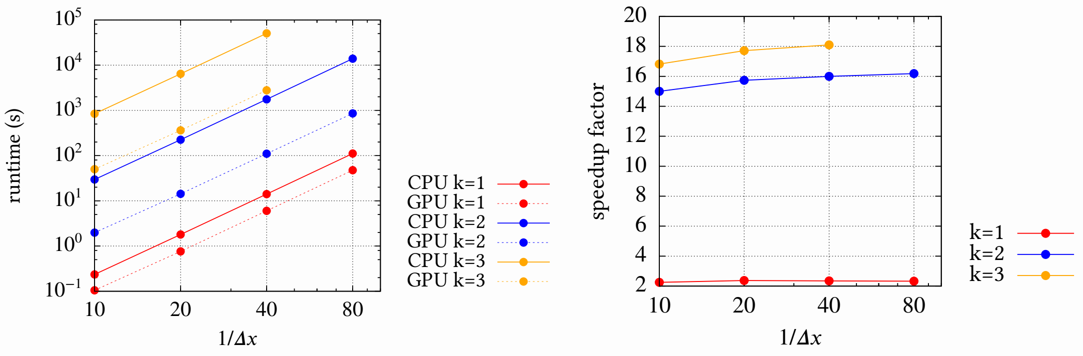


Figure 6.18: Compute Time and Speedup of One 3D Reinitialization RK Step

Chapter 7

Concluding Remarks

7.1 Summary

In order to construct an interface capturing method for 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 constraints Eqs. (3.22), (3.23), and (3.24). The RKDG-CLS scheme for advection is executed on the GPU using CUDA, via Alg. 1. Finally, normal vector fields required by reinitialization are calculated using direct differentiation of the local level set scalar field via Eq. (4.21) and normalized via Eq. (4.2). Ap. A lists integrals that are precomputed and stored in arrays.

The method was tested using the method of manufactured solutions, Zalesak's disk, and deforming columns and spheres in a time-reversing vortex. In general, it was found that the RKDG method can solve both the advection and reinitialization equations with the arbitrarily high convergence rates of $k + 1$ predicted by the method. However, interface transport tests showed normal vectors calculated from the local level set field to be inadequate, since they amplify noise and existing error in the system, resulting in increased shape and volume errors. In practice, therefore, it is necessary for normals to be evaluated in an ACLS sense for reinitialization to properly mitigate dissipative errors without introducing errors from level set trapping.

By taking advantage of sparsity, threading, and coalesced memory access on the GPU, an overall speedup of 30x between GPU/CPU for the advection routine, advocating the applicability and benefit of GPUs in numerical algorithms, particularly calculations independent from one another. Sec. 6.5 indicated that more work given to the GPU results in more speedup, especially when increasing polynomial order, up until the point where the amount of local memory on a chip is insufficient. This compliments the results in Fig. 6.6, showing that the discontinuous Galerkin conservative level set method is more effective at higher orders. Reinitialization showed similar speedup potential, but the integral arrays for higher orders were too large to fit on the GPU all at once.

7.2 Future Work

Future work involves development of several algorithms and techniques, as well as optimization of existing ones. Normal vectors and curvature ought to be calculated from the interface geometry alone, which can be done through a variety of methods. This would include investigation into newer Hamilton-Jacobi solvers [4, 20], and possibly a brute force approach. The GPU timing results call attention to avenues for future software development, which ought to target reducing pressure on the shared and global memory spaces. There is also some potential benefit to investigating a multitude of other accelerator programming paradigms and libraries. 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] Y. Cheng and Z. Wang. *J. Comput. Phys.* ().

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

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

[7] *CUDA C Programming Guide*. ver. 5.5. NVIDIA Corporation. 2013.

[8] 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).

[9] 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.

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

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

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

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

[14] J. Grooss and J.S. Hesthaven. “A level set discontinuous Galerkin method for free surface flows”. *Comput. Methods Appl. Mech. Engrg.* 195 (2006), pp. 3406–3429.

[15] 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.

[16] C. Hu and C.-W. Shu. “A Discontinuous Galerkin Finite Element Method for Hamilton-Jacobi Equations”. *SIAM J. Sci. Comput.* 21 (1999), pp. 666–690.

[17] 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.

[18] F. Li and S. Yakovlev. “A Central Discontinuous Galerkin Method for Hamilton Jacobi Equations”. *J. Sci. Comput.* 45 (2010), pp. 404–428.

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

[20] H. Liu and M. Pollack. “Alternating evolution discontinuous Galerkin methods for Hamilton-Jacobi equations”. *J. Comput. Phys.* 258 (2014), pp. 31–46.

[21] 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.

[22] 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.

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

[24] 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.

[25] J. McCaslin and O. Desjardins. “A localised re-initialization equation for the conservative level set method”. *J. Comput. Phys.* 262 (2014), pp. 408–426.

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

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

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

[29] 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.

