High-Performance High-Order Simulation of Wave and Plasma Phenomena

by

Andreas Klöckner

Dipl.-Math. techn., Universität Karlsruhe (TH); Karlsruhe, Germany, 2005

M.S., Brown University; Providence, RI, 2006

A dissertation submitted in partial fulfillment of the

requirements for the degree of Doctor of Philosophy

in The Division of Applied Mathematics at Brown University

PROVIDENCE, RHODE ISLAND

May 2010

This dissertation by Andreas Klöckner is accepted in its present form
by The Division of Applied Mathematics as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date________________ ____

Jan Sickmann Hesthaven, Ph.D., Advisor

Recommended to the Graduate Council

Date________________ ____

Johnny Guzmán, Ph.D., Reader

Date________________ ____

Chi-Wang Shu, Ph.D., Reader

Approved by the Graduate Council

Date________________ ____

Sheila Bonde, Dean of the Graduate School

Vitae

Biographical Information

Birth	August 5th, 1977 Konstanz, Germany

Education

2005 – 2010	Ph.D. in Applied Mathematics (in progress) Division of Applied Mathematics, Brown University, Providence, RI *Advisor: Jan Hesthaven*
2005	Diplom degree in Applied Mathematics (Technomathematik) Institut für Angewandte Mathematik, Universität Karlsruhe, Germany *Advisor: Willy Dörfler*
2001 – 2002	Exchange Student, Department of Mathematics University of North Carolina at Charlotte, Charlotte, NC
2000	Vordiplom in Computer Science, Universität Karlsruhe, Germany

Experience

6/2006 – 9/2006	J. Wallace Givens Research Associate *Mathematics and Computer Science Div., Argonne Nat'l Laboratory, Illinois* Worked on high-order unstructured electromagnetic simulation of particle accelerators (with Paul Fischer, Misun Min, and colleagues at ANL's Advanced Photon Source).

2/2005 – 7/2005	Research Associate (Wissenschaftlicher Mitarbeiter) *Institut für Angewandte Mathematik, Universität Karlsruhe, Germany* Worked on various extensions of my thesis research (with Willy Dörfler).
5/2002 – 11/2002	Research Intern *DaimlerChrysler Research & Technology, Palo Alto, CA* Worked on driver stress detection, precision GPS, and software infrastructure (with Stefan Schrödl).

Publications

2010	Viscous Shock Capturing with an Explicitly Time-Stepped Discontinuous Galerkin Method. AK, T. Warburton, J.S. Hesthaven. *In preparation.*
2009	PyCUDA: GPU Run-Time Code Generation for High-Performance Computing. AK, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. *Submitted, available at* `http://arxiv.org/abs/0911.3456`.
2009	Nodal Discontinuous Galerkin Methods on Graphics Processors. AK, T. Warburton, J. Bridge, J.S. Hesthaven. *Journal of Computational Physics, Volume 228, Issue 21, 20 November 2009.*
2009	Deterministic Numerical Schemes for the Boltzmann Equation. A. Narayan, AK. *Brown University Scientific Computing Technnical Report 2009-39.*
2005	On the Computation of Maximally Localized Wannier Functions. *Diplom Thesis, Universität Karlsruhe, Germany.*

Acknowledgments

First and foremost, the support of my advisor Jan Hesthaven has been the cornerstone of my working life in the past five years. He was a source of questions, of answers, of inspiration, he encouraged me to be bold in the scientific questions I pursue, all while giving me great freedom in following my interests. He has also patiently put up with the things that turned out not to be so smart in hindsight. Beyond science, he has been a role model and a tremendous influence on my life as a whole. I consider myself lucky to have had him as a mentor.

Over the years, I have worked very closely with Tim Warburton at Rice University on many of the topics that this thesis discusses. His generosity, help, and insight have benefited me in many ways.

Both of the above, along with Chi-Wang Shu and Johnny Guzmán have graciously agreed to serve on my PhD committee, spent time thinking about my work, and provided invaluable feedback.

Throughout my graduate studies, I have had the honor of working on various projects with a large and diverse group of people. Their insights, commentary and encouragement, shared in many conversations, were and continue to be a great asset to my scientific life.

The graduate student and postdoc community at Brown's Division of Applied Mathematics is a great crowd in which to grow up academically. Many of you have become my friends, and I hope we will be able to stay close even as life scatters us across the globe.

Nvidia Corporation have been very generous with equipment and travel support and were instrumental in initiating, furthering and publicizing the GPU-based part of my research.

Many contributors around the world have created the open-source software and tools on which my work has crucially depended. This notably includes the communities that have formed around my various projects.

Parts of this thesis are based on two publications, [Klöckner et al., 2009b] and [Klöckner et al., 2009a]. My coauthors have contributed considerably to both articles through their ideas, suggestions, and feedback.

Last, but by no means least, my parents Bina and Heinrich Klöckner have, throughout my entire life, given me their unconditional support, advice, and love.

Thank you, all of you.

Some of the computational meshes used in this work were generated using Triangle [Shewchuk, 1996] and TetGen [Si and Gaertner, 2005]. The surface mesh for Figure 5.10 originates in the FlightGear flight simulator and was processed using Blender and MeshLab, a tool developed with the support of the Epoch European Network of Excellence.

Abstract of "High-Performance High-Order Simulation of Wave and Plasma Phenomena"
by Andreas Klöckner, Ph.D., Brown University, May 2010

This thesis presents results aiming to enhance and broaden the applicability of the discontinuous Galerkin ("DG") method in a variety of ways. DG was chosen as a foundation for this work because it yields high-order finite element discretizations with very favorable numerical properties for the treatment of hyperbolic conservation laws.

In a first part, I examine progress that can be made on implementation aspects of DG. In adapting the method to mass-market massively parallel computation hardware in the form of graphics processors ("GPUs"), I obtain an increase in computation performance per unit of cost by more than an order of magnitude over conventional processor architectures. Key to this advance is a recipe that adapts DG to a variety of hardware through automated self-tuning. I discuss new parallel programming tools supporting GPU run-time code generation which are instrumental in the DG self-tuning process and contribute to its reaching application floating point throughput greater than 200 GFlops/s on a single GPU and greater than 3 TFlops/s on a 16-GPU cluster in simulations of electromagnetics problems in three dimensions. I further briefly discuss the solver infrastructure that makes this possible.

In the second part of the thesis, I introduce a number of new numerical methods whose motivation is partly rooted in the opportunity created by GPU-DG: First, I construct and examine a novel GPU-capable shock detector, which, when used to control an artificial viscosity, helps stabilize DG computations in gas dynamics and a number of other fields. Second, I describe my pursuit of a method that allows the simulation of rarefied plasmas using a DG discretization of the electromagnetic field. Finally, I introduce new explicit multi-rate time integrators for ordinary differential equations with multiple time scales, with a focus on applicability to DG discretizations of time-dependent problems.

Contents

List of Tables

List of Figures

List of Algorithms

CHAPTER ONE

Introduction

1.1 About this Thesis

The aim of this thesis is to present results that enhance and broaden the applicability of the discontinuous Galerkin method (“DG”, cf. Section 2.1) in a variety of ways.

Roughly the first third of this thesis (Chapters 3 through 5) concerns itself with implementation aspects of DG. Chapter 3 describes how the method can be implemented in a way that combines the often opposing goals of using established software engineering practice while achieving high performance, and Chapter 5 explains how computation performance can be increased by an order of magnitude or more through the use of mass-market massively parallel computation hardware. In support of this latter advance, some parallel programming tools are introduced that were created specifically to support it, but have found a much broader use in the scientific community (Chapter 4).

The latter two thirds of the thesis (Chapters 6 through 8) introduce a number of new numerical methods focused on certain application problems. All of these methods were designed to be used in conjunction with DG, and some are general enough to be used in a broader context. Chapter 6 focuses on the treatment of shock-laden flows. I construct and examine a novel shock detection method, which, in conjunction with an artificial viscosity, helps stabilize DG computations in gas dynamics and a number of other fields. Chapter 7 describes my pursuit of a method that allows the simulation of rarefied plasmas using a DG discretization of the electromagnetic field. Lastly, Chapter 8 introduces new methods for ordinary differential equations with multiple time scales and is applicable, among other things, to DG discretizations of time-dependent problems.

1.2 The Scientific Method and the Computational Experiment

In the second part of this thesis, new knowledge is gained by examining the behavior of newly-introduced methods in purposefully chosen computational experiments. In applied mathematics, such experiments are often maligned for being poorly specified, difficult or impossible to reproduce, and for these reasons of questionable value.

Unlike (for example) physics or chemistry, applied mathematics does not have a mature culture of experimentation, let alone a fully accepted experimental branch. Before the advent of computers, experiments in mathematics were often impractically tedious to conduct, and therefore mathematical culture has grown mainly around theoretical results. But now that the possibility exists, the field would–in my opinion–do well to mimic the other sciences and embrace experimentation as one of its accepted methods. It should do so in the interest of capturing knowledge that might otherwise not be available, but it should also do so carefully.

Applied mathematicians conduct computational experiments in large quantities every day. Regrettably however, most of these experiments are ad-hoc, and poor (if any) records are kept about them. A part of this problem is that physicists and chemists are routinely trained in the task of thorough, well-documented, reproducible experimentation, while applied mathematicians are not.

In a recent paper, LeVeque [2009] addresses this issue and notes that scientific results obtained by experimentation are first and foremost expected to be reproducible. This has long been demanded of published research. Yet, scientific computing, while often experimental, seems to have been strangely exempt from this requirement. Based on

LeVeque's recommendations, I would like to suggest the following guidelines that I have tried to follow in my work on this thesis:

Availability in source code form. Computer programs used to obtain results in published research should be available for other researchers to inspect and, if possible, execute. Computational experiments are influenced by numerous small details to which the written word describing the experiment seldom does justice. Therefore, the ability to inspect the code that produced the result is key to finding details which the original authors may have deemed irrelevant, but which might turn out to be crucial in deciding about success or failure. Compared to inspection, exact duplication by re-running the experiment is a desirable, but secondary, goal.

Unlike in the physical sciences, experimental setups in computational science can be transported and duplicated easily if some care is invested. So far, this is a large missed opportunity. Ideally, the same distribution channels that so far convey the research article describing the results should also convey the code and data used to obtain them.

Readability. The ability to inspect code of course diminishes in value if that code is sufficiently inscrutable. Along with wide distribution of code, an understanding needs to grow that code is a valid expression of the ideas that it contains. The more apparent these ideas are from the code, the better. Or, in other words, codes should be written mostly to be read, not so much to be executed.

Reduced dependencies on for-pay software. While one will never be able to fully capture the multitude of details influencing a computation, steps can be taken to minimize factors that are beyond the experimenter's control. By its definition, commercial software is controlled by an outside interest wishing to derive monetary gain. One way of achieving this is by creating a lock-in situation in which the software becomes irreplaceable to the user. Quite obviously, this is at odds with reproducibility and wide availability. Fortunately, a sufficiently rich ecosystem of software unencumbered by commercial interests has been emerging, whose use for my purposes I will briefly explore in Section 1.4.

1.3 An Argument for Hybrid Codes

Scientific codes are–by design–experimental in nature. This results in a fundamental difference between scientific and other types of software. The former explores uncharted ground, and therefore has a higher probability of resulting in failure. Assuming that everything is already being done to minimize the probability of that event, another measure to reduce expected loss is to minimize time and effort spent on the experimental code in the first place, at the expense of other qualities, such as computational speed, readability, or maintainability.

I would argue that of those three properties, it is wisest to sacrifice computational speed. The popularity of environments such as Matlab, which is aimed at easing experimentation, confirms this assertion. Speed should be sacrificed because of those three properties, it is easiest to recover later. A prototype of a proposed method can usually very quickly be created in scripting-type languages such as Matlab or Python. One of the core advantages of these languages is the seamless use of high-level abstractions. (The reader may think of Matlab's arrays and vector indexing as an example of such an abstraction.) In particular, this use of abstractions is one of the ways in which these languages avoid sacrificing code readability for fast development time. Other factors further contribute to the rapid experimentation, such as the lack of a separate compilation step. In summary, their design makes these languages productive experimentation grounds.

Once a proposed computational method has proven its worth, speed becomes a concern. It is clearly desirable to respond to this demand in an evolutionary manner, i.e. without rewriting large amounts of computer code. This desire is furthered by the common observation that often only one small part of the program is responsible for most of its run time. This observation naturally leads to *hybrid* code–code that mixes a scripting-type language with a compiled language to regain performance. This idea is far from new and is enabled, for example, by Matlab's MEX facility. The downfall of such hybrid systems is, most often, their complexity. I would next like to introduce an assembly of open tools which removes much of the burden of hybrid development. Some of these tools have existed for years, whereas others were created specifically for this thesis.

1.4 Assembling a Set of Tools

In bringing together a tool set for hybrid development, I was guided by three main factors: maintenance, quality, and suitability. These criteria are first applied to the choice of the high- and the low-level language. At the high level, I chose Python [van Rossum et al., 1994], whose ecosystem has grown to provide many essential capabilities such as array computations (through a package called "numpy" [Oliphant, 2006]), parallel computation [Dalcín et al., 2005, 2008] and plotting. As the compiled, "low-level" language, C++ is a standard, safe choice.

The next decision to be made involves how the two languages are to be connected. I have decided in favor of the library "Boost.Python" [Abrahams et al., 2003], which was created as contract work for Lawrence Berkeley National Laboratory's Computational Crystallography Initiative in 2003. Boost.Python is part of a larger assembly of peer-reviewed, high-quality libraries called "Boost C++", which also contains Boost.UBlas, a package for linear algebra in C++. Because its use does not add further dependencies, it was expedient to use UBlas for my C++-based linear algebra needs. PyUblas, a package created for this thesis, creates a bridge between numpy and UBlas, completing the “glue layer”.

Further needs of a hybrid discontinuous Galerkin solver include simplicial mesh generation and visualization. The former is provided by a package called “MeshPy”, which connects Python to the simplicial mesh generators Triangle [Shewchuk, 1996], TetGen [Si and Gaertner, 2005] and Gmsh [Geuzaine and Remacle, 2009]. For the latter, a package called “Pylo” enables Python codes to write “Silo” visualization data files for use with the VisIt large scale visualization program [Childs et al., 2005]. I have created both packages for the work in this thesis. In addition, I have created the packages PyCUDA, PyOpenCL, and CodePy that enable crucial run-time code generation capabilities. These packages will be described in more detail in Chapter 4. Lastly, this infrastructure is then put to use by the DG solver “hedge” whose basic design and use is surveyed in Chapter 3. To achieve the above goal of availability, I have released all of my software under open-source licenses, making it free to download, use, modify, and redistribute, in addition to opening it up to user inspection.

1.5 Reproducibility for Results in this Thesis

To ensure that my results are reproducible, I will use this section to specify both the computational environment in which I have conducted my experiments and the precise versions of the software that I have used. First, Table 1.1 gives a comprehensive summary of third-party components which performed operations that led to results shown in this work. If multiple version numbers of a component are given, no significant change in the results was observed across all specified versions.

Next, as I have already indicated, all my code is freely available. It may be downloaded from my version control repositories from the address

```
http://git.tiker.net/trees/NAME.git
```

where the name of the component as seen in Table 1.2 should be substituted for `NAME`. Observe that these URLs are designed for automated download of revision history by the "git" version control system. A more human-friendly interface to the code is found under the URL

```
http://git.tiker.net/NAME.git
```

where the user may directly view my source code without having to download it. Table 1.2 contains unique version specifiers, to be used in conjunction with git, of my software which produced the results of this thesis.

Dependency	Versions
Processor	Intel Xeon E5472, Intel Core 2 Quad Q6600, both in x86-64 mode
GNU C library	2.3.4, 2.10.2
GNU C Compiler	4.3.3, 4.4.1
libgmp	4.2.4, 4.3.2
libmpfr	2.3.2, 2.4.2
Boost C++ library	1.38, 1.42
LAPACK	3.1.1 lite
ATLAS	3.8.3
SQLite	3.6.11, 3.6.23
Python	2.6.1, 2.5.4
numpy	1.3.0, 1.2.1
CUDA/nvcc	2.3, 3.0

Table 1.1. Versions of outside hardware and software that had a direct influence on my results.

Submodule	git Version ID
`pytools`	d232ecb87daff9eacee8fc5d9a6e02850f8398d5
`pymbolic`	1bc6017e8b2fa575ae1366ef9d2386c1187c00e6
`pyublas`	46f2ea542b54341fec1fdb0d54b2bcdfe90417cc
`meshpy`	bd9fa47082b8b1f8b18959de8ceb64d4c0f47d3b
`pylo`	fb7582aab1f7093d81b6ee297c1b2331c15649b9
`codepy`	9a880ece091d3f26d4aaed014001a7a370d04f22
`pycuda`	b235481e82edf92016d21a8954ac048a2860c4a1
`pyopencl`	fb115c21cdffea8ee8cd25ecba863d1829601059
`pymetis`	0d9c6467ff03e2c580a3be127129082ad4bbe6f2
`hedge`	782cc5a4ffedbacba3140054b2172263076a6230
`pyrticle`	edf4d7a95004232e0c5bfb3de8c202b27b955589
`thesis-experiments`	dc55619a4b0f4738810d66a9ba276236a7aa7e41

Table 1.2. Precise versions of my own software that I have used to obtain the results herein. Version numbers are given as SHA1 version hash IDs for the "git" version control system.

CHAPTER TWO

Preliminaries

To facilitate better understanding of later chapters, and to fix notation and terminology, this chapter summarizes a few preliminaries that underlie the subsequent material. Nothing in this chapter is original. Readers familiar with the subject may browse though the section headings and skip material that is familiar to them, referring back when specific terminology or notation needs to be clarified.

2.1 The Discontinuous Galerkin Method

Discontinuous Galerkin (DG) methods [Cockburn et al., 1990, Hesthaven and Warburton, 2007, Lesaint and Raviart, 1974, Reed and Hill, 1973] are, at first glance, a rather curious combination of ideas from Finite-Volume and Spectral Element methods. Up close, they are very much high-order methods by design. But instead of perpetuating the order increase like conventional global methods, at a certain level of detail, they switch over to a decomposition into computational elements and couple these elements using Finite-Volume-like surface Riemann solvers. This hybrid, dual-layer design allows DG to combine advantages from both of its ancestors. But it adds a third advantage: By adding a movable boundary between its two halves, it gives implementers an added degree of flexibility when bringing it onto computing hardware.