[30] M. Owkes and O. Desjardins. "A discontinuous Galerkin conservative level set scheme for interface capturing in multiphase flows". *J. Comput. Phys.* 249 (2013), pp. 275–302.

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

[32] 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.

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

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

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

[36] C.-W. Shu. "TVB uniform high-order schemes for conservation laws". *Math. Comp.* 46 (1987), pp. 105–121.

[37] 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).

[38] 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.

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

[40] 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.

[41] B. van Leer. "Towards the ultimate conservation difference scheme, II". *J. Comput. Phys.* 14 (1974), pp. 361–376.

[42] B. van Leer. "Towards the ultimate conservation difference scheme, V". *J. Comput. Phys.* 32 (1979), pp. 1–136.

[43] J. Yan and S. Osher. "A local discontinuous Galerkin method for directly solving the Hamilton-Jacobi equations". *J. Comput. Phys.* 230 (2011), pp. 232–244.

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

[45] 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:

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

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

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

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

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

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

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

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

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

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

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

Projection

For 3-cell projection, the precomputed integrals are:

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

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

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

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

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

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

Appendix B

GPU Timing Data

Table B.1: Compute Time of One Advection 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	1.35e-4	9.49e-4	0.14x	3.85e-3	3.31e-3	1.16x
	20	5.06e-4	1.30e-3	0.39x	3.23e-2	1.82e-2	1.77x
	40	2.01e-3	2.50e-3	0.80x	2.54e-1	1.28e-1	1.98x
	80	7.93e-3	9.20e-3	0.86x	2.08	9.16e-1	2.27x
2	10	4.19e-4	1.38e-3	0.30x	4.29e-2	1.04e-2	4.13x
	20	1.17e-3	2.24e-3	0.52x	3.31e-1	5.81e-2	5.70x
	40	4.57e-3	4.89e-3	0.93x	2.59	4.27e-1	6.07x
	80	1.84e-2	1.51e-2	1.22x	2.06e+1	3.33	6.19x
3	10	1.07e-3	1.61e-3	0.66x	4.21e-1	2.68e-2	15.7x
	20	4.13e-3	2.94e-3	1.40x	3.24	1.77e-1	18.3x
	40	1.58e-2	7.51e-3	2.10x	2.55e+1	1.35	18.9x
	80	6.55e-2	2.23e-2	2.94x	2.02e+2	1.05e+1	19.2x

Table B.2: Compute Time of One Advection RK-Step (cont'd)

Polynomial Degree	$1/\Delta x$	2D Simulation			3D Simulation		
		CPU time (s)	GPU time (s)	Speedup	CPU time (s)	GPU time (s)	Speedup
4	10	4.09e-3	2.16e-3	1.89x	2.86	1.08e-1	26.5x
	20	1.30e-2	4.39e-3	2.96x	2.18e+1	7.28e-1	30.0x
	40	5.09e-2	1.28e-2	3.98x	1.71e+2	5.65	30.3x
	80	2.02e-1	3.73e-2	5.42x	1.35e+3	4.43e+1	30.5x
5	10	9.82e-3	2.77e-3	3.55x	1.30e+1	4.84e-1	26.9x
	20	3.79e-2	6.44e-3	5.89x	9.97e+1	3.37	29.6x
	40	1.49e-1	1.80e-2	8.28x	7.86e+2	2.60e+1	30.2x
	80	5.89e-1	5.54e-2	10.6x	6.26e+3	-	-
6	10	2.38e-2	3.65e-3	6.52x	4.47e+1	2.17	20.6x
	20	9.18e-2	1.14e-2	8.05x	3.45e+2	1.57e+1	22.0x
	40	3.61e-1	2.85e-2	12.7x	2.71e+3	1.21e+2	22.4x
	80	1.44	9.98e-2	14.4x	2.14e+4	-	-

Table B.3: Compute Time of One Reinitialization 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	2.34e-3	6.40e-3	0.37x	2.36e-1	1.05e-1	2.26x
	20	8.83e-3	1.11e-2	0.80x	1.80e+0	7.60e-1	2.36x
	40	3.44e-2	3.92e-2	0.88x	1.41e+1	6.01	2.34x
	80	1.37e-1	1.47e-1	0.93x	1.11e+2	4.77e+1	2.33x
2	10	4.20e-2	9.89e-3	4.25x	2.97e+1	1.98	15.0x
	20	1.34e-1	2.87e-2	4.67x	2.25e+2	1.43e+1	15.8x
	40	5.23e-1	1.06e-1	4.95x	1.76e+3	1.10e+2	16.1x
	80	2.07	4.03e-1	5.14x	1.39e+4	8.59e+2	16.1x
3	10	3.68e-1	3.91e-2	9.39x	8.41e+2	5.00e+1	16.8x
	20	1.41	1.21e-1	11.7x	6.43e+3	3.63e+2	17.7x
	40	5.52	4.39e-1	12.6x	5.05e+4	2.79e+3	18.1x
	80	2.20e+1	1.69	13.0x	-	-	-