By their design and origins, DG methods are particularly suited to approximating the solution of a hyperbolic system of conservation laws

$$u_t + \nabla \cdot F(u) = 0 \tag{2.1}$$

on a domain $\Omega = \biguplus_{k=1}^{K} \mathsf{D}_k \subset \mathbb{R}^d$ consisting of disjoint, face-conforming tetrahedra D_k

with boundary conditions

$$u|_{\Gamma_i}(x,t) = g_i(u(x,t),x,t), \qquad i = 1,\dots,b,$$

at inflow boundaries $\biguplus \Gamma_i \subseteq \partial\Omega$. As stated, I will assume the flux function F to be linear. I find a weak form of (2.1) on each element D_k:

$$\begin{aligned} 0 &= \int_{\mathsf{D}_k} u_t\varphi + [\nabla\cdot F(u)]\varphi \,\mathrm{d}x \\ &= \int_{\mathsf{D}_k} u_t\varphi - F(u)\cdot\nabla\varphi \,\mathrm{d}x + \int_{\partial\mathsf{D}_k} (\hat{n}\cdot F)^*\varphi \,\mathrm{d}S_x, \end{aligned}$$

where φ is a test function, and $(\hat{n}\cdot F)^*$ is a suitably chosen numerical flux in the unit normal direction $\hat{n}$. Following [Hesthaven and Warburton, 2007], I find a 'strong'-DG form of this system as

$$0 = \int_{\mathsf{D}_k} u_t\varphi + [\nabla\cdot F(u)]\varphi \,\mathrm{d}x - \int_{\partial\mathsf{D}_k} [\hat{n}\cdot F - (\hat{n}\cdot F)^*]\varphi \,\mathrm{d}S_x. \tag{2.2}$$

I seek to find a numerical vector solution $u^k := u_N|_{\mathsf{D}_k}$ from the space $P_N^n(\mathsf{D}_k)$ of local polynomials of maximum total degree N on each element. I choose the scalar test function $\varphi \in P_N(\mathsf{D}_k)$ from the same space and represent both by expansion in a basis of $N_p := \dim P_N(\mathsf{D}_k)$ Lagrange polynomials l_i with respect to a set of interpolation nodes [Warburton, 2006]. I define the mass, stiffness, differentiation, and face mass matrices

$$M_{ij}^k := \int_{\mathsf{D}_k} l_i l_j \,\mathrm{d}x, \tag{2.3a}$$

$$S_{ij}^{k,\partial\nu} := \int_{\mathsf{D}_k} l_i \partial_{x_\nu} l_j \,\mathrm{d}x, \tag{2.3b}$$

$$D^{k,\partial\nu} := (M^k)^{-1} S^{k,\partial\nu}, \tag{2.3c}$$

$$M_{ij}^{k,A} := \int_{A\subset\partial\mathsf{D}_k} l_i l_j \,\mathrm{d}S_x. \tag{2.3d}$$

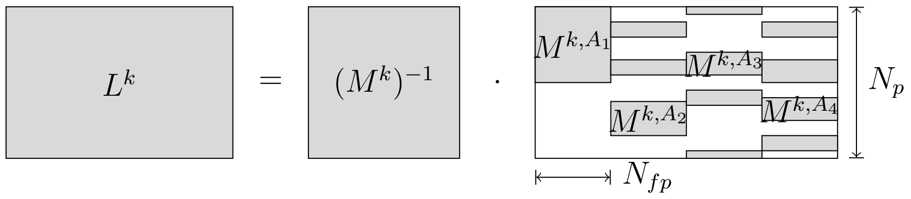


Figure 2.1. Construction of the Lifting Matrix L^k.

Using these matrices, I rewrite (2.2) as

$$0 = M^k \partial_t u^k + \sum_\nu S^{k,\partial_\nu}[F(u^k)] - \sum_{F \subset \partial \mathsf{D}_k} M^{k,A}[\hat{n} \cdot F - (\hat{n} \cdot F)^*],$$
$$\partial_t u^k = -\sum_\nu D^{k,\partial_\nu}[F(u^k)] + L^k[\hat{n} \cdot F - (\hat{n} \cdot F)^*]|_{A \subset \partial \mathsf{D}_k}. \tag{2.4}$$

The matrix L^k used in (2.4) deserves a little more explanation. It acts on vectors of the shape $[u^k|_{A_1}, \dots, u^k|_{A_4}]^T$, where $u^k|_{A_i}$ is the vector of facial degrees of freedom on face i. For these vectors, L^k combines the effect of applying each face's mass matrix, embedding the resulting facial values back into a volume vector, and applying the inverse volume mass matrix. Since it "lifts" facial contributions to volume contributions, it is called the *lifting matrix*. Its construction is shown in Figure 2.1.

It deserves explicit mention at this point that the left multiplication by the inverse of the mass matrix that yields the explicit semidiscrete scheme (2.4) is an element-wise operation and therefore feasible without global communication. This strongly distinguishes DG from other finite element methods. It enables the use of explicit (e.g., Runge-Kutta) time stepping and greatly simplifies parallel implementation efforts (cf. Chapter 5).

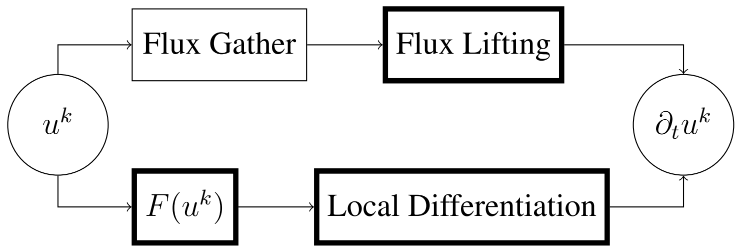


Figure 2.2. Decomposition of a DG operator into subtasks. Element-local operations are highlighted with a bold outline.

2.1.1 Implementing DG

DG decomposes very naturally into four stages, as visualized in Figure 2.2. This clean decomposition of tasks stems from the fact that the discrete DG operator (2.4) has two additive terms, one involving an element volume integral, the other an element surface integral. The surface integral term then decomposes further into a 'gather' stage that computes the term

$$[\hat{n} \cdot F(u_N^-) - (\hat{n} \cdot F)^*(u_N^-, u_N^+)]|_{A \subset \partial \mathsf{D}_k}, \tag{2.5}$$

and a subsequent lifting stage. The notation u_N^- indicates the value of u_N on the face A of element D_k, u_N^+ the value of u_N on the face opposite to A.

As is apparent from the use of a Lagrange basis, I employ a *nodal* version of DG, in which the stored degrees of freedom ("*DOF*s") represent the values of u_N at a set of interpolation nodes. This representation allows us to find the facial values used in (2.5) by picking the facial nodes from the volume field. (This contrasts with a *modal* implementation in which DOFs represent expansion coefficients in a non-Lagrange basis. Finding the facial information to compute (2.5) requires a different approach in these schemes.)

Observe that most of DG's stages are *element-local* in the sense that they do not use information from neighboring elements. Moreover, these local operations are often efficiently represented by a dense matrix-vector multiplication on each element.

It is worth noting that since simplicial elements only require affine transformations Ψ_k from reference to global element, the global matrices can easily be expressed in terms of reference matrices that are the same for each element, combined with scaling or linear combination, for example

$$M_{ij}^k = \underbrace{\left|\det \frac{\mathrm{d}\Psi_k}{\mathrm{d}r}\right|}_{J_k:=} \underbrace{\int_{\mathsf{I}} l_i l_j \,\mathrm{d}x}_{M_{ij}:=}, \tag{2.6a}$$

$$S_{ij}^{k,\partial\nu} = J_k \sum_{\mu} \frac{\partial \Psi_\nu}{\partial r_\mu} \underbrace{\int_{\mathsf{I}} l_i \partial_{r_\mu} l_j \,\mathrm{d}x}_{S_{ij}^{\partial\mu}:=}, \tag{2.6b}$$

where $\mathsf{I} = \Psi_k^{-1}(\mathsf{D}_k)$ is a reference element. I define the remaining reference matrices D, M^A, and L in an analogous fashion.

2.2 GPU Hardware: A Brief Introduction

One of the most significant problems that modern processor design needs to address is the slowness of memory. While there have been significant advances in latency and access speed to affordable, large-scale, off-chip random access memory, these advances have in no way kept pace with the progress made in the throughput of processor cores. Variants of Moore's Law predicted this latter progress to be exponential in nature, and so far reality has kept pace with prediction. This pace was not matched by the development of dynamic RAM (DRAM), the presently dominant technology for such memory. Therefore, the time between the issuing of a memory request by a core and the subsequent response from off-chip memory can be very long, measured in processor time scales.

One of the main differentiators between processor types is how they deal with the problem of the slowness of memory. The two types of processors that presently have mass-market relevance are CPUs and GPUs, which are defined by the target workloads for which they are designed. For CPUs, the set of design workloads typically includes web browsers, word processors and a diverse collection of other desktop programs–characterized by high complexity and marginal potential for parallelization. GPUs, on the other hand, are aimed at applying uniform, moderately complex floating point operations to large volumes of data (i.e. “stream processing” [Venkatasubramanian, 2003]). It is obvious that the latter promise to be the kind more suited to a large range of scientific computing tasks.

In the early days of GPU programming, the programmer had to repurpose marginally programmable fixed-function graphics hardware for computing purposes by a variety of methods [Owens et al., 2007]. With today’s generation of GPUs, this is not true any more. Instead, GPUs should be viewed as general-purpose floating point processors that are designed for a different type of target workload than current CPUs, and “GPU” becomes just a convenient moniker for this type of technology.

Returning to the subject of memory slowness, one should consider that while bandwidth can be increased to some extent by widening and improving the memory interface, latency cannot, as it is a fundamental property of the type of memory. Obviously, the design workloads for CPUs are very vulnerable to memory delays, and therefore CPU designers tend to take extreme measures to mitigate their effects. Three types of strategies are particularly popular here: First, include large amounts of fast cache memory on the chip to avoid having to wait for off-chip memory at all. Second, engage in many forms of prediction and speculation to make sure that required data is already present on-chip when it is needed. And finally, reorder the instruction stream to lessen the impact of memory-related stalls.

It is apparent that the hardware implementation of all these strategies can easily occupy large amounts of silicon. In contrast, the target workloads for a GPU are much less vulnerable to memory-related stalls. Since GPUs aim to apply similar operations to large amounts of data, exact ordering is less important. This allows the use of a much larger number of execution contexts, each of which may occupy a functional (e.g. floating-point or integer) unit whenever it has data available. While the management of large numbers of contexts is nontrivial in itself, the associated management logic is less expensive to implement than the CPU's strategies, freeing a GPU to dedicate much more chip space to functional units, further increasing parallelism.

This abundance of functional units confronts GPU designers with yet another interesting challenge. Context management logic grows strongly superlinearly with the number of contexts it manages. One set of central logic that would manage the execution of all contexts on all functional units on the chip would be prohibitively large. This, together with physical limits of on-chip signal propagation speed, strongly suggests dividing up the available chip are into individual sub-processors, each of which manages a more limited set of execution contexts. It is the same thinking that drives heavyweight CPUs towards integrating multiple cores on a single die. Likewise, modern GPUs contain tens of management subdomains, each of which may manage hundreds of execution contexts. (These subdomains are called 'compute units' by OpenCL, 'multiprocessors' by Nvidia, and simply 'cores' by others. Execution contexts are called 'threads' by Nvidia and 'work items' by OpenCL.) To further improve the functional-unit-to-control-logic ratio and reach the cited width of hundreds of contexts per subdomain, most GPUs are built as relatively wide SIMD (Single Instruction Multiple Data) vector machines.

The chip$\rightarrow$unit$\rightarrow$context hierarchy has a twofold effect on GPU software: First, each unit is typically designed to operate independently of its siblings, limiting communication to contexts executing on the same unit. Second, programs must explicitly specify how to use each level of parallelism, typically by providing a suitable decomposition of an index space. Together with the remaining possibility of sequential execution, this poses the problem of *loop slicing*. Given a sequential description of the algorithm as a set of nested loops, loop slicing refers to the combined process of

- identifying loop axes that can serve as parallelization indices,
- assigning loop axes to available parallelization axes, such as compute units, execution context numbers within a unit, and SIMD lanes,
- interchanging loop orders to achieve a more beneficial order of memory accesses, and lastly,
- finding size restrictions on each loop axis, and splitting axes as necessary.

Observe that each of the above steps may depend on the outcome of all the others, resulting in a complicated joint optimization problem. The purpose of Chapter 4 is to explore these (and other) software challenges and propose solutions for some of them.

2.2.1 Specifics of Nvidia hardware

On the Nvidia hardware [Lindholm et al., 2008] targeted in some of this work (especially Chapter 5), up to 30 independent, parallel *multiprocessors* form the highest level of the processing hierarchy. Each of these multiprocessors is capable of maintaining several hundred threads in flight at any given time, giving rise to the lower level.

One such multiprocessor consists of eight functional units controlled by a single instruction decode unit. Each of the functional units, in turn, is capable of executing one basic single-precision floating-point or integer operation per clock cycle. The instruction decode unit feeding the eight functional units is capable of issuing one instruction every four clock cycles, and therefore the smallest scheduling unit on this hardware is what Nvidia calls a *warp*, a set of 32 threads. The architecture is distinguished from conventional single-instruction-multiple-data (*SIMD*) hardware by allowing threads within a warp to take different branches, although in this case each branch is executed in sequence. To emphasize the difference, Nvidia calls this a single-instruction-multiple-thread (*SIMT*) architecture.

Up to 16 of these warps are now aggregated into a logical unit called a *thread block* and sent to execute on a single multiprocessor. Threads in a block share a piece of execution hardware, and are hence able to take advantage of communication facilities present in a multiprocessor, namely, a memory fence that may optionally serve as a barrier, and 16KiB[1] of banked[2] shared memory. The shared memory has 16 banks, such that half a warp accesses shared memory simultaneously. If all 16 threads access different banks, or if all 16 read the same memory location (a *broadcast*), the access proceeds at full speed. Otherwise, the whole warp waits as maximal subsets of non-conflicting accesses are carried out sequentially.

A potentially very large number of thread blocks is then aggregated into a *grid* and forms the unit in which the controlling host processor submits work to the GPU. There is no guaranteed ordering between thread blocks in a grid, and no communication is allowed between them. Only after successful completion of a grid submission, the work of all thread blocks is guaranteed to be visible. In that sense, grid submission serves as a synchronization point.

Indices within a thread block and within a grid are available to the program at run time and are permitted to be multi-dimensional to avoid expensive integer divisions. I will refer

[1]"KiB" stands for *Kilobyte binary* or *Kibibyte* and represents $1024 = 2^{10}$ bytes. [The International Electrotechnical Commission, 2000]

[2]"Banking" is one technique of designing memory for parallel access. It refers to a partitioning into *banks*, in which each such bank receives its own addressing logic and data bus. As a result, only addresses in distinct banks can be accessed simultaneously. Banking is a typical feature of parallel on-chip memory.

to these indices by the symbols t_x, t_y, t_z, and b_x, b_y.

All threads have read-write access to the GPU's on-board ('*global*') memory. A single access to this off-chip memory has a latency of several hundred clock cycles. To hide this latency, a multiprocessor will schedule other warps if available and ready. A few things influence how many threads can be started: Each thread requires a number of registers. Also, the work of a group of threads often involves a certain amount of shared memory. More threads may therefore also consume more shared memory. Since both the register file and the amount of shared memory is finite, their use may lead to artificial limits on the number of threads in a block. If there are very few threads in a block and there isn't space for many blocks on the same multiprocessor, the device may fail to find warps it can run while waiting for memory transactions. This decreases global memory bandwidth utilization. Another aspect influencing the available bandwidth to global memory is the pattern in which access occurs. Taking 32-bit accesses as an example, loads and stores to global memory achieve the highest bandwidth if, within a warp, thread i accesses memory location $b + \pi(i)$, where b is a 16-aligned base address and π is a mapping obeying $\lfloor \pi(i)/16 \rfloor = \lfloor i/16 \rfloor$. Note that for global *fetches* only, these restrictions can be alleviated somewhat through the use of *texture units*.

A final bit of perspective: While graphics hardware achieves an order of magnitude larger bandwidth to its global memory than conventional CPUs do to their main memory, its floating point capacity eclipses this already large bandwidth by yet another order of magnitude. If one visualizes both compute and memory bandwidth as physical "pipes" with a certain diameter, the challenge in designing algorithms for this architecture lies in keeping each pipe flowing at capacity while using a minimum of buffer space.

CHAPTER THREE

A Code-Generating Discontinuous Galerkin Solver

3.1 On the Design of a Discontinuous Galerkin PDE Solver

The purpose of this chapter is to explain the make-up of the software that forms the tool foundation of many of the later chapters in this thesis. Aside from this supporting role, I believe the way in which this discontinuous Galerkin solver, called “hedge”, is built, is interesting in its own right, and I will use this chapter to describe some (in my opinion) interesting decisions in its design.

Software design is more craft than science, involving personal preference and taste as much as it does a strict set of rules. It is not unlike mathematical notation–its results can be obscure or transparent, expedient or cumbersome, well-aligned or misaligned with the way its users think. Like mathematical notation, software is a tool. Users are enthusiastic about end results, not tools. Therefore, software design rightfully is not a core interest to many people, and it is generally only noticed when it has gone wrong.

Unlike most of the other chapters, this chapter is not scientific content, but instead a narrative compiling a few tricks and design decisions I have found helpful. I provide it in the hope that it may be useful to some–the rest may safely skip forward as nothing in later chapters depends on this information.

Every design starts with a number of goals. In the case of hedge, these were the following:

Split jobs by aptitude. Humans are good at creativity, but terrible at tedious tasks, and computers the other way around. This should serve as a guideline on how tasks should be split. A different way of expressing this is: If a task is so simple and so unambiguous that a computer can carry it out, it probably should do so. Conversely, if a task requires choices or creativity, a computer should not attempt the task.

Be concise. The shortest, least redundant way to specify desired behavior is usually the best one.

No frameworks. Good tools are built from small parts that work well together. The word "framework" is often used when the design has gone awry and the small parts have become so entangled and interdependent that it does not make sense to use them in isolation.

I will not attempt to rigorously derive the design of hedge from these principles, but instead describe the design, and in doing so, refer back to these goals as necessary.

As justified in Section 1.3, hedge is a hybrid code, mostly written in Python [van Rossum et al., 1994], a high-level scripting-type language. It decomposes into a number of modules:

- Operator Template Specification and Processing (module `hedge.optemplate`)
- Geometry handling (module `hedge.mesh`)
- Pre-built templates for e.g., waves, Poisson, gas dynamics (module `hedge.models`)
- Visualization (module `hedge.visualization`)
- Global DG discretizations (module `hedge.discretization`)
- Element-local discretizations (module `hedge.discretization.local`)
- Time Integrators (module `hedge.timestep`)

Useful as they all may be, all but the first of these modules follow established practice in finite element software literature. The only part of how hedge is built that is non-standard and therefore worth describing in detail is the way in which DG operators are described and processed. To fix terminology, the term *operator template* (or just *template*) refers to the collection of operations that yield a discretization of an application problem–e.g. a discretization of Maxwell's equations. This terminology was chosen to avoid confusion with the term 'operator', which is used for the smaller parts from which the template is built, for example an element-local differentiation "operator" or a flux evaluation "operator". This (somewhat obscure) terminology is also used within hedge's code. It has nothing to do with and should not be confused with the C++ term 'template'.

One comment on these modules before I move on: To serve the goals of "*no frameworks*" and "*be concise*", most of these modules only intersect in user code and know nothing about each other, making, e.g. time discretization entirely independent of spatial discretization. (As an example of the consequences that such separation may have: this particular one forces a method-of-lines approach for time-dependent PDEs upon the user.) Some interdependencies are unavoidable, for example visualization must know about the DG discretization, and the discretization must know about the geometry. Nonetheless, these dependencies are kept to a minimum and happen along well-defined interfaces. Remarkably, aside from aspects of quadrature, operator templates are built without knowledge of the DG discretization.

I remark that the design of special-purpose languages for finite element operators templates in itself is far from new [Pironneau et al., 2010, and predecessor projects]. Also the idea of embedding operator templates in another language, as I will do here, has been discussed many times before, e.g. for Scheme by Bagheri and Scott [2004], for C++ by Prud'homme [2006], for Python by Logg and Wells [2010], and for even discontinuous Galerkin methods *and* Python, by Ølgaard et al. [2008].

So what sets hedge apart from these similar projects?

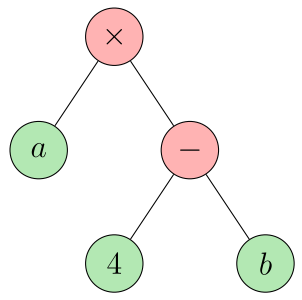
- **Matrix-free.** Unlike most other finite element codes, hedge is not designed to assemble sparse matrices. Instead, it focuses exclusively on the fast application of an operator template's effect to a discretization state vector, reaping a large performance benefit because it can exploit local matrix structure far better than a generic sparse-matrix code.

- **Multiple targets.** One and the same piece of hedge user code can be run on a single processor, using distributed-memory (MPI) parallelism on multiple machines, on one graphics processing unit (GPU) (see Chapter 5), or on multiple GPUs using MPI. The decision to be matrix-free has important consequences in this point, as it enables target-specific optimizations that would not be available if a sparse-matrix approach were pursued.

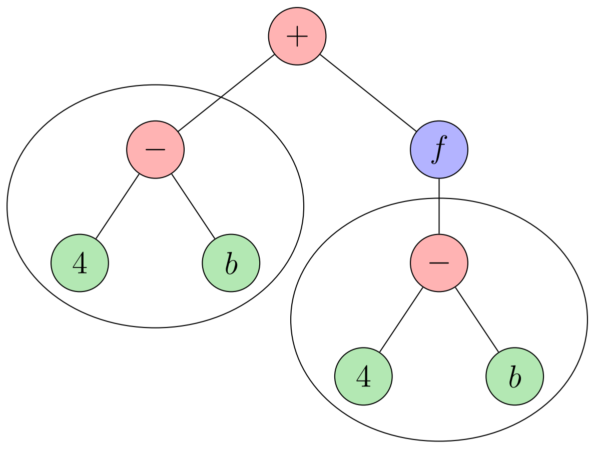
- **Optimization.** Hedge knows enough about the meaning of the operator templates it deals with to be able to rewrite them in such a way that their execution will proceed as quickly as possible. Several examples of this will be provided in Sections 3.3.2 and 3.4.1.

- **Run-time code generation.** Once an operator template has been suitably rewritten, hedge generates C or C++ code that is, transparently to the user, compiled to machine code, loaded into core, and used as part of the running process. Chapter 4 explains the infrastructure that I have created to support this procedure on the GPU.

- **Superscalar virtual machine.** Hedge rewrites each operator template into a partially ordered stream of instructions for a simple "virtual machine" (or *VM* for short). This virtual machine has the primitive operations necessary to apply DG operators, some of which are created on the fly by the run-time code generation technique above. The only *partial* order of the instruction stream allows the VM to decide to exploit



(a) A tree representation of the expression $a \times (4 - b)$.



(b) The problem of common subexpressions: a tree representation of the expression $(4 - b) + f(4 - b)$.

Figure 3.1. Tree representation of expressions.

asynchrony and advance the evaluation of multiple parts of an operator template at once. This mirrors the behavior of modern microprocessors, where a similar property is called 'superscalar' because it involves the processing of more than one scalar at a time. Details of this are explained in Section 3.4.2.

3.2 A Language for Discontinuous Galerkin Methods

The foundation for these features setting hedge apart is a flexible data structure to represent DG operators. This may sound trivial, but it again is a differentiator. Many solver codes involve a one-to-one mapping from operations specified by the user to executed operations. In hedge, the mapping is not one-to-one, but mediated by a number of different representations, the first, user-facing one of which is the subject of this section.

This user-facing operator representation uses expression trees, as illustrated in Figure 3.1(a) on the preceding page. The goal of these expressions is to simply represent DG operators of the form (2.3) in a fairly one-to-one manner, with node types for bilinear forms, arithmetic operators, and more.

A variety of node kinds are admissible in the expression tree, each compatible with certain argument types to which its child subtrees must conform. Before I detail the kinds of nodes, I consider it helpful to enumerate to which *types* are allowed, or equivalently, with which objects hedge can deal. These are the following: scalars, whole-volume state vectors, interior face vectors, and boundary state vectors. Boundary vectors' types are further differentiated by their *boundary tag*, which is a symbolic name for a part of the discretization domain's boundary. Boundary tags may overlap, i.e. one face may have more than one boundary tag, or none at all.

In addition, each of the vector types has a *representation tag*, which may either indicate a simple nodal data format as explained in Section 2.1.1, or a quadrature format, in which the quadrature in use is identified by a *quadrature tag*. The quadrature tag is a symbolic name to which the discretization will assigns a meaning later on. (Quadrature is used to provide extra integration accuracy if one or more arguments of a bilinear form is nonlinear in its argument vectors.)

Hedge is a scalar code, by which I mean that each of the vectors mentioned are vectors only to capture their spatial dependencies. Vector quantities, such as the three-component electric field, are broken up into their scalar constituents and dealt with entirely separately. No operation works on more than one such vector at a time.

The following operations are admissible node kinds in hedge's expression trees:

- **Arithmetic:** binary (division, power) and n-ary (sums and products). Subtraction is not available separately, but represented as $a + (-1) \cdot b$. Numpy's [Oliphant, 2006] scalar-to-vector broadcasting rules are recognized, i.e. one may add or multiply a vector and a scalar and get the expected result.

- **Constants:** Literal scalars such as the (-1) used to represent subtractions.

- **Variables:** Scalar- or vector-valued placeholders whose value is supplied by the user at evaluation time. All user-facing data in hedge is nodal, therefore vector variables are also assumed nodal. Scalar variables must be explicitly declared as such.

- **Common Subexpression Tags:** Used to remove redundant evaluation in the expression tree, see Section 3.2.2.

- **Operator bindings:** A node with two children, representing a unary operation on a vector. The first child represents the operator being applied, the second one the vector to which it is bound. The following types of operators exist:

  - **Element-wise volume operations:** These capture the application of bilinear forms such as the mass matrix

$$(Mu)_i = \sum_{\mathsf{D}_k \subset \Omega} \sum_{j=1}^{N_p} \int_{\mathsf{D}_k} u_{k,j} l_{k,j}(x) \phi_i(x) \, \mathrm{d}x,$$

    its inverse, or modal filtering. In the expression above, $l_{k,j}$ refers to the jth Lagrange interpolation polynomial on element k and ϕ_i to the i global basis function–in this case also a Lagrange polynomial.

  - **Volume local differentiation:** This type of operator captures single-sided, element-local stiffness matrices of the following types:

$$(S_l u)_i = \sum_{\mathsf{D}_k \subset \Omega} \sum_{j=1}^{N_p} \int_{\mathsf{D}_k} u_{k,j} \partial_l l_{k,j}(x) \phi_i(x) \, \mathrm{d}x,$$

$$(S_l^T u)_i = \sum_{\mathsf{D}_k \subset \Omega} \sum_{j=1}^{N_p} \int_{\mathsf{D}_k} u_{k,j} l_{k,j}(x) \partial_l \phi_i(x) \, \mathrm{d}x,$$

for $l \in \{1, \dots, d\}$ with d the number of dimensions of Ω.

  - **Surface bilinear forms:** Used for flux evaluation. See Section 3.2.1.
  - **Conversion:** Various conversion operators are defined that convert between the admissible vector types. Operators exist to extract volume values opposite to a certain boundary (identified by a boundary tag), or for up-sampling to a volume, facial, or boundary quadrature grid.
  - **Further miscellaneous operations:** Finally, a number of auxiliary operators, such as element-wise maxima for the benefit of Rusanov fluxes, or surface data exchanges via MPI, are available. This latter data exchange will be examined in Section 3.3.3, the rest of these operators are secondary to this discussion.

- **Function Calls:** Function calls are a "hook" by which the user can insert arbitrary processing into an operator template. Functions are registered with the virtual machine (see Section 3.4) under a symbolic name and can then be invoked by that name in operator templates.

- **Conditionals:** There is limited support for conditionals, with no means for branching, i.e. both the 'then' and the 'else' part of a conditional are necessarily evaluated, and then results chosen based on whether a conditional is positive.

These operations define a language rich enough to capture most DG operators and largely goes back to abstractions already present in the book by Hesthaven and Warburton [2007].

Section 3.2.3 provides an example of how one proceeds to create an operator template. But before then, I would like to explain two more involved design features of the language: fluxes and common subexpressions.

3.2.1 Fluxes and Flux-Local Binding

This section describes how hedge's language captures the evaluation of bilinear forms of the general shape

$$(Fu)_i = \int_\Gamma F^*(u(x_j))l_{k,j}(x)\phi_i(x)\,\mathrm{d}S_x, \tag{3.1}$$

where

$$\Gamma = \left(\bigcup_{\mathsf{D}_k\subset\Omega} \partial\mathsf{D}_k\right)\setminus\partial\Omega \qquad \text{or} \qquad \Gamma\subseteq\left(\bigcup_{\mathsf{D}_k\subset\Omega} \partial\mathsf{D}_k\right)\cap\partial\Omega,$$

cases referred to as 'interior fluxes' or 'boundary fluxes' respectively In the latter case the subset Γ of the boundary is again specified by a boundary tag.

The function F^* may of course depend on more than one argument, unlike in the example of (3.1). Since the same flux expression F^* is often used in more than one situation (such as for both interior and boundary fluxes), it is defined in a separate expression and can then be used repeatedly, with its arguments, to which it refers by number, rebound to different expressions (such as boundary conditions) each time.

Taking into account quadrature, a number of different computational cases can arise in the evaluation of surface flux expressions like (3.1), in each of which flux evaluation accepts different argument types:

Nodal interior fluxes. Only nodal volume vectors are accepted as arguments.

Nodal boundary fluxes. Nodal volume and boundary vectors are accepted. The boundary vectors' boundary tag must match the boundary tag for which the flux is to be computed.

Quadrature interior fluxes. Only quadrature interior face vectors are accepted as arguments. All arguments must have matching quadrature tags.

Quadrature boundary fluxes. Quadrature interior face vectors and quadrature boundary vectors are accepted. The boundary vectors' boundary tag must match the boundary tag for which the flux is to be computed. All arguments must have matching quadrature tags.

Fluxes are the only operators in hedge that break the 'scalar-ness' rule above, in that they accept any number of vectors as arguments. Boundary fluxes require even more flexibility since they accept interior and boundary arguments and must specify the boundary tag to which they apply. This triple is captured in a "*boundary pair*", which then becomes the argument to the flux operator created from the expression F^*.

It was discussed at length in Section 2.1 that the evaluation of (3.1) actually proceeds in two stages–first, the values of F^* are computed along Γ, and then the integral along Γ is carried out through the lifting matrix. Hedge's language does not capture these two steps separately–they are always lumped together, although all actual target implementations separate them again.

3.2.2 Common Subexpression Elimination

One issue with tree-based expression representations such as the one of Figure 3.1(a) on page 26 is that they fail to properly capture redundancy in the expression tree. If this problem is not dealt with appropriately, identical subexpressions may be evaluated multiple

```
d = dimensions

from hedge.flux import (
        FluxVectorPlaceholder, make_normal)

w = FluxVectorPlaceholder(1+d)
u = w[0]
v = w[1:]
normal = make_normal(d)

from hedge.tools import join_fields
flux = - join_fields (
        dot(v.avg, normal) - 0.5*(u.int-u.ext),

        u.avg * normal
        - 0.5*(normal * dot(normal, v.int-v.ext)))

from hedge.optemplate import (make_vector_field,
        BoundaryPair, get_flux_operator,
        make_stiffness_t, InverseMassOperator,
        BoundarizeOperator, make_normal)

w = make_vector_field("w", d+1)
u = w[0]
v = w[1:]
from hedge.mesh import TAG_ALL

dir_u = BoundarizeOperator(TAG_ALL)(u)
dir_v = BoundarizeOperator(TAG_ALL)(v)
dir_bc = join_fields (-dir_u, dir_v )

# operator assembly
flux_op = get_flux_operator ( flux )

op_template = InverseMassOperator()(
        join_fields (
            -dot(make_stiffness_t(d), v ),
            -(make_stiffness_t(d)*u)
            )
        - (flux_op(w) + flux_op(
            BoundaryPair(w, dir_bc, TAG_ALL))))
```

Figure 3.2. An example of the construction of an operator template in hedge.

times, leading to gross inefficiency, as seen in Figure 3.1(b).

Hedge avoids this by allowing the user to wrap these subexpressions in a special node type labelling them as *common*. This alerts hedge's operator template processing machinery to the redundancy and enables its removal. It could be argued that these common subexpressions should be found automatically–however this constitutes taking options away from the user, who may need to precisely control the granularity at which expression evaluation takes place, for example to maintain storage constraints.

3.2.3 An Example

To augment all the abstract discussion of how hedge's operator language works with a practical example, Figure 3.2 shows a simple example of how one might build an operator template for the second-order wave equation $\partial_t u = \triangle u$ rewritten in first-order form

$$\partial_t u + \nabla_x \cdot v = 0,$$
$$\partial_t v + \nabla_x u = 0$$

with upwind fluxes

$$u^* = \hat{n} \cdot \{v\} - \frac{1}{2}(u^- - u^+), \qquad v^* = \hat{n} \left(\{u\} - \frac{\hat{n}}{2} \cdot (v^- - v^+) \right) \tag{3.2}$$

and homogeneous Dirichlet boundary conditions, where $\hat{n}$ represents a unit face normal and u^-, u^+ and u represent interior, exterior, and average values respectively.

After an index-based flux vector placeholder is created in line 4, it is partitioned into parts for u and v in lines 5–6. Lines 10 through 14 then almost literally transcribe fluxes of (3.2). Further on, in lines 23–25, a named user variable `w` is created and again partitioned into subfields for u and v. Lines 28–30 extract boundary values from the volume vectors `u` and `v` on the entire domain boundary (`TAG_ALL`) and combine them into a Dirichlet boundary condition $(-u, v)$. To conclude, the operator is assembled in lines 35–41.

I would like to make one final observation on the example. It was stated above that hedge is purely scalar internally. That is certainly true and will be visible from the processed form of this example that will be discussed in Section 3.4.2. Nonetheless, the user is at liberty to use vectorial abstractions in assembling the operator template. Hedge will immediately break the operator down into its scalar components–but this fact is transparent to the user. As an example of this, the template of Figure 3.2 makes use of this to produce a working operator for the wave equation in *any* dimension.

3.2.4 Discussion

Hedge's language is expressive enough to capture most discontinuous Galerkin schemes describable within the confines of Section 2.1. It does so in a way that is concise and closely matches the mathematical notation one might use in a scientific publication. It therefore successfully achieves the goals set forth above.

Nonetheless, some of the decisions influencing its design could rightfully be considered questionable by now and are mainly historically based:

- **Operator bindings** were introduced to support two alternative syntaxes of specifying operator application, the binding-type `op(arg)` and the multiplication type `op*arg`. The latter enables, for instance, the intuitive syntaxes `dot(nabla, vec)` and `cross(nabla, vec)` for divergences and curls, but this necessitated that operators be separate objects that can be multiplied *and* bound. One of hedge's first processing steps is to convert all multiplied operators into bound operators, and the reasoning for allowing multiplied operators seems questionable to me by now.
- **Boundary pairs** are largely an artifact of the operator binding issue discussed above– a workaround for the fact that operators can only support a single argument.
- **Positional flux argument binding** and the entire secondary level of evaluation for fluxes could have been avoided without much loss in expressiveness, as the creation of fluxes with identical flux expressions could have been handled through Python functions at the user level, whose rebinding behavior is more immediately transparent to the user and would have removed one small detail that a user needs to learn in order to use hedge.

Fortunately, none of these warts cast doubt on the future maintainability of hedge's language, and some of them might even be removable in a backwards-compatible fashion.

3.3 The Processing Pipeline

Once an operator template is specified by the user, hedge's processing machinery goes to work and, through a number of steps, transforms the template into an intermediate representation that can be executed by the VM. These processing steps are described in this section.

3.3.1 Type Inference and Operator Specialization

Throughout the description of hedge's language in the previous section, it has already become clear that the language is strongly typed, and using data of the wrong type can easily have disastrous consequences (consider adding two boundary vectors for separate boundaries that happen to be of the same length). To protect the user from such mistakes, and to free the user from having to differentiate all the operators that can arise (e.g. as described in Section 3.2.1), hedge performs type inference on the expression trees it is given by the user.

In this type inference step, each subexpression's type is deduced by propagating type information up and down the expression tree and, at each node, unifying this information. As an example of unification, consider an argument of boundary flux that is the sum of two user variables. From 'above', i.e. from the boundary flux expression, the sum can infer that it must be a boundary vector for a known boundary tag, and from below it can infer that it must be using a nodal representation (by being made of user variables, which are nodal).

This propagation step is repeated until the type information “converges”, i.e. a further propagation step does not derive any additional information. By the monotonicity of unification, the process must converge or end in a unification error.

Once type inference is complete, complete type information is available and operator specialization takes place, integrating argument type information into the operators being applied (such as which quadrature tag is used, etc.). This information is then used in code generation.

3.3.2 Optimizations

It is often convenient for the user to generate slightly suboptimal representations of an operator template. Hedge incorporates a number of processing steps whose goal it is to turn such templates into ones that can be evaluated as efficiently as possible.

These are the processing steps that hedge performs for optimization:

- **Constant Folding** short-circuits the evaluation of expressions whose value is known at processing time, such as any expression multiplied by zero, or a number of multiplications by constant scalars which can be collapsed into one.
- **Boundary-Condition-to-Flux-Rewriting.** Many boundary conditions (such as especially homogeneous Dirichlet and Neumann types) are expressed in terms of the interior values of a certain quantity. There are two ways in which these boundary conditions can be expressed in hedge: First, by using an extraction operator, computing the correct boundary vector of exterior values, and then feeding this vector back into a boundary flux. Second, by substituting the boundary condition directly into the flux expression, which then only needs to refer to the interior fields.

Both ways have advantages: The first way is very straightforward and lets the computer code reflect boundary conditions as they are written on paper, as evidenced in Section 3.2.3. The second way is cumbersome for the user, but much more efficient. In keeping with hedge's goals, this processing step rewrites the first form into the second, achieving both efficiency and convenience for the user.

- **Derivative Joining.** Hedge provides infrastructure to generate certain types of operator templates in a mechanized way, such as for second-order operators of type $\nabla \cdot F(\nabla u)$ [Arnold et al., 2002] of IP, LDG and a few more flavors. When combining these with other parts of an operator template, the user may obtain an expression of the type $\partial_x(A) + \partial_x(B)$. Derivative joining performs the obvious rewrite to $\partial_x(A + B)$.

- **Inverse Mass Contraction.** For time-dependent simplicial DG, one obtains schemes of the form (2.2). As already seen in (2.3), it is computationally efficient to combine the inverse mass matrix arising from the inner product around the time derivative with other element-local matrices if the discretization allows it. (Curvilinear elements for example might prevent this.) Inverse Mass Contraction automatically performs this rewriting step, taking into account the linearity of the operation to find further simplifications.

3.3.3 Target-Specific Rewriting

The ability to reason about and rewrite the users' operator template is not only very useful in providing convenience while achieving speed, it turns out to be crucial for an entirely different task as well: The support of different computational targets. In a distributed-memory domain-decomposing parallelization of a DG operator using MPI, interior fluxes are broken up into parts on each subdomain–the interior faces of the subdomain, and new boundaries that have emerged along the element faces at which the domain was split. Hedge's parallel computing support makes use of the operator template rewriting capabilities to explicitly add these boundary fluxes, along with further expressions that facilitate MPI-based exchange of boundary data. In the compilation step detailed in Section 3.4.1, these flux exchange expressions are split up into separate send and receive instructions.

Support for GPUs (Chapter 5) also employs rewriting in a somewhat more complicated manner to facilitate efficient GPU-DG flux computations.

3.4 The Virtual Machine

While a tree-based operator template representation as introduced above is convenient for rewriting and simplification, such a representation can become cumbersome when it comes to the efficient evaluation of the expression, i.e. the execution of the processing steps indicated by the expression tree. The optimizations applied in the compilation from tree-shape to instruction stream (that I will describe below) will make the superiority of the approach over direct recursive-descent evaluation immediately obvious.

3.4.1 The Compilation Step

Like most optimization and rewriting steps above, compilation proceeds through a number of depth-first traversals of the expression tree.

The first such traversal collects all differentiation-like operator applications occurring in the tree. Many of these apply to the same subexpression, but may differentiate along different (e.g. x and y) axes. It is important to realize at this stage that DG differentiation conceptually proceeds in two stages–first, by differentiating the expansion along the element-local unit directions, and then by transforming the local derivatives to ones aligned with the global x and y axes. Hence the local derivatives are a shared "raw material" that should be reused. Unfortunately, a tree-traversal-based evaluation would be required to store *all* local derivative based on the possibility that they *might* be reused somewhere else in the tree. This can result in prohibitive amounts of storage use. In hedge's compilation step, this information is gather ahead of time, and all required derivatives of a given subexpression (and only the required ones!) are computed together, so that the local derivatives may be safely discarded thereafter.

A similar "economy of scale" arises in the evaluation of surface flux terms. Since each flux evaluation needs to read certain information, e.g. index data indicating where facial nodes are located within a volume vector, or what the surface normals of a given face are, it again makes sense to amortize these fetch costs over as many flux expressions as possible. The second pass of hedge's compilation step uses another tree traversal to find all occurring flux operators, groups them by their integration domains Γ and, through an analysis of their data dependencies, determines which fluxes may be batched together. This "flux batching" is one of the optimizations that allow hedge to operate in a purely scalar fashion while still being efficient.

Yet another type of expression occurring in hedge's language can benefit from batching–vector arithmetic. Imagine the two operations $d \leftarrow a + b$ and $e \leftarrow a + c$. If both of these expressions were evaluated separately, then a would have to be read twice, an unnecessary expense in memory bandwidth. Therefore, again based on a data dependency analysis that respects the common subexpression information of Section 3.2.2, hedge is able to group together vector arithmetic on vectors of matching size, saving large amounts of memory bandwidth.

3.4.2 The Execution Model

The end result of compilation is a stream of instructions, each of which closely represents one of the operations given in Section 3.2, while taking into account the batching of the previous section. Each instruction contains a list of its data dependencies and its output variables. Output variables are named either artificially, or, for ease of debugging, by names specified along with common subexpressions. The generated code is free of side effects, i.e. variables are only assigned once. To reduce the amount of storage occupied, the virtual machine detects which variables are still 'live' (i.e. still required in a future computation) and discards the rest.

As indicated above, the best way to view this dependency-annotated instruction stream is as a partially-ordered plan for the evaluation of a compiled operator template. It would of course be trivial to remove the ambiguity and turn the partial order into a total one, at which point one could also conveniently discard dependency information. I would argue that this is unwise, because some of the instructions supported by the virtual machine may require an unspecified and varying time to complete (such as MPI transfers across a physical network or GPU-host transfers across a host bus) and no single static schedule may work in all cases.

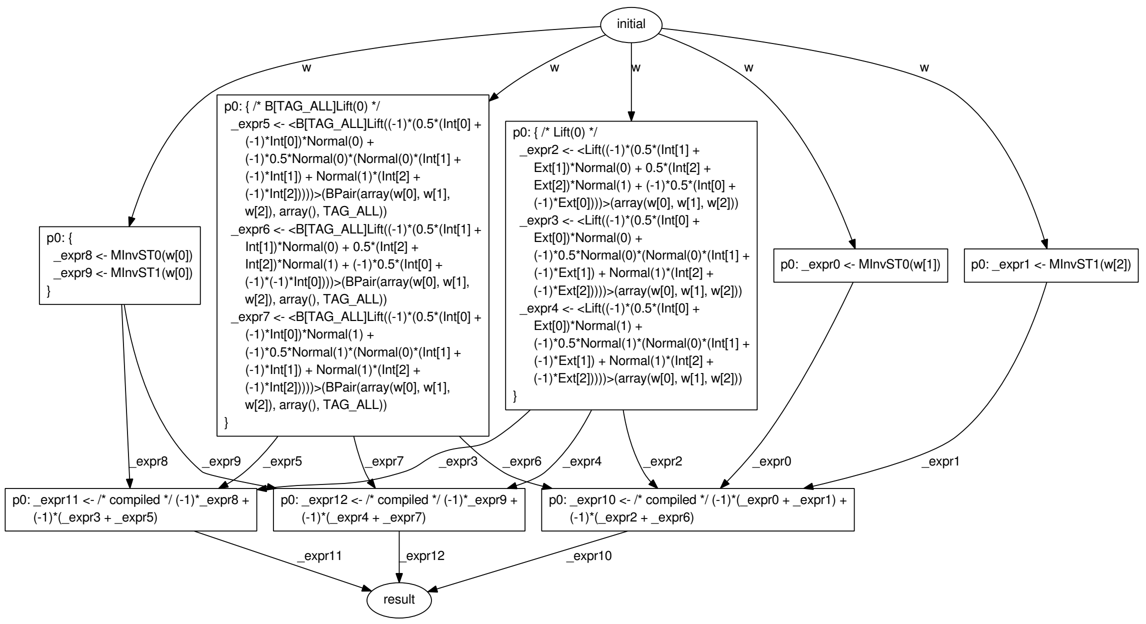
In addition to taking an indeterminate amount of time, many of these instructions are also asynchronous, i.e. a command is issued to perform them, and the main processor is free to pursue other tasks while the operation completes. Such operations are captured in hedge's execution model by *futures* [e.g. Friedman and Wise, 1976] (often also called "*promises*")–handles to data which are not yet available but will be at some future time. Hedge's futures have two capabilities: They can be queried whether their underlying operation has finished, and they can be asked to yield their promised datum, waiting if necessary.

Based on the future's query facility, hedge's virtual machine makes sure that, as long as instructions exist whose data dependencies are available, processing continues–an important property that a static schedule cannot ensure. Of course, static scheduling can be more efficient when asynchrony is not a concern. Therefore hedge still makes five attempts to create a totally ordered static schedule based on observation of execution order of the dynamic schedule. Each static schedule is reused until it produces a "stall", i.e. a situation where further instructions are available, but cannot be evaluated because their data dependencies are not ready yet. After five non-stall-free static schedules, hedge gives up and resorts to dynamic scheduling full-time.

To give the reader an impression of the instruction streams that hedge creates from operator templates, I have included data flow graphs that were automatically generated from these streams. Figure 3.3 on the following page shows the (simple) data flow graph resulting from the wave equation example of Section 3.2.3, the largest part of which is taken up by the surface flux calculations, which are represented by the big boxes in the middle of the figure. The boundary flux is given in the slightly bigger box on the left, while the interior flux is shown in the smaller box to the right. From the resulting graph, one can see how the various optimizations have been applied by hedge–in particular, the effects of derivative batching, flux batching, and inverse mass contraction can be observed.

It is instructive to notice how the data flow graph (or, equivalently, hedge's dependency graph information, from which it is derived) neatly exposes the inherent opportunities for parallel execution of different parts of the operator–a capability of which I plan to make more use in the future.

The example of Figure 3.4 on page 43 shows a processed operator template for the compressible Navier-Stokes equation with quadrature. My goal in showing this operator is not so much to shed light on how it is processed by hedge (as it is not exactly legible at the



```
p0: {
  _expr8 <- MInvST0(w[0])
  _expr9 <- MInvST1(w[0])
}
```

```
p0: { /* B[TAG_ALL]Lift(0) */
  _expr5 <- <B[TAG_ALL]Lift((-1)*(0.5*(Int[0] +
    (-1)*Int[0])*Normal(0) +
    (-1)*0.5*Normal(0)*(Normal(0)*(Int[1] +
    (-1)*Int[1]) + Normal(1)*(Int[2] +
    (-1)*Int[2]))))>(BPair(array(w[0], w[1],
    w[2]), array(), TAG_ALL))
  _expr6 <- <B[TAG_ALL]Lift((-1)*(0.5*(Int[1] +
    Int[1])*Normal(0) + 0.5*(Int[2] +
    Int[2])*Normal(1) + (-1)*0.5*(Int[0] +
    (-1)*(-1)*Int[0])))>(BPair(array(w[0], w[1],
    w[2]), array(), TAG_ALL))
  _expr7 <- <B[TAG_ALL]Lift((-1)*(0.5*(Int[0] +
    (-1)*Int[0])*Normal(1) +
    (-1)*0.5*Normal(1)*(Normal(0)*(Int[1] +
    (-1)*Int[1]) + Normal(1)*(Int[2] +
    (-1)*Int[2]))))>(BPair(array(w[0], w[1],
    w[2]), array(), TAG_ALL))
}
```

```
p0: { /* Lift(0) */
  _expr2 <- <Lift((-1)*(0.5*(Int[1] +
    Ext[1])*Normal(0) + 0.5*(Int[2] +
    Ext[2])*Normal(1) + (-1)*0.5*(Int[0] +
    (-1)*Ext[0])))>(array(w[0], w[1], w[2]))
  _expr3 <- <Lift((-1)*(0.5*(Int[0] +
    Ext[0])*Normal(0) +
    (-1)*0.5*Normal(0)*(Normal(0)*(Int[1] +
    (-1)*Ext[1]) + Normal(1)*(Int[2] +
    (-1)*Ext[2]))))>(array(w[0], w[1], w[2]))
  _expr4 <- <Lift((-1)*(0.5*(Int[0] +
    Ext[0])*Normal(1) +
    (-1)*0.5*Normal(1)*(Normal(0)*(Int[1] +
    (-1)*Ext[1]) + Normal(1)*(Int[2] +
    (-1)*Ext[2]))))>(array(w[0], w[1], w[2]))
}
```

```
p0: _expr0 <- MInvST0(w[1])
```

```
p0: _expr1 <- MInvST1(w[2])
```

```
p0: _expr11 <- /* compiled */ (-1)*_expr8 +
    (-1)*(_expr3 + _expr5)
```

```
p0: _expr12 <- /* compiled */ (-1)*_expr9 +
    (-1)*(_expr4 + _expr7)
```

```
p0: _expr10 <- /* compiled */ (-1)*(_expr0 + _expr1) +
    (-1)*(_expr2 + _expr6)
```

Figure 3.3. Data flow graph for the second-order wave operator from the example of Section 3.2.3.

size at which it is shown), but rather to clarify that the techniques described in this chapter apply equally well to large, more complicated operators.

3.5 Conclusions

In this chapter, I have described a set of methods by which I have implemented a hybrid discontinuous Galerkin solver that underlies much of the further work in this thesis. I have described in how far the design goals set forth at the beginning have influenced the design of the solver, and I have shown tricks which enabled me to achieve the design goals while maintaining full efficiency even when the two seemed critically at odds with each other. In addition to allowing large-scale, parallel, and GPU computations, it is efficient and competitive with (often much more verbose and less flexible) codes written in compiled languages.

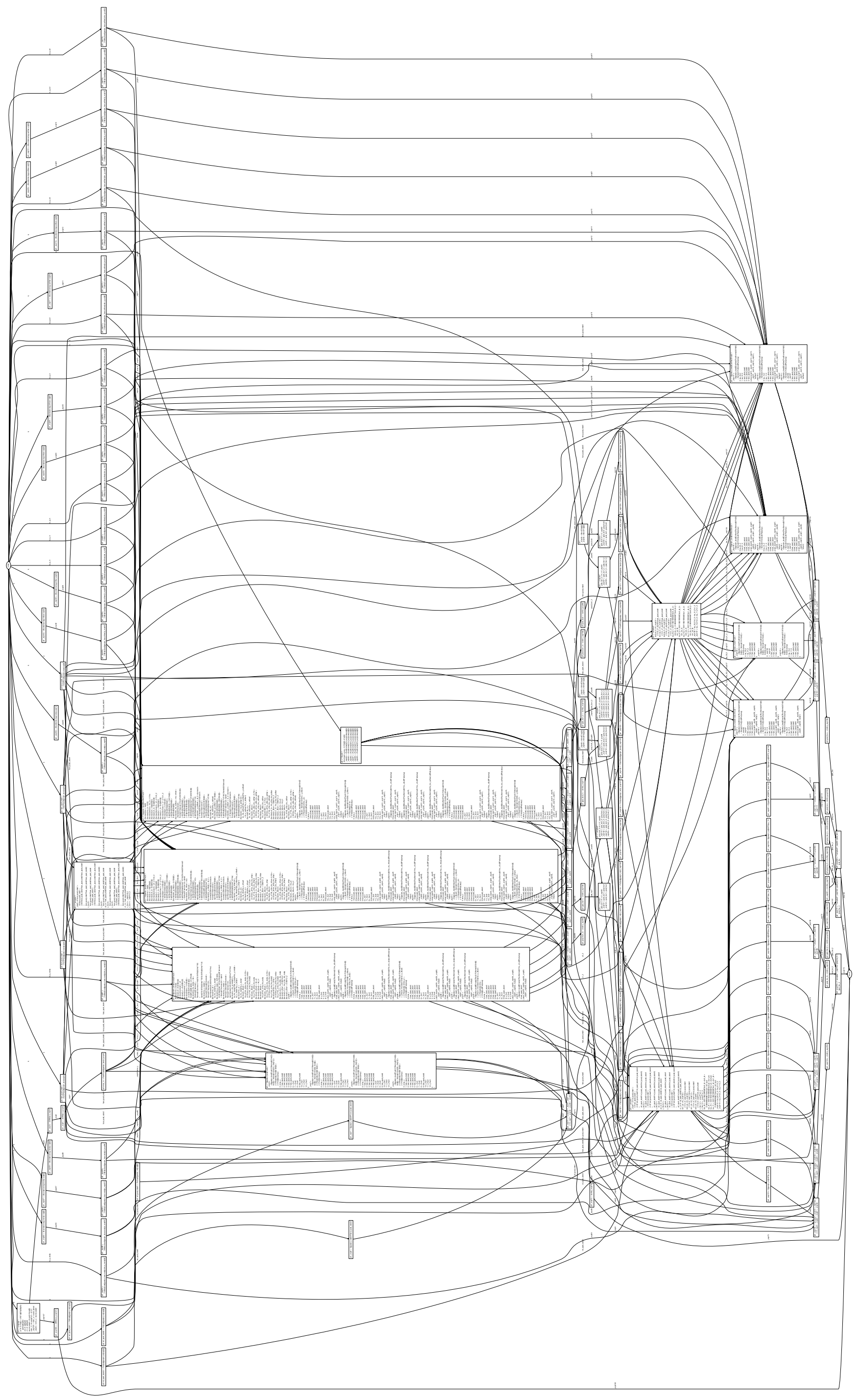


Figure 3.4. Data flow graph for the compressible Navier-Stokes operator with quadrature, discretized using Lax-Friedrichs fluxes for the first-order part and stabilized central fluxes for the second-order part [Arnold et al., 2002].

Of course, work on a big project such as this is never ‘finished’. Nonetheless, hedge has reached a stage where it is useful and, by virtue of its open-source licensing, has found a number of users outside Brown University. A few things could still be done to make hedge even more efficient. For example, DG-specific operators such as element-wise differentiation could be mixed with vector arithmetic to avoid a few more store-fetch cycles. I would expect that this type of tail computation optimization might, depending on approximation order, be able to achieve another 20 per cent of speed gain on both CPUs and GPUs.

As a conclusion, I can say that in addition to being an interesting exercise in software design, hedge has been a productive vehicle for my research, as I hope to demonstrate in subsequent chapters, and I anticipate that this will continue to be the case in the future.

CHAPTER FOUR

Code Generation on Graphics Processors

4.1 Introduction

Graphics Processing Units (GPUs) [Dally et al., 2003, Lindholm et al., 2008, Seiler et al., 2008] promise tremendous advantages in throughput over conventional processor architectures, ideally resulting in a large reduction of execution time for suitable compute- or bandwidth-bound algorithms. However, execution time is not the only time scale to consider when comparing computer architectures. Indeed, the development time for a scientific code will, in many cases, be a significant fraction of its useful lifespan. GPUs now threaten to tip this balance even further out of the programmer's favor, through the following four factors.

First, there is still much change going on in the area of massively parallel processors. These changes are driven by many factors–chip manufacturing processes change, new ideas and abstractions in hardware and software emerge and disappear at a rapid pace, market conditions change. Programs that work well on last year's machines may not continue to represent optimal choices today. While the recent ratification of the OpenCL standard [The Khronos OpenCL Working Group, 2008] may bring a moment of stability, the landscape of devices that may be accessed is still large and ever-changing. Even though some patterns are emerging, the world is still very far from having settled on a programming model for massively parallel machines–a model that is as stable as the one programmers have enjoyed on CPUs for the last few decades.

Second, GPU code is very sensitive to seemingly innocent changes. Hardware implementation details are much more visible and have a much greater performance effect in GPU programs than they do in today's CPU programs. Relative component clock rates, bus widths, vector widths, memory and buffer sizes all have an immediate impact on a successful code. The *very premise* of GPU computing is to try and find a better use for the silicon tied up in the caching, speculation and out-of-order execution that frees a modern CPU developer from having to worry about hardware peculiarities. I therefore expect that GPU developers will continue to be exposed to these details.

Third, and potentially a corollary of the last point, GPUs offer many more implementation choices, and often little guidance on which choice may lead to efficient code. It is not uncommon to see differences of an order of magnitude in execution time between codes that accomplish the same basic task. This is not likely to occur on a current-generation CPU, where, with few exceptions, “reasonably coded” and “highly optimized” fall within at most a factor of two or three of each other.

The fourth and possibly worst factor is that GPU development tools are in their infancy. Many years have been spent creating development tools that help the CPU developer achieve high productivity. These tools range from high-level languages and libraries that allow the programmer to deal in convenient abstractions, to optimizing compilers, debuggers, and profilers, which likewise shield the programmer from having to deal with the full complexity of the hardware. Many of these tools are either unavailable, inadequate or rudimentary on today’s parallel architectures.

I propose that *GPU run-time code generation* (“RTCG”) helps the programmer reclaim a significant share of the productivity lost to these factors. By GPU RTCG, I mean the ability to seamlessly execute arbitrary, generated low-level C (or C-like) source code for high-volume computational tasks in the context of the generating program. In the form described in this chapter, the generation and execution of the low-level code is performed from a high-level scripting language. By the term “scripting language” or “high-level language”, I mean a language that

- enables various programming paradigms (e.g. functional, procedural, object, aspect, etc.),

- is dynamically typed,
- includes error reporting facilities,
- manages resources automatically,
- offers comprehensive built-in functionality,
- requires no user-visible compilation (i.e. suitable for interactive use), and
- works well as a "glue language" for lower level building blocks.

The family of major general-purpose scripting languages at the time of this writing includes Python, Ruby, Lua, and JavaScript and numerous others.

The present work describes lessons learned from many earlier approaches. GPU RTCG is a form of "metaprogramming": instead of directing computer code immediately at a problem, one directs code at the creation of and reasoning about another piece of code which then solves the problem at hand. It is not initially clear that this additional level actually results in any tangible gain, but I will defer this discussion to the later parts of this chapter. For now, it should suffice to say that I am by no means the first to apply the basic principle. Today, perhaps the most common mechanism used to implement metaprogramming ideas is the template mechanism of the C++ programming language. Many things have been implemented in this effective (if cumbersome) way: Expression evaluators [Veldhuizen and Jernigan, 1997], parser generators [de Guzman, 2008], even entire PDE solver frameworks [Prud'homme, 2006, Reynders et al., 1996]. The template-based technique is however constrained to being applied at the time when the software is built, which limits its usefulness. A variety of ways have been devised to circumvent this restriction, reaching from assembly of small prefabricated pieces into a full code [Frigo and

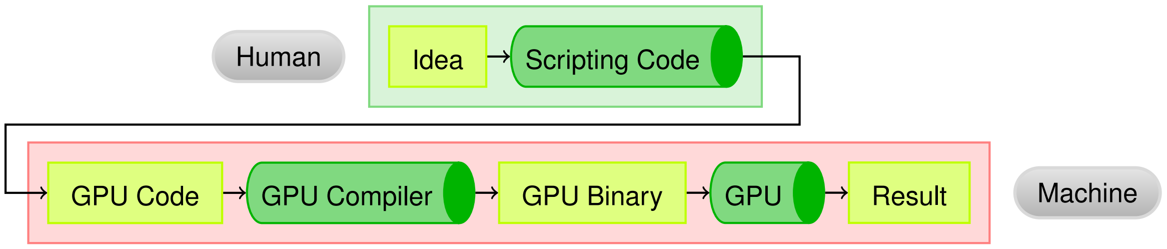


Figure 4.1. Operating principle of GPU code generation.

Johnson, 2005], to build-time evaluation of different code versions [Whaley et al., 2001]. It should further not be forgotten that the Lisp programming language already brought the fundamental insight of the von Neumann architecture, namely that 'code is data', to higher-level languages in the early 1960s [McCarthy, 1962], albeit not necessarily with computational efficiency as the primary target.

In the context of GPUs, metaprogramming has so far been applied mainly in a graphics and image processing context [Lejdfors and Ohlsson, 2006, Wedekind et al., 2008] and to ease the use of a standard rendering pipeline for general-purpose uses [Tarditi et al., 2006]. Other projects focus on generating GPU code using a compile-time C++-based framework [McCool and Du Toit, 2004, McCool and RapidMind Inc., 2006].

Further, this work can be seen in the context of recent [Lengauer et al., 2004] and not so recent [Keppel et al., 1991] efforts to promote program generation as a mainstream idea. In comparison however, I am choosing a decidedly simple approach that values pragmatism over theoretical appeal: Why should one invent new tools from scratch when good results are achievable using a scripting language with a GPU and a C compiler? Curiously, many previous authors give up the immeasurable advantage of being able to generate code at run time all too easily. This capability is the main point of this chapter.

The text is organized as follows: I begin by giving a very brief overview of how GPUs differ from other computing platforms from the point of view of software in Section 4.2.

I continue in Section 4.3 by providing a sampling of problems arising from a GPU's special structure where GPU RTCG can be profitably applied. Section 4.4 then describes a scripting-based approach to these problems that is supported by my open-source *PyCUDA* toolkit. Section 4.5 describes how a number of applications from varied disciplines have benefited from the approach in general and PyCUDA in particular. Finally, in Section 4.6, I close with a few remarks and ideas for future work.

4.2 GPU Software Creation

In the preceding sections, I have already argued that software for GPUs is far more subject to influences beyond its own control than is likely to be the case for CPU software. Such external influences may include, in no particular order,

- the width and number of available compute units,
- the amount of available on-chip buffer memory,
- the speed of various access patterns to on- and off-chip memory,
- the ratio of available memory bandwidth to compute bandwidth,
- the latency and bandwidth between the host (CPU) and the device (GPU), and
- the instruction scheduling details of the processor in use.

Section 2.2 explained that GPUs are aimed at computations of a 'streaming' nature. It is therefore appropriate to visualize a computation running on a GPU as a network of "streams" with varying throughputs, connected to buffer spaces and processing elements that turn inputs into results in certain batch sizes. The goal of designing GPU algorithms is to first map the desired computation (e.g. matrix multiplication) onto such a network of streams, and, simultaneously, to find a mapping from these streams, buffers, and processing elements to the physically available hardware. From this picture, it becomes apparent that every nontrivial piece of GPU software represents a complicated trade-off. In many cases, the programmer making these trade-offs has incomplete information on the factors involved. For example, design details of the compute device may be unavailable to the programmer. But even if they are, program execution in massively parallel processors is a complicated and non-local process that may defy easy comprehension even by the processor's designers.

GPU programming therefore relies extensively on experimentation and micro-benchmarking to overcome missing knowledge of causes by obtaining measurements of symptoms. As a software developer, this is a very unsatisfying place to be in: the obtained results may not be robust to changes of hardware, problem sizes or other parameters. Further, this experimentation and benchmarking is generally tedious work that needs to be carried out systematically, consistently and repeatably. It is therefore not far-fetched to wish for these tasks to be automated. From there, it is a small step to *metaprogramming*, the automated reasoning about programs, and RTCG.

4.3 Problems Solved by GPU Run-Time Code Generation

This section is devoted to describing a number of issues that are commonly faced when programming a GPU. In each case, I point out how a GPU RTCG strategy can be used to address these issues in a natural and straightforward manner.

4.3.1 Automated Tuning

During the creation of a GPU program, it is natural for the programmer to come up with a number of variants of a given code, each of which will be observed to have certain properties regarding data layout and computation speed. The conventional approach to code tuning then calls for the fastest variant to survive, while the others will be discarded. This is not necessarily a desirable course of action, as information is lost. Instead, it seems more appropriate to retain as many of these variants as is practical, assuming that they hold at least some promise. Further, each variant may have a number of tunable parameters, such as loop lengths, block sizes, etc. Retaining variant information permits choosing the best one from a reasonable-size pool of candidates in an automated fashion, guided by some metric such as execution speed. This is the basic premise of automated tuning, which is trivially enabled by GPU RTCG. Further, automated tuning is not just enabled by RTCG, it is enabled *at the right time*–namely at run time–when complete information is available. I present three examples illustrating the type of choices optimally resolved by automatic tuning:

The first and perhaps the most important choice in GPU algorithm design is that of loop slicing, as explained in Section 2.2. Even loops that are trivially linear on the CPU must typically be subdivided into several levels for the GPU to be efficient, with levels corresponding to SIMD lanes, execution units, as well as serial execution. For some algorithms such as matrix multiplication, loop slicing is important even on the CPU to preserve locality of access and thereby the efficiency of on-chip caches. Since GPUs have even less cache and even more slicing levels, getting the loop slicing right is of paramount importance to obtaining reasonable performance.

Second, many GPU architectures have *user-managed on-chip memories*. Upon creation of a code, it is often not obvious which pieces of data will yield the most benefit from low latency local storage. It is almost certain that on-chip memory will remain a scarce resource for the foreseeable future. Thus, peak performance necessitates trade-offs that adapt to the hardware situation at hand.

Third, GPU architectures achieve high memory throughput not through high memory clock rates, but rather through wide data buses. Unfortunately, wide data buses only achieve acceptable net bandwidths when used to transfer large numbers of consecutive data words. Further, the bus widths are often closely matched with the widths of SIMD units in a GPU. It is to be expected that both the loop slicing of the algorithm and the layout of the data it uses will be influenced by these performance characteristics of memory access. Many strategies have been invented to deal with these restrictions, and almost all of them come with drawbacks limiting their usefulness–e.g. wasted space and SIMD lanes in the case of padding. As in the case of user-managed on-chip memory, it is desirable, but nontrivial, to choose a layout that balances advantages and disadvantages.

4.3.2 The Cost of Flexibility

Flexibility is commonly seen as a desirable feature of a computer code–where "code" usually means a user-facing executable. The more functions a certain executable can perform without having to be modified, the better. Yet there exists a flexibility versus performance trade off. As an example that is the polar opposite of flexibility, one may consider an optimized code that can only multiply matrices of a certain size. No matter how fast or otherwise attractive such a code may be, unless the user's desired application requires matrix multiplications of this size, it is entirely useless. Thus almost all computer codes are built with at least some flexibility.

It should then be realized that flexibility comes at a cost: Constants get replaced by variables, formerly fixed loop trip counts become variable, and quite generally a compiler has less knowledge available at compile time, making its optimizer less effective. The process of removing such flexibility, on the other hand, is generally frowned upon and derisively called “hardcoding”. I feel, however, that this point of view has no merit once run-time code generation is available, as one is at liberty to generate code for exactly one purpose–any extra flexibility is likely just unneeded ballast.

In *compile-time* metaprogramming frameworks, hardcoding is sometimes replaced by generating a large number of potentially needed code variants ahead of time by considering anticipated needs for different problem sizes, data types, etc. Once the number of variants surpasses “a few”, the costs of this approach quickly become very significant both in compilation time and memory footprint of the executable. In comparison, GPU RTCG suffers no such scaling penalty: It can use information available only at run time to cut down the number of variants that need to be generated, it can use caching to amortize the cost of finding the optimal code, and unused code variants can be disposed of immediately.

4.3.3 High-Performance Abstractions

Nearly all computer programs are built in ‘layers’, where each individual layer solves a certain subproblem and presents a more abstract, ‘higher-level’ interface to surrounding layers. This is good engineering practice, as it allows partitioning a big problem into many smaller ones, and it enables reuse of engineering effort. In some cases, this layering is easily achieved and results in very little loss for the ‘consumer’ of the interface. In other cases, such abstractions can be made uneconomical by coding circumstance. I will first look at examples of how this might happen, and then at what RTCG does to improve the situation. One common instance of uneconomical abstractions occurs when a consumer of an interface needs to specify details about an operation that is to be performed on large volumes of data, as part of an inner loop in the abstraction. As a trivial example, consider an abstract form of vector addition allowing a variety of scalar types.

An easy (but unsuitable) run-time technique is the use of function pointers (or equivalently, virtual methods). In the frame of the example, each scalar addition under this scheme would require a computed call to a subroutine carrying out the addition on the scalar level. While this allows the required level of run-time polymorphism, it is very expensive: A floating point addition can usually be carried out in a single machine clock cycle, but a computed jump may defeat prediction logic, stall the execution pipeline, and can easily take several orders of magnitude longer than the operation it is meant to perform. Furthermore, the requisite computed calls are unavailable on many types of GPUs.

The disadvantages of the function pointer approach drove the development of mechanisms for compile-time-polymorphism on the CPU and the GPU. In C++, this is achieved through the use of class and function `templates`. If the user's customization is assumed to be known at compile time, the compiler can make use of that knowledge and generate efficient code. In the example, the vector addition would be written with respect to an unspecified type, relying (for example) on the assumption that the underlying scalar supplies addition. The type of the scalar is required to be known at compile time, and hence the compiler can statically find the addition routine and substitute ("in-line") its use, ideally eliminating all overhead. This is a popular approach, but it has two shortcomings: First, it requires early concretization. In the example, all desired uses of the vector addition code have to be known before the program is run. Second, the C++ `template` mechanism in particular responds unfavorably to complexity growth. It makes simple things like type substitution quite easy. But `templates` alone, even without the rest of C++, form a fully capable–if awkward–programming language [Veldhuizen, 2003], and some implementers have seen this as an invitation to do rather advanced things with them. While such use validates the need for a meta-level where code is able to reason about other code, the actual end results in this case tend to be both brittle and complicated.

The ideal solution would be a compromise of these two. Function pointers are simple, flexible and do not require early concretization, while templates have very little overhead. By removing the distinction between ‘compile time’ and ‘run time’, RTCG fills this void. Once RTCG is available, appropriate code can be generated whenever a different requirement arises, leading to flexibility. RTCG code is also fast–it can do away with any sort of flexibility, because it can safely be considered “single-purpose”. Further, code generation can be seen as a text processing task. Since one is not limited in the choice of tools with which to perform this generation, RTCG-based codes can be as simple as possible and respond favorably to complexity growth.

4.3.4 GPUs and the Need for Flexibility

As a final comment, it should be emphasized that in the past, due to the associated development complexity especially for C++-based techniques, metaprogramming was restricted to high-need applications. The cost of metaprogramming outweighed the disadvantages of “hardcoding” only for the largest of projects.

GPUs however democratize this need, as they put a larger penalty on inflexible, untuned code. By deciding to perform a GPU port of an algorithm, one implicitly states that one is willing to trade some implementation effort for a substantial performance gain. As explained above, finding a good implementation is often nontrivial, and therefore the potential gain from RTCG is large. In other words, GPUs increase the relative cost of not using metaprogramming techniques, and therefore it is likely that code generation and techniques like it will see much wider adoption. However, good tools are required to allow the broadest possible cross-section of developers to take advantage of RTCG.

4.4 PyCUDA: A Scripting-Based Approach to GPU RTCG

The previous section has shown that GPU RTCG solves a number of pressing problems in the development of high-performance compute-oriented codes. In this section, I present PyCUDA, a practical and mature open-source toolkit supporting GPU RTCG.

While its name already suggests that PyCUDA connects the high-level Python programming language [van Rossum et al., 1994] with the Nvidia CUDA compute abstraction [Nvidia Corporation, 2009], at least the first choice deserves justification. The major factor in choosing a high-level, dynamic programming language over a potentially better-performing, low-level, static one is the complementarity of tasks between the GPU and the host processor. The GPU is optimally suited to carrying out throughput-oriented parts of a program, namely the part that would have conventionally constituted the 'inner loops'. Freed from this duty, the CPU now is responsible for "only" control and communication (including, e.g., disk input/output). In other words, it now works at a higher level of abstraction. Therefore a high-level scripting language (such as Python) can perform this higher-level job equally well or better, simply because the performance demands are reduced, and both code generation and execution control can be of considerable complexity. Control input is needed by the GPU about once every millisecond, and code generation is needed even less frequently. A Python-based GPU compute code will have no trouble realizing the same full performance potential of GPU hardware as a C-controlled GPU compute code, but with much less effort on the part of the programmer. This reduction in effort is achieved in many ways–for example, data types and resources are managed by the language itself instead of by a human, also closures and other high-level constructs are available. Relatedly I would like to emphasize that PyCUDA does not inhabit Python's software ecosystem by itself: a large number of packages for such diverse purposes as plotting, computer algebra, or optimization are available easily and under liberal licenses [Langtangen, 2009]. Significantly, the `mpi4py` package [Dalcín et al., 2005] in conjunction with PyCUDA allows a straightforward combination of shared-memory GPU-based and distributed-memory MPI-based parallelism. The easy availability of a multitude of packages contributes to making scripting languages more productive than their conventional compiled counterparts. Scripting languages such as Python or even MATLAB are already popular for exploratory prototyping, but in combination with a GPU, their usefulness extends well into the territory of 'full-scale' codes.

PyCUDA itself is built from multiple levels. At the lowest level, PyCUDA makes the *entirety* of the CUDA run-time system available from Python by introducing a thin object-oriented shell. In this context, I would like to emphasize the word "entirety": Every feature of the CUDA run-time system is accessible from Python via PyCUDA, including textures, pinned host memory, OpenGL interaction, zero-copy host memory mapping, etc.

While this low-level interface translation is relatively straightforward, care was taken to make the interface a "good citizen" of the high-level-language system: Memory allocation and resource management concerns are handled automatically in close coordination with the Python garbage collector, avoiding spurious resource shortages. Entities such as textures, code modules, and compute devices are reflected into Python using object-oriented terms, providing better abstraction than the low-level C interface. Errors are detected and reported automatically. Further, programmers of high-level languages expect that their programs do not abort upon executing erroneous code, that most error conditions are recoverable and that useful feedback is available on what happened that caused the error. PyCUDA satisfies these expectations. Care is taken however that these automatisms do not turn into a liability. For example, a program under tight memory constraints may not have the luxury of allowing automatic resource management. For this use case, PyCUDA still allows the user to manually control deallocation of resources.

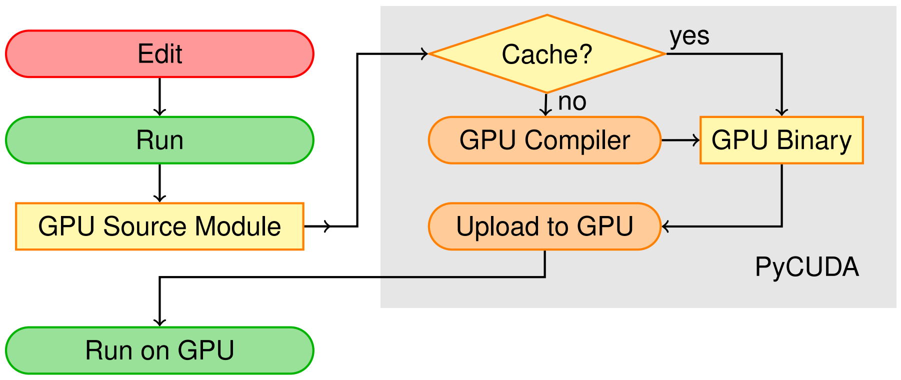


Figure 4.2. Work flow of PyCUDA GPU program compilation. PyCUDA aims to maintain a scripting-like "edit-run-repeat" style of working for the user. The compilation and caching operations in the gray box are performed without user involvement.

The basic shell described so far establishes the basis for more interesting, higher-level features. PyCUDA augments the runtime system by a critical capability: It allows the user to easily create on-GPU binaries simply by providing C-like CUDA[1] source code as a simple character string. This capability is what enables GPU run-time code generation.

Two factors contribute to making this process easy and transparent: First, the user makes no contact with the underlying CUDA compiler infrastructure unless desired. Second, the result of the compilation process is stored in a semi-permanent cache and reused if possible. The cache is sensitive to changes in the hardware and software environment and initiates recompilation when necessary. As a result, compilation of source code and subsequent loading of the binary code becomes nearly instantaneous and invisible to the user, and the quick turn-around time of a scripting-based programming environment is retained. Figure 4.2 illustrates the principle, the end result of which is to make computations specified by C source code a library service that is available cheaply.

Further, whenever GPU RTCG is used for automated tuning, it is desirable that the expense of time and processing power involved in the tuning is only incurred once per

[1]For completeness, it should be mentioned that PyCUDA also allows the just-in-time compilation of code expressed in Nvidia's lower-level "PTX" abstract machine language.

a)

```
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

a = numpy.random.randn(4,4).astype(numpy.float32)
a_gpu = cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(a_gpu, a) # host−to−device

mod = cuda.SourceModule("""
   __global__ void multiply_by_two( float *a)
   {
     int idx = threadIdx.x + threadIdx.y*4;
     a[idx] *= 2;
   }                                Compute Kernel
   """)

func = mod.get_function("multiply_by_two")
func(a_gpu, block=(4,4,1))
```

a) cont'd.

```
a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu) # device−to−host
print a_doubled
print a
```

b)

```
import numpy
import pycuda.autoinit
import pycuda.gpuarray as gpuarray

a_gpu = gpuarray.to_gpu(
     numpy.random.randn(4,4).astype(numpy.float32))
a_doubled = (2*a_gpu).get()
print a_doubled
print a_gpu
```

Figure 4.3. a) An example of the use of PyCUDA, showing the use of the `SourceModule` facility for (static) GPU run-time code generation. This simple program uploads a 4×4 array of single-precision floating point numbers, multiplies them by two on the GPU, and retrieves the result. b) An example performing the same function as a), but using `GPUArray`s.

relevant code change. In most cases, the presence of a compiler cache is already sufficient here, as compilation is usually several orders of magnitude more time-consuming than the actual timing run of the code. However, when that is not the case, PyCUDA supports the building of an application-level cache by offering means for the easy gathering of identifying information regarding hardware, software and their corresponding versions.

The combination of RTCG with services of the run-time system such as high-precision timing and code property access already suffices to enable the strategies laid out in Section 4.3. Figure 4.3a) illustrates, by way of a sample program, how the pieces of PyCUDA explained so far fit together.

4.4.1 Abstractions in PyCUDA

One of the fundamental principles in PyCUDA is that while high-level features are desired, their use should never obstruct access to low-level features, and their use should never obscure the underlying processes. The purpose of this is twofold:

- Uninhibited low-level access ensures that all opportunities for unanticipated uses of low-level facilities are retained.
- Whenever a high-level abstraction is used, the developer deciding to use it assumes a responsibility to know what the abstraction does, fix it if it breaks, or adapt it if is no longer suitable.

Keeping this in mind, PyCUDA does include a number of abstractions, but strives to keep them simple and "flat". It further strives to only include "popular" abstractions that are expected to be useful to a significant share of client codes, lessening the maintenance burden on every individual user.

PyCUDA GPU Arrays

PyCUDA provides computational linear algebra involving vectors and multi-dimensional arrays that are designed to match the interface of the widely-used (CPU-based) Python array package `numpy` [Oliphant, 2006]. This array class, called `GPUArray`, offers a complete set of features, including

- element-wise algebraic operations such as addition, multiplication, etc.,
- a full set of floating-point transcendental as well as utility functions,

a)

```
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
from pycuda.curandom import rand as curand
from pycuda.elementwise import ElementwiseKernel

x = curand((500000,))
y = curand((500000,))
z = gpuarray.empty_like(x)

lin_comb = ElementwiseKernel(
    "float a, float *x, float b, float *y, float *z",
    "z[i] = a*x[i] + b*y[i]")
lin_comb(5, x, 6, y, z)
```

b)

```
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
from pycuda.curandom import rand as curand
from pycuda.elementwise import ElementwiseKernel, \
        VectorArg, ScalarArg

x = curand((500000,))
y = curand((500000,))
z = gpuarray.empty_like(x)

lin_comb = ElementwiseKernel([
    ScalarArg(x.dtype, "a"), VectorArg(x.dtype, "x"),
    ScalarArg(y.dtype, "b"), VectorArg(y.dtype, "y"),
    VectorArg(x.dtype, "z")],
    "z[i] = a*x[i] + b*y[i]")

lin_comb(5, x, 6, y, z)
```

Figure 4.4. Element-wise linear combinations implemented via PyCUDA's element-wise operation code generator, accessible as `pycuda.elementwise.ElementwiseKernel`. a) shows a simple, statically typed version. b) shows a version that relies on type introspection to generate code that is appropriate for the given combination of array types. (The result type is defaulted to the first argument's type for simplicity.)

- type promotion and arbitrary combinations of data types (e.g. adding 32-bit integers to 32-bit floating point values results in 64-bit floating point values to preserve precision),
- reductions such as sums, maxima, and inner products, and
- tight integration with the `numpy` [Oliphant, 2006] Python array package.

Using the `GPUArray` infrastructure, PyCUDA also implements GPU-based sparse matrix-vector multiplication, as described by Garland and Bell [Bell and Garland, 2008, 2009]. Based on this feature, in turn, I was able to include a fast conjugate-gradient-based [Hestenes and Stiefel, 1952] linear system solver, which uses the GPU to solve large systems about ten times faster than competing CPU implementations. Both of these facilities interact seamlessly with the CPU-based SciPy module [Jones et al., 2001–].

On top of `GPUArrays`, PyCUDA offers code generation features for custom element-wise and reduction operations. These work by letting the user specify only short snippets of C code for core functionality, while supplying loop slicing and driver code automatically. Figure 4.4a) illustrates this for the element-wise operation case, implementing a two-vector linear combination. The reduction code generator is similar in spirit. I would like to emphasize the ease with which this simple RTCG tool overcomes the common problem of proliferation of temporary variables plaguing abstract, operator-overloading array packages. C++ packages employing template techniques can achieve a similar degree of efficiency through the *expression template* mechanism [Veldhuizen and Jernigan, 1997], but a robust, usable implementation of this technique is far more complex than the simple generation of C code involved in the RTCG solution. In general, the effort required to create RTCG programs scales very gently with the degree of sophistication required. Figure 4.4b) illustrates this by extending the previous linear combination code to adapt the vector types in the generated code dynamically, by making use of Python's run-time type introspection. It may be argued that these examples look pleasant only because PyCUDA contains a nice enough pre-made user interface that suits this purpose. This is certainly true, but this should be seen in a different light: Only by working in a high-level language was I able to provide this type of user interface. Since providing usable, abstract interfaces is more straightforward in scripting environments, this niceness becomes the rule rather than the exception.

4.4.2 Code Generation with PyCUDA

I now turn to how a user might go about creating abstractions such as `ElementwiseKernel` herself. Since PyCUDA can natively process C code (or rather CUDA's flavor thereof), the objective is the generation of such code. PyCUDA makes no assumptions about the origins of the code it processes, which allows the logic involved in the generation to be designed to match the needs of the application. There are, however, three suggested ways of generating

a)

```
from jinja2 import Template

tpl = Template("""
    __global__ void add(
            {{ type_name }} *tgt,
            {{ type_name }} *op1,
            {{ type_name }} *op2)
    {
      int idx = threadIdx.x +
        {{ thread_block_size }} * {{block_size}}
        * blockIdx.x;

      {% for i in range(block_size) %}
          {% set offset = i*thread_block_size %}
          tgt[idx + {{ offset }}] =
            op1[idx + {{ offset }}]
            + op2[idx + {{ offset }}];
      {% endfor %}
    }""")

rendered_tpl = tpl.render(
    type_name="float", block_size=block_size,
    thread_block_size=thread_block_size)

smod = SourceModule(rendered_tpl)
```

b)

```
from codepy.cgen import FunctionBody, \
        FunctionDeclaration, Typedef, POD, Value, \
        Pointer, Module, Block, Initializer , Assign
from codepy.cgen.cuda import CudaGlobal

mod = Module([
    FunctionBody(
        CudaGlobal(FunctionDeclaration(
            Value("void", "add"),
            arg_decls=[Pointer(POD(dtype, name))
                for name in ["tgt", "op1", "op2"]])),
        Block([
            Initializer (
                POD(numpy.int32, "idx"),
                "threadIdx.x + %d*blockIdx.x"
                % (thread_block_size*block_size)),
            ]+[
            Assign(
                "tgt[idx+%d]" % (o*thread_block_size),
                "op1[idx+%d] + op2[idx+%d]" % (
                    o*thread_block_size,
                    o*thread_block_size))
            for o in range(block_size )]))])

smod = SourceModule(mod)
```

Figure 4.5. Different methods of Run-Time Code Generation (RTCG) with PyCUDA. Example a) generates a piece of C code from a textual template implementing an unrolled version of vector addition. (using the `Jinja2` engine [Ronacher, 2009] in this instance) Example b) builds a data structure approximating a C syntax tree for the same purpose as a). This tree is then converted to C code using the authors' `codepy` package [Klöckner, 2009]. Full context for both examples can be found in the PyCUDA source tree as `examples/demo_meta_template.py` and `examples/demo_meta_codepy.py`.

code which I have found to cover a variety of needs.

Simple textual keyword replacement. This simple technique performs the equivalent of search-and-replace on source code. It suffices for a surprisingly large range of use cases, such as the substitution of types and constants into source code at run time. Its technological reach is increased by combining it with C preprocessor macros. Further contributing to its attractiveness, Python's standard library can perform keyword substitution without relying on external software.

Textual Templating. For code generation applications where control flow and conditionals are required, but all code variants are textually related, the use of a so-called templating engine, commonly used for the generation of web pages, offers a natural escalation of the capabilities of keyword substitution. Many templating engines (and correspondingly, templating languages) exist. Figure 4.5a) demonstrates the use of the Jinja2 [Ronacher, 2009] engine for the generation of a simple, partially unrolled vector addition code.

Syntax Tree Building. The use of templating finds its limits if the codes to be generated cease to be textually related. Then it becomes appropriate to introduce a full representation of the target code in the host language. The most general such representation is in the form of a syntax tree. Syntax tree building allows code to be generated using all facilities of the host language. In particular, while templating is mostly "flat" and oriented along the lines of the output, syntax tree building allows the user to use, e.g., a hierarchy of functions to generate the desired code.

Figure 4.5b) demonstrates the use of the authors' CodePy [Klöckner, 2009] package for the generation of the same unrolled vector addition code as in the previous example. Comparing Figures 4.5a) and b) also reveals that syntax tree generation does not represent a "giant leap" when compared to templating. This again serves to emphasize the gentle growth of complexity in GPU RTCG with PyCUDA.

I have already emphasized various times that one of the central goals of PyCUDA is to facilitate the construction of abstractions, the more sophisticated of which amount to *domain-specific languages*. From a compiler construction perspective, the three strategies above amount to using C as an intermediate representation in the building of a compiler for such a language. Given that PyCUDA is not aimed at optimization at the lowest, machine-language levels, this seems to be an appropriate choice.

PyCUDA is available from `http://mathema.tician.de/software/pycuda` under the liberal MIT open-source software license. Full documentation is available online and packaged with the distribution, along with a large body of examples and tests. The package supports all platforms on which CUDA is available. PyCUDA has been used in a variety of research codes (see Section 4.5 for a few examples). In addition, PyCUDA can be used interactively from the command line as well as from the notebook interface of the Sage exploratory computation system [Stein and Joyner, 2005].

4.4.3 PyOpenCL: OpenCL and GPU RTCG

For those concerned about the vendor specificity of the CUDA compute abstraction, PyOpenCL, a sister project of PyCUDA, has recently been released by the authors under the same terms and is available from `http://mathema.tician.de/software/pyopencl`. It targets the OpenCL [The Khronos OpenCL Working Group, 2008] industry standard compute abstraction. PyOpenCL extends the methods presented thus far to a significantly wider range of devices and vendors. At the time of this writing, PyOpenCL enables the basic premise of this chapter, but has not yet grown to include most of the high-level facilities available in PyCUDA.

4.5 Successful Applications

PyCUDA has been used successfully in a considerable number of research projects. I outline a few projects and their use of RTCG in detail below. Beyond those, the following researchers have agreed to let me mention their use of PyCUDA:

- Ian Cullinan and the SAFE Advanced Surveillance group at NICTA are using PyCUDA to search large facial image databases. Their work seamlessly integrates a GPU-accelerated search algorithm with a Python web interface written using the

Django framework. Using PyCUDA for this task approximately halved the time it takes to run a search.

- Tomasz Rybak at Bialystok Technical University is applying GPU computing to the generation of recurrence diagrams for time series analysis. Using PyCUDA for his analyses, he was able to achieve an 85-fold speedup of his computations. He is using code generation strategies to achieve even greater speeds in cases when data set characteristics allows for using faster memory.

- Chris Heuser with the Center for the Study of Complex Systems at the University of Michigan used PyCUDA to implement an agent-based model. PyCUDA allowed for the easy integration of many of the model's features. In the future, RTCG will be used to allow run-time alterations of agent characteristics, world size, and other model parameters.

- Romain Brette and Dan Goodman are using PyCUDA to simulate spiking neural networks with their simulator "Brian" [Goodman and Brette, 2008]. Brian relies on PyCUDA to generate run-time GPU code for the integration of differential equations provided by the user in a Python script. GPU performance was up to 60 times faster than a comparable CPU implementation for some models.

- Nicolas Pinto, David Doukhan, James J. DiCarlo, and David D. Cox [Pinto et al., 2009] are using PyCUDA as a means of accelerating their investigation in Computational Visual Neuroscience. Their approach involves the highly parallel, high-throughput screening of very many brain-inspired models of the human visual cortex. Because of the multitude of parameters examined, RTCG is crucial to their approach.

- Bryan Catanzaro, Yunsup Lee and coworkers [Catanzaro et al., 2009] have used PyCUDA as the foundation of higher level programming tools, performing Selective Embedded Just In Time Specialization [Catanzaro et al., 2009]. The idea behind

SEJITS is to provide highly productive environments for parallel programming through the use of specialized runtime code-generation. They create domain specific modules, called specializers, which use metaprogramming to analyze a high level description of a particular computation, and then perform JIT code generation for that particular computation.

- Lastly, of course, in the implementation of discontinuous Galerkin finite element PDE solvers, as will be described in Chapter 5.

A comprehensive, up-to-date listing of successful uses of PyCUDA, PyOpenCL and GPU run-time code generation in general is maintained on the web at `http://wiki.tiker.net/PyCuda/ShowCase`.

4.6 Conclusions

I have described the powerful consequences of the confluence of two events in high-performance computing: First, the emergence of general-purpose programmable GPUs as a viable mass market product has made performance jumps of an order of magnitude or more a reality for a number of important applications. Second, the maturing of open-source scripting languages and their software ecosystems has enabled similar jumps in productivity for creators of scientific software. It is straightforward to see that a hybrid model combining GPUs and scripting offers numerous advantages over more traditional models of software creation.

The main message of this chapter is that through the natural addition of GPU run-time code generation to this mixture, one automatically combines the strengths and compensates for the weaknesses of each of the technologies involved, leading to a compelling way of constructing high-performance computational software.

To make GPU RTCG accessible, I have built, documented, and published PyCUDA, a toolkit that allows the easy application of the principles described here. I have described the facilities available in PyCUDA and demonstrated their use. I will continue to extend and maintain both PyCUDA and PyOpenCL.

Based on these toolkits, I will explore the construction of tools that allow researchers to focus on their target areas, while leaving the detailed work involved in accomplishing basic computational tasks to the machine. One effort that is currently underway will use empirical optimization to try and find well-performing kernels for a certain set of basic array operations, such as those involved in dense numerical linear algebra or certain PDE solvers. Further, it should not be forgotten that PyCUDA was born out of the need of actual applications, as Section 4.5 illustrated. As the research in these application areas progresses, I fully expect that more advanced needs will drive the implementation of even better tools.

In summary, I believe that the flexibility of run-time generated code provides a crucial tool in unlocking the performance capabilities of advanced hardware to a broader mass of developers, and I look forward to the opportunities and challenges that future hardware generations will bring.

CHAPTER FIVE

Discontinuous Galerkin Methods on Graphics Processors

5.1 Introduction

In this chapter, I will explore how and with what benefit discontinuous Galerkin (DG) methods as introduced in Section 2.1 can be brought onto GPUs.

Two main questions arise in this endeavor: First, how shall the computational work be partitioned? In a distributed-memory setting, the answer is quite naturally domain decomposition. For the shared-memory parallelism of a GPU, there are several possibilities, and there is often no single answer that works well in all settings. Second, DG implementations on serial processors often rely heavily on the availability of off-the-shelf, pre-tuned linear algebra and communication primitives. These aids are either unavailable or unsuitable on a GPU platform, and in stark contrast to the relatively straightforward implementation of DG on serial machines, optimal use of graphics hardware for DG presents the implementer with a staggering number of choices. I will describe these choices as well as a generative approach that exploits them to adapt the method to both the problem and the hardware at run time.

Using graphics processors for computational tasks is by no means a new idea. In fact, even in the days of marginally programmable fixed-function hardware, some (especially particle-based) methods obtained large speedups from running on early GPUs. (e.g. [Li et al., 2003]) In the domain of solvers for partial differential equations, Finite-Difference Time-Domain (FDTD) methods are a natural fit to graphics processors and obtained speedups of about an order of magnitude with relative ease (e.g., [Krakiwsky et al., 2004]). Finite Element solvers were also brought onto GPUs relatively early on (e.g., [Göddeke et al., 2005]), but often failed to reach the same impressive speed gains observed for the simpler FD methods. In the last few years, high-level abstractions such as Brook and Brook for GPUs [Buck et al., 2004] have enabled more and more complex computations on streaming hardware. Building on this work, Barth et al. [Barth and Knight, 2005] already predicted promising performance for two-dimensional DG on a simulation of the Stanford Merrimac streaming architecture [Dally et al., 2003]. Nowadays, compute abstractions are becoming less encumbered by their graphics heritage [Lindholm et al., 2008, Nvidia Corporation, 2009]. This has helped bring algorithms of even higher complexity onto the GPU (e.g. [Gumerov and Duraiswami, 2008]). Taking advantage of these recent advances, this chapter presents, to the best of my knowledge, one of the first general finite-element based solvers that achieves more than an order of magnitude of speedup on a single real-world consumer graphics processor when compared to a CPU implementation of the same method.

A sizable part of this speedup is owed to my use of high-order approximations. High-order methods require more work per degree of freedom than low-order methods. This increased arithmetic intensity shifts the method from being limited by memory bandwidth towards being limited by compute bandwidth. The relative abundance of cheap computing power on a GPU makes high-order methods especially beneficial there.

In this chapter, I will discuss the numerical solution of linear hyperbolic systems of conservation laws using DG methods on the GPU. Important examples of this class of partial differential equations (*PDE*s) include the second-order wave equation, Maxwell's equations, and many relationships in acoustics and linear elasticity. Certain nontrivial adjustments to the discontinuous Galerkin method become necessary when treating nonlinear problems (see, e.g., [Hesthaven and Warburton, 2007, Chapter 5]). One possible scheme to deal with the issues arising in the solution of nonlinear systems of conservation laws using DG on a GPU will be investigated in the subsequent chapter.

I will further focus on tetrahedra as the basic discretization element for a number of reasons. First, it is undisputed that three-dimensional calculations are in many cases both more practically relevant and more plagued by performance worries than their lower-dimensional counterparts. Second, they have the most mature meshing machinery available of all commonly used element shapes. And third, when compared with tensor product elements, tetrahedral DG is both more arithmetically intense and requires fewer memory fetches. Overall, it is conceivable that tetrahedral DG will benefit more from being carried out on a GPU.

This chapter describes the mapping of DG methods onto the Nvidia CUDA programming model. Hardware implementations of CUDA are available in the form of consumer graphics cards as well as specialized compute hardware. In addition, the CUDA model has been mapped onto multi-core CPUs with good success [Stratton et al., 2008]. Rather than claim an artificial generality, I will describe my approach firmly in the context of this model of computation. While that makes this work vendor-specific, I believe that most of the ideas presented herein can be reused either identically or with mild modifications to adapt the method to other, related architectures. The emerging OpenCL industry standard [The Khronos OpenCL Working Group, 2008] specifies a model of parallel computation that is a very close relative of CUDA, promising broad applicability of the methods presented herein. It should be noted, however, that OpenCL can be used with a multitude of device types whose suitability for DG in general and my methods in particular will of course vary.

The chapter is organized as follows: I give a brief overview of the theory and serial implementation of DG in Section 2.1. The CUDA programming model is described in Section 2.2. Section 5.2 explains the basic design choices behind my approach, while Section 5.3 gives detailed implementation advice and pseudo-code. Section 5.4 characterizes my computational results in terms of speed and accuracy. Finally, in Section 5.5 I conclude with a few remarks and ideas for future work.

5.2 DG on the GPU: Design

The answers to three questions emerge as the central design decisions in mapping a numerical method into an algorithm that can run on a GPU:

Computation Layout. How can the task be decomposed into a grid of thread blocks, given there cannot be any inter-block communication? Is it necessary to use a sequence of grids instead of a single grid?

Data Layout. How well does the data conform to the device's alignment requirements? Where and to what extent will padding be used?

Fetch Schedule. When will what piece of the data be fetched from global into on-chip memory, i.e. registers or shared memory?

Note that the computation layout and the data layout are often the same, and rarely independent. For the bandwidth reasons described in Section 2.2, the index of the thread computing a certain result should match the index where that result is stored. Post-computation permutations come at the cost of setting aside shared memory to perform the permutation. It is therefore common to see algorithms designed around the principle of *one thread per output*. The fetch schedule, lastly, determines how often data can be reused before it is evicted from on-chip storage.

Unstructured discontinuous Galerkin methods have a number of natural granularities:

- the number N_p of DOFs per element,
- the number N_{fp} of DOFs per face,
- the number N_f of faces per element,

- the number n of unknowns in the system of conservation laws.

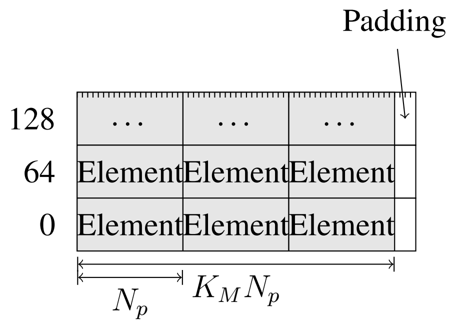
The number of elements K also influences the work partition, but it is less important in the present discussion.

The first three granularities above depend on the chosen order of approximation as well as the shape of the reference element. Figure 5.1(a) gives a few examples of their values. Perhaps the first problem that needs to be addressed is that many of the DOF counts, especially at the practically relevant orders of 3 and 4, conform quite poorly to the hardware's preference for batches of 16 and 32. A simple solution is to round the size of each element up to the next alignment boundary. This leads to a large amount of wasted memory. More severely, it also leads to a large amount of wasted processing power, assuming a one-thread-per-output design. For example, rounding N_p for a fourth-order element up to the next warp size boundary ($T = 32$) leads to 45% of the available processing power being wasted. It is thus natural to aggregate a number of elements to get closer to an alignment boundary. Now, each of the parts of a DG operator is likely to have its own preferred granularity corresponding to one thread block. One option is to impose one such part's granularity on the whole method. I find that a better compromise is to introduce a sub-block granularity for this purpose. My method aggregates the smallest number K_M of elements to achieve less than 5% waste when padding up to the next multiple N_{pM} of $T/2 = 16$. Figure 5.1(b) illustrates the principle. I then impose the restriction that each thread block work on an integer number of these *microblocks*. I assign the symbol $n_M := \lceil K/K_M \rceil$ to the total number of microblocks.

The next question to be answered involves decomposing a task into an appropriate set of thread blocks. This decomposition is problem-dependent, but a few things can be said in general. For instance, assume a task that has to be performed in parallel, independently, on a number of work units, and that requires some measure of preparation before actual

N	N_p	N_{fp}	$N_f N_{fp}$
1	4	3	12
2	10	6	24
3	20	10	40
4	35	15	60
5	56	21	84
6	84	28	112
7	120	36	144

(a) DOF counts for moderate-order tetrahedral elements.



(b) Microblocked memory layout.

Figure 5.1. Matching DG granularities to GPU alignment boundaries.

work units can be processed. The issue is to find the right amount of work to be done by a single thread block. One may let the block complete work units in parallel, alongside each other in a single thread (‘*in-line*’ for brevity), or sequentially. I will use the symbols w_p, w_i and w_s for the number of work units processed in each way by one thread block. Thus the total number of work units processed by one thread block is $w_p w_i w_s$. A large w_p may improve speed through increased parallelism and reuse of data in shared memory, but typically also requires additional shared memory buffer space. Increasing w_i gains speed through reuse of data in registers. Take, for example, a two-operand procedure like matrix multiplication. Here, increasing w_i allows a single thread to use data from the first operand, once loaded into registers, to process more than one column of the second operand. Like w_p, varying w_i also influences buffer space requirements. w_s, finally, amortizes preparation work over a certain number of work units, at the expense of making the computation more granular. Achieving a balance between these aspects is not generally straightforward, as Figure 5.8(b) will demonstrate. Note that each of the methods discussed below will have its own values for w_p, w_i, and w_s.

I noted above that the number n of variables in the system of conservation laws (2.1) also introduces a granularity. In some cases, it may be advantageous to allow this system size to play a role in deciding data and computation layouts. One might attempt do this by choosing a packed field layout, i.e. by storing all field values at one node in n consecutive memory locations. However, a packed field layout is not desirable for a number of reasons, the most significant of which is that it is unsuited to a one-thread-per-output computation. If thread 0 computes the first field component, thread 1 the second, and so on, then each field component is found by evaluating a different expression, and hence by different code. This cannot be efficiently implemented on SIMT hardware. One could also propose to take advantage of the granularity n by letting one thread compute all n different expressions in the conservation law for one node. It is practical to exploit this for the gathering of the fluxes and the evaluation of $F(u)$. For the more complicated lifting and differentiation stages on the other hand, this leads to impractical amounts of register pressure. I find that, especially at moderate orders, the extra flexibility afforded by ignoring n outweighs any advantage gained by heeding it. If desired, one can always choose $K_M = n$ or $w_i = n$ to closely emulate the strategies above. Further, note that for the linear case discussed here, one has significant freedom in the ordering of operations, for example by commuting the evaluation of $F(u^k)$ with local differentiation.

A final question in the overall algorithm design is whether it is appropriate to split the DG operator into the subtasks indicated in Figure 2.2, rather than to use a single or only two grids to compute the whole operator. Field data would need to be fetched only once, leading to a good amount of data reuse. But at least for the scarce amounts of shared memory buffer space in current-generation hardware, this view is too simplistic. Each individual subtask tends to have a better, individual use for on-chip memory. Also, it is tempting to combine the gather and lift stages, since one works on the immediate output of the other. Observe however that there is a mismatch in output sizes between the two. For each element, the gather outputs $N_{fp}N_f$ values, while the lift outputs N_p. These two numbers differ, and therefore the optimal computation layouts for both kernels also differ. While it is possible to use the larger of the two computation layouts and just idle the overlap for the other computation, this is suboptimal. I find that the added fetch cost is easily amortized by using an optimal computation layout for each part of the flux treatment.

5.3 DG on the GPU: Implementation

5.3.1 How to read this Section

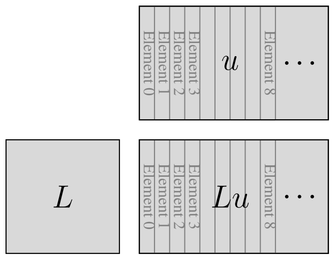
To facilitate a detailed, yet concise look at my implementation techniques, this section supplements its discussion with pseudo-code for some particularly important subroutines. Pseudo-code contains all the implementation details and exposes the basic control and synchronization structure at a single glance. In addition to the code, there is text discussing every important design decision reflected in the code.

To maximize readability, I will rely on a number of notational conventions. First, $\lceil x \rceil_n$ is the smallest integer larger than x divisible by n. Next, $[a, b\rangle$ denotes the 'half-open' set of integers $\{a, \dots, b-1\}$. Using this notation, I may indicate 'vectorized' statements, e.g. an assignment $a_{[k,k+n\rangle} \leftarrow k_{[0,n\rangle}$. The loops indicated by these statements are always fully unrolled in actual code. Depending on notational convenience, I alternate between subscript notation a_i and indexing notation $a[i]$. Both are to be taken as equivalent. Sometimes, I use both sub- and superscripts on a variable. This helps brevity and readability, but is only done if the memory layout of the corresponding variable is clarified elsewhere. Otherwise, for multidimensional indices, C-like (row-major) data layout is assumed.

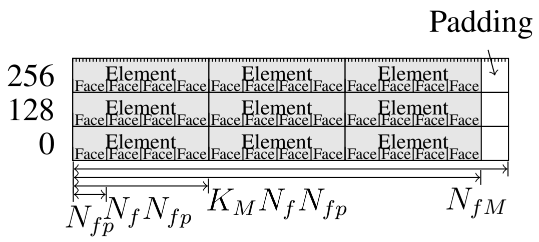
Lastly, the GPU offers many different types of storage. To avoid confusion, I assign each type of storage a separate typographical convention, as outlined in Table 5.1. If and only if two storage locations of different types are used for related data, I use the same

Convention		Storage Type
v	Italic font	Constant or unrolled loop variable
$\texttt{v}$	Typewriter font	Register variable
$\texttt{v}^{\texttt{S}}$	Superscript $\texttt{S}$	Variable in shared memory
$\texttt{v}^{\texttt{G}}$	Superscript $\texttt{G}$	Variable in global memory
$\texttt{v}^{\texttt{T}}$	Superscript $\texttt{T}$	Variable bound to a texture

Table 5.1. Typographical conventions for different types of GPU storage.



(a) Applying an element-local DG operator L to a field u by a matrix-matrix product.



(b) Output memory layout for the flux gather stage, input memory layout of the flux lifting stage.

Figure 5.2. Implementation aspects of flux lifting.

letter for both.

5.3.2 Flux Lifting

Lifting is one of the *element-local* components of a discontinuous Galerkin operator, and, for simplicial elements, is efficiently represented by a matrix-matrix multiplication as in Figure 5.2(a), followed by an element-wise scaling.

The first, tempting approach to implementing this is to take advantage of the vendor-provided GPU-based BLAS work-alike. This is hampered by sub-optimal performance and strict alignment requirements. As a result, a custom algorithm is in order.

One key to high performance on the GPU is to find a good use for the scarce amount of shared memory. Both operands in an element-local matrix multiplication see large amounts of reuse: Each field value is used N_p times, and each entry of a local matrix is used N_p times *for each element*. It is therefore a sensible wish to load both operands into shared memory. For the tetrahedral elements targeted here, this is problematic. Even for elements of modest order, the matrix data quickly becomes too large. This restricts the applicability of a matrix-in-shared approach to low orders, and I will therefore first examine the more broadly applicable method of using the shared memory for field data. Still, matrix-in-shared does provide a benefit for certain low orders and is examined in the context of element-local differentiation in Section 5.3.4.

I choose a one-thread-per-output design for flux lifting. This dictates that computation and output layouts match Figure 5.1(b). But the input layout for lifting is mildly different: The flux gather, which provides the input to lifting, extracts $N_f N_{fp}$ DOFs per element. Recall that the layout of Figure 5.1(b) provides N_p DOFs per element. Since typically $N_p \neq N_f N_{fp}$, I introduce a mildly different layout as shown in Figure 5.2(b), using the same number K_M of elements as found in a microblock, padded to half-warp granularity. This padding is likely somewhat more wasteful than the carefully tuned one of Figure 5.1(b). Fortunately, this is irrelevant: I will not be using Figure 5.2(b) as a computation layout, and data in this format is used only for short-lived intermediate results. Overall, the resulting memory layout has $N_{fM} := \lceil N_f N_{fp} K_M \rceil_{T/2}$ DOFs per microblock.

I am now ready to discuss the actual algorithm, at the start of which one needs to fetch field data into shared memory. Because I chose a one-thread-per-output computation layout, I will have N_p threads per element fetching data. Due to the mismatch between N_p and $N_f N_{fp}$, multiple fetch cycles may be required to fetch all data. In addition, the last such fetch cycle must involve a length check to avoid overfetching. It is important to unroll this fetch loop and to use some care with the ending conditional to still allow fetch pipelining[1]

[1]Pipelining is a fetch optimization strategy. It performs high-latency fetches in batches ahead of a

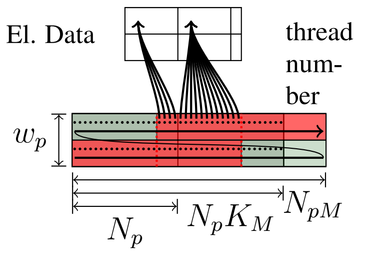
to occur.

With the field data in shared memory, the matrix data is fetched using texture units. By way of the texture cache, I hope to take advantage of the significant redundancy in these fetches. The matrix texture should use column-major order: Realize that within a block, a large number of threads, each assigned to a row of the matrix, load values from each column in turn. Column-major order gives the most locality to this access pattern.

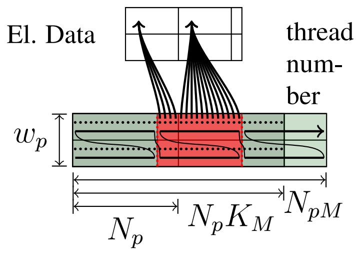
With this preparation, the actual matrix-matrix product can be performed. Since all threads within one element load each of the element's nodal values from shared memory in order, these accesses are handled as a broadcast and therefore conflict-free. Conflicts do occur, however, if a half-warp straddles an element boundary within a microblock. In that case, threads before and after the element boundary access different elements, and therefore a double-broadcast bank conflict occurs. Figure 5.3(a) shows the genesis of this conflict. Fortunately, that does not automatically mean that microblocking is a bad idea. It turns out that the performance lost when using no microblocking and hence full padding is about the same as the one lost to these bank conflicts. Even better: there is a way of mitigating the conflicts' impact *without* having to forgo the performance benefits of microblocking. The key realization is that even if only one half of a warp encounters a conflict, the other half of the warp is made to wait, too, regardless of whether it conflicted. Conversely, if one assembles warps in such a way that conflict-prone and non-conflict-prone half-warps are kept separate, then one avoids unnecessary stalling. If $w_p > 1$, then one can achieve such a grouping by laying out the computation as seen in Figure 5.3(b).

Algorithm 5.1 represents the aggregate of the techniques described in this section.

computation. Since a warp only stalls when unavailable data is actually used in a computation, this allows a single thread to wait for multiple memory transactions simultaneously, decreasing latency and reducing the need for parallel occupancy. The Nvidia compiler automatically pipelines fetches if the code structure allows it.



(a) ‘Conventional’, conflict-aggravating layout. The first and third warp (red) serialize access because of conflicts in the second half-warp of each microblock. Only the second warp (green) proceeds without conflicts.



(b) Improved, conflict-mitigating layout. Only the second warp (red) serializes access for conflicts. The first and third warp (green) remain conflict-free.

Figure 5.3. Computation layouts for matrix multiplication with fields in shared memory.

Observe that since there is no preparation work, one may set $w_s := 1$. I should stress at this point that both the field-in-shared and the matrix-in-shared approach can be used for both lifting and element-local differentiation. Adapting the strategy of Algorithm 5.1 for the latter is quite straightforward.

5.3.3 Flux Extraction

In a strong-form, nodal implementation of the discontinuous Galerkin method, flux extraction or ‘gather’ iterates over the node indices of each face in the mesh and evaluates the flux expression (2.5) at each such node. As such, it is a rather quick operation characterized by few arithmetic operations and a very scattered fetch pattern. This non-local memory access pattern is the most expensive aspect of flux extraction on a GPU, and one’s foremost goal should therefore be to minimize the number of fetches at all costs. For linear conservation laws, one may with very little harm treat the element-local parts of a DG operator as if they acted on scalar fields. This is however not true of the non-local flux extraction. Fetching all fields only once and then computing all n fluxes saves a significant $n^2 - n$ fetches of each facial node value.

Algorithm 5.1 Flux Lifting, field-in-shared.

```
Require: A grid of $\lceil n_M/w_p w_i \rceil \times 1$ blocks of size $T/2 \times w_p \times N_{pM}/(T/2)$.
Require: Inputs: $\mathtt{L}^\mathtt{T}$, the reference element's lifting matrix; $\mathtt{i}^\mathtt{T}$, the per-element inverse
    Jacobians; $\mathtt{f}^\mathtt{G}$, the surface fluxes in the format of Figure 5.2(b).
Ensure: Output: $\mathtt{r}^\mathtt{G}$, the surface fluxes $\mathtt{f}^\mathtt{G}$ multiplied by the per-element lifting matrix $L^k$.
  $\mathtt{m} \leftarrow (b_x w_p + t_y) w_i$ { the base microblock number }
  $\mathtt{i} \leftarrow (T/2) t_z + t_x$ { this thread's DOF number within its microblock }
  { load data }
  for all unrolled $b \in [0, \lceil \lceil N_{fM} \rceil_T / \lceil N_{pM} \rceil_T \rceil \rangle$ do
      if $b N_{pM} + \mathtt{i} < N_{fM}$ then
          $\mathtt{f}^\mathtt{S}_{t_y,[0,w_i\rangle, b N_{pM} + \mathtt{i}} \leftarrow \mathtt{f}^\mathtt{G}_{(\mathtt{m}+[0,w_i\rangle)\lceil N_{fM} \rceil_T + b N_{pM} + \mathtt{i}}$
      end if
  end for
  ―――――― Barrier+Memory Fence ――――――
  { perform matrix multiply }
  if $\mathtt{i} < K_M N_{pM}$ then
      $\mathtt{r}_{[0,w_i\rangle} \leftarrow 0$
      for all unrolled $n \in [0, N_f N_{fp} \rangle$ do
          $\mathtt{r}_{[0,w_i\rangle} \leftarrow \mathtt{r}_{[0,w_i\rangle} + \mathtt{L}^\mathtt{T}[\mathtt{i} \bmod N_p, n] \mathtt{f}^\mathtt{S}_{t_y,[0,w_i\rangle,n}$
      end for
      $\mathtt{r}^\mathtt{G}_{(\mathtt{m}+[0,w_i\rangle) N_{pM} + \mathtt{i}} \leftarrow \mathtt{i}^\mathtt{T}[(\mathtt{m} + [0,w_i\rangle) K_M + \lfloor \mathtt{i}/N_p \rfloor] \mathtt{r}_{[0,w_i\rangle}$
  end if
```

The next potential savings comes from the fact that the fluxes on the two sides of an interior face pair use the same face data. By computing fluxes for such face pairs together, one can cut the number of interior face fetches in half. Computing and storing opposite fluxes together is of course only possible if the task decomposition assigns both to the same thread block. This decomposition should therefore be constructed carefully.

To help find the properties of the task decomposition, observe that by choosing to compute opposite fluxes together, I am implicitly rejecting a one-thread-per-output design. To accommodate opposite faces' fluxes being computed simultaneously, I will allow the gathered fluxes to be written into a shared memory buffer in random order in time, but conforming to the output layout of Figure 5.2(b). Once completed, this shared memory buffer will then be flushed to global memory in one contiguous write operation. This limits the range of possible choices for task decomposition: Thread blocks will output contiguous pieces of data in the output layout. This means that the smallest granularity on which a thread block for flux extraction may begin and end is that of a microblock: I will let each thread block compute fluxes on an integer number M_B of microblocks. Observe that this is not ideal: The natural task decomposition for flux extraction is by face pair, not by element, nor, even worse, by a group of elements as large as a microblock. Nonetheless, given my output memory layout, this decomposition is inevitable.

But all is not lost. By carefully controlling the assignment of elements to microblocks, and again by carefully choosing the assignment of microblocks to flux extraction thread blocks, I can hope to recover many block-interior face pairs within a thread block. Note the far-reaching consequences of what was just decided: One needs to have the elements participating in a flux-gather thread block sit adjacent to each other in the mesh. To achieve this, I partition the mesh into pieces of at most $K_M M_B$ elements each and then assign the elements in each piece to microblocks sequentially. This means nothing less than letting the mesh numbering be decided by what is convenient for the gathering of fluxes.

What can be said about the required partition? It is important to realize that this is a fairly different domain decomposition problem than the one for distributed-memory machines. First, there is a hard limit of $K_M M_B$ elements per piece, as determined by the amount of shared memory set aside for write buffering. Second, there is a (somewhat softer) limit on the number of block-external faces. This limit stems from the fact that information about the faces on which fluxes are gathered needs to be stored somewhere. Obviously, block-internal face pairs can share this information and therefore require less storage–one descriptor for each two faces. Face pairs on a block boundary are less efficient. They require one descriptor for each face. If the block size $K_M M_B$ is relatively large, a bad, splintered partition may have too many boundary faces and therefore exceed the “soft” limit on available space for face pair descriptors. Therefore, for large blocks, one requires a ‘good’ partition with as few block-exterior face pairs as possible. For very small blocks, on the other hand, the problem is exactly opposite: If $K_M M_B$ is small, the absolute quality of the mesh partition is not as critically important: The small overall number of faces means that one will not run out of descriptor space, making the soft limit even softer.

So, how can the needed partition be obtained? A natural first idea is to use conventional graph partitioning software (e.g. [Karypis and Kumar, 1999]). Problematically, these packages tend to fail when partitioning very large meshes into very many small parts. In addition, the ‘soft’ and ‘hard’ limits are difficult to enforce in these packages, so that obtaining a conforming partition may take several ‘attempts’ with increasing target partition sizes. Increased target partition sizes, in turn, mean that there are microblocks where element slots go unassigned. This means that generic graph partitioners are not a universal answer. They work well and generate good-quality partitions if $K_M M_B \gtrapprox 10$. Otherwise, I fall back on a simple greedy breadth-first agglomerator designed to exactly meet the ‘hard’ limit. It picks elements by a total connectivity heuristic and is illustrated in Algorithm 5.2. The greedy algorithm may produce a few very ‘bad’ scattered blocks with many external faces, but I have found that they matter neither in performance, nor in keeping the ‘soft’ limit.

Once the partition is constructed, one obtains for each block a number of elements whose faces fall into one of three categories: intra-block interior, inter-block interior, and boundary faces. One then designs the algorithm to walk an array of data structures describing face pairs, each of which falls into one of these categories. Within this array, each face pair structure contains all information needed to gather and compute the fluxes for its target face(s). Descriptors for intra-block interior face pairs drive the flux computation for two faces at once, while the other two kinds only drive the computation for one face. The array is loaded from global into shared memory when each thread block begins its work. To minimize branching and to save storage space in each descriptor, one makes the kind of each face pair descriptor implicit in its position in the array. To achieve this, one

Algorithm 5.2 Simple Greedy Partition.

```
Require: Input: set of elements $E$ with connectivity $C := \{(e_1, e_2) : e_1$ and $e_2$ share a face$\}$.
Ensure: Output: the partition, a set of blocks $P$, each of size $\leq l$.
  $P \leftarrow \emptyset$
  while $E \neq \emptyset$ do
      $Q \leftarrow \{$a seed element from $E\}$ (a queue of candidate elements)
      $B \leftarrow \emptyset$ (the block currently being generated)
      loop
          Find and remove the element $e \in Q$ that shares the most faces with $B$.
          if $e \in E$ then
              Remove $e$ from $E$, add it to $B$.
              if $|B| = l$ then
                  Make first entry of $Q$ the new seed element, break the loop.
              end if
              $Q \leftarrow Q \cup \{f : (e, f) \in C\}$
          end if
          if $Q = \emptyset$ then
              if $E = \emptyset$ then
                  Break the loop.
              else
                  Add an arbitrary element from $E$ to $Q$.
              end if
          end if
      end loop
      $P \leftarrow P \cup \{B\}$
  end while
```

orders the array by the face pair's category and store how many face pairs of each category are contained in the array.

Because I am implementing a nodal DG method, face index lists play an important role in the gather process: Each face's nodal values need to be extracted from a given volume field. Since a tetrahedron has four faces, there are four possible index subsets at which each face's DOFs are found, all of length N_{fp}. Knowing these index subsets enables one to find surface nodal values for one element. But one needs to find *corresponding* nodal values on two opposite elements. Therefore, one may need to permute the fetch ordering of one of the elements in a face pair. Altogether, to find opposing surface nodal values, one needs to store two index lists. Since the number of distinct index lists is finite, it is reasonable to remove each individual index list from the face pair data structure and to instead refer to a global list of index lists. I find that a small texture provides a suitable storage location for this list. Finally, note that intra-block face pairs require another index list: If one strives to conform to an assumed 'natural' face ordering of one 'dominant' face, writing the other's data into the purely facial structure from Figure 5.2(b) requires a different index list than the one needed to read the element's volume data.

Of all the parts of a DG operator, the flux gather stage is the one that is perhaps least suited to execution on a GPU. The algorithm is data-driven and therefore branch-intensive, it accesses memory in an erratic way, and, as n grows, it tends to require a fair bit of register space. It is encouraging to see that despite these issues, it is possible to design a method, given in Algorithm 5.3, that performs respectably on current hardware.

Algorithm 5.3 Flux Extraction.

```
Require: a grid of $\lceil n_M/M_B \rceil \times 1$ blocks of size $N_{fp} \times w_p \times 1$.
Require: Inputs: $(\mathtt{u}^{\mathtt{T}})^{[0,n\rangle}$, the set of fields of which fluxes are to be computed, each as a
    separate texture, $\mathtt{d}^{\mathtt{G}}$, face information records, $\mathtt{J}^{\mathtt{T}}$, face index list array.
Ensure: Outputs: $(\mathtt{f}^{\mathtt{G}})^{[0,n\rangle}$, the surface fluxes for each face of each element, as a sequence
    of scalar fields.
  Load face information records from $\mathtt{d}^{\mathtt{G}}[b_x]$ into the shared memory variable $\mathtt{d}^{\mathtt{S}}$.
  ———— Barrier+Memory Fence ————
  $\mathtt{e} \leftarrow t_y$ { *initialize the number of the face pair this thread is working on* }
  while $\mathtt{e} <$ # of interior face pairs in $\mathtt{d}^{\mathtt{S}}$ do
      $(\mathtt{i}^-, \mathtt{i}^+) \leftarrow \mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{fetch\_base}^{-,+} + \mathtt{J}^{\mathtt{T}}[\mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{fetch\_idx\_list\_nr}^{-,+}, t_x]$
      $\mathtt{u}_{-,+}^{[0,n\rangle} \leftarrow (\mathtt{u}^{\mathtt{T}})_{\mathtt{i}^{-,+}}^{[0,n\rangle}$
      $(\mathtt{f}^{\mathtt{S}})^{[0,n\rangle}[\mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{store\_base}^- + t_x]$
          $\leftarrow \mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{face\_jacobian} \cdot [\hat{n} \cdot F - (\hat{n} \cdot F)^*]^{[0,n\rangle}(\mathtt{u}_-^{[0,n\rangle}, \mathtt{u}_+^{[0,n\rangle})$
      $(\mathtt{f}^{\mathtt{S}})^{[0,n\rangle}[\mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{store\_base}^+ + \mathtt{j}^{\mathtt{T}}[\mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{store\_idx\_list\_nr}^+, t_x]]$
          $\leftarrow \mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{face\_jacobian} \cdot [(-\hat{n}) \cdot F - ((-\hat{n}) \cdot F)^*](\mathtt{u}_+^{[0,n\rangle}, \mathtt{u}_-^{[0,n\rangle})$
      $\mathtt{e} \leftarrow \mathtt{e} + w_p$
  end while
  while $\mathtt{e} <$ # of interior and exterior face pairs in $\mathtt{d}^{\mathtt{S}}$ do
      $(\mathtt{i}^-, \mathtt{i}^+) \leftarrow \mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{fetch\_base}^{-,+} + \mathtt{J}^{\mathtt{T}}[\mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{fetch\_idx\_list\_nr}^{-,+}, t_x]$
      $\mathtt{u}_{-,+}^{[0,n\rangle} \leftarrow (\mathtt{u}^{\mathtt{T}})_{\mathtt{i}^{-,+}}^{[0,n\rangle}$
      $(\mathtt{f}^{\mathtt{S}})^{[0,n\rangle}[\mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{store\_base}^- + t_x]$
          $\leftarrow \mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{face\_jacobian} \cdot [\hat{n} \cdot F - (\hat{n} \cdot F)^*](\mathtt{u}_-^{[0,n\rangle}, \mathtt{u}_+^{[0,n\rangle})$
      $\mathtt{e} \leftarrow \mathtt{e} + w_p$
  end while
  while $\mathtt{e} <$ # of face pairs in $\mathtt{d}^{\mathtt{S}}$ do
      $\mathtt{i}^- \leftarrow \mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{fetch\_base}^- + \mathtt{J}^{\mathtt{T}}[\mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{fetch\_idx\_list\_nr}^-, t_x]$
      $\mathtt{u}_-^{[0,n\rangle} \leftarrow (\mathtt{u}^{\mathtt{T}})_{\mathtt{i}^-}^{[0,n\rangle}$
      $\mathtt{u}_+^{[0,n\rangle} \leftarrow b(\mathtt{u}_-^{[0,n\rangle}, \mathtt{d}^{\mathtt{S}}[\mathtt{e}])$ { *calculate boundary condition* }
      $(\mathtt{f}^{\mathtt{S}})^{[0,n\rangle}[\mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{store\_base}^- + t_x]$
          $\leftarrow \mathtt{d}^{\mathtt{S}}[\mathtt{e}].\mathtt{face\_jacobian} \cdot [\hat{n} \cdot F - (\hat{n} \cdot F)^*](\mathtt{u}_-^{[0,n\rangle}, \mathtt{u}_+^{[0,n\rangle})$
      $\mathtt{e} \leftarrow \mathtt{e} + w_p$
  end while
  ———— Barrier+Memory Fence ————
  $(\mathtt{f}^{\mathtt{G}})_{b_x M_B N_{fM} + [0, M_B N_{fM}\rangle}^{[0,n\rangle} \leftarrow (\mathtt{f}^{\mathtt{S}})_{[0, M_B N_{fM}\rangle}^{[0,n\rangle}$ (not unrolled)
```

5.3.4 Element-Local Differentiation

Unlike lifting, element-local differentiation must be represented not as one matrix-matrix product (see Figure 5.2(a)), but as $d = 3$ separate ones whose results are linearly combined to find the global x-, y- and z-derivatives. Each of the d differentiation matrices has $N_p \times N_p$ entries and is applied to the same data. To maximize data reuse and minimize fetch traffic, it is immediately apparent that all d matrix multiplications should be carried out "in-line" along with each other.

Superficially, this makes differentiation look quite like a lift where one has chosen $w_i = d$. But there is one crucial difference: the three matrices used for differentiation are all different. Increasing w_i drives data reuse in lifting simply by occupying more registers. As will be seen in Section 5.4, this suffices to make it go very fast. Differentiation on the other hand already has a built-in "w_i multiplier" of d *and* has to deal with different matrices. Both factors significantly increase register pressure. Stated differently, this means that it is unlikely that I will be able to drive matrix data reuse by using more registers as I was able to do for lifting. But the matrix remains the most-reused bit of data in the algorithm. In this section, I will therefore attempt to exploit this reuse by storing the matrix, not the field, in shared memory.

I have already discussed in Section 5.3.2 that the matrix-in-shared approach can only work for low orders because of the rapid growth of the matrix data with N. At first, this seems like a problematic restriction that makes the approach less general than it could be. It can however be turned into an advantage: Since I can assume that the algorithm runs at orders six and below, I can exploit this fact in my design decisions.

I will begin the discussion of this approach by figuring how the matrix data should be loaded into shared memory. As in Section 5.3.2, I will be adopting a one-thread-per-output approach. A straightforward first attempt may be to load all d local differentiation matrices into shared memory in their entirety. Then each thread computes a different row of the matrix-vector product, and in doing so, thread number i accesses the ith row of the matrix. Without loss of generality, let the matrix be stored in row-major order, so that thread i accesses memory cell number iN_p. Shared memory has $T/2 = 16$ distinct memory banks, and therefore the access is conflict-free iff N_p and 16 are relatively prime, or, more simply, iff N_p is odd. This is encouraging: One can achieve a conflict-free access pattern simply by adding a 'padding' column if necessary to enforce an odd stride S. Figure 5.5(a) shows the resulting assignment of matrix data to shared memory banks, and Figure 5.5(b) illustrates the resulting conflict-free access pattern.

Unfortunately, this is too simplistic. In the presence of microblocking, conflict-free access becomes more difficult. If a half-warp straddles one or more element boundaries, bank conflicts are likely to result. The access not only has a stride S, but also incorporates a jump from the end of the matrix to its beginning, a stride of $(N_p - 1)S$. And unlike in the previous case, one cannot simply add a pad row to make the access conflict-free. Figure 5.5(c) displays the problem.

One way to avoid the disastrous end-to-beginning jump and to maintain the conflict-free access pattern would be to duplicate the matrix data from the first rows beyond the end of the matrix. This is workable in principle, but in practice I am already filling the entire shared memory space with matrix data and am unlikely to be able to afford the added duplication. Fortunately, the duplication idea can be saved, and there exists a conflict-free matrix storage layout that does not require one to abandon microblocking.

Departing from the idea that I will store the *entire* matrix, I now aim at storing just a constant-size row-wise *segment* of the matrix. Then, if the end of the matrix falls within a segment, I fill up the rest of the segment with rows from the beginning, providing the

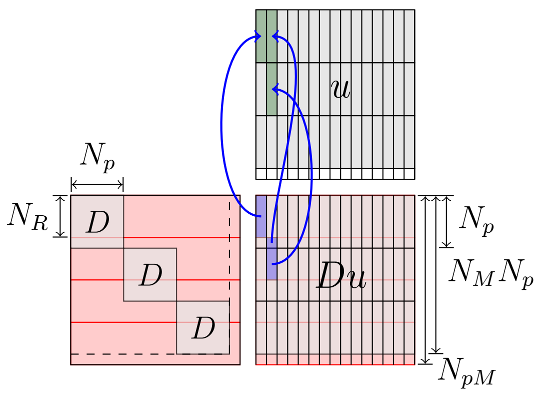


Figure 5.4. Row-wise segmentation of a microblocked matrix-matrix product. Element boundaries are shown in black, segment boundaries in red. Also shown: Fetch redundancy caused by segmentation. The second segment fetches field data from both the first *and* the second element because it overlaps rows from both.

necessary duplication for conflict-free access. For this layout, I consider a composite matrix made up of N_M vertically concatenated copies of the $D^{\partial\mu}$. This composite matrix is then segmented into pieces of N_R rows each, where N_R is chosen as a multiple of $T/2$. Each such matrix segment has a naturally corresponding range of degrees of freedom in a microblock, and I limit the thread block that loads this matrix segment to computing outputs from this range. Figure 5.4 illustrates the principle.

This computation layout makes the shared memory access conflict-free. Unfortunately, it also introduces a different, smaller drawback: there now is fetch redundancy. A segment needs to fetch field data for each element “touched” by its rows. This may lead it to fetch the same field values as the segment above and below it. Figure 5.4 gives an indication of this fetch redundancy, too. Fortunately, these duplicated accesses tend to happen in adjacent thread blocks and therefore possibly at the same time. I speculate that the L2 texture cache in the device can help reduce the resulting increased bandwidth demand.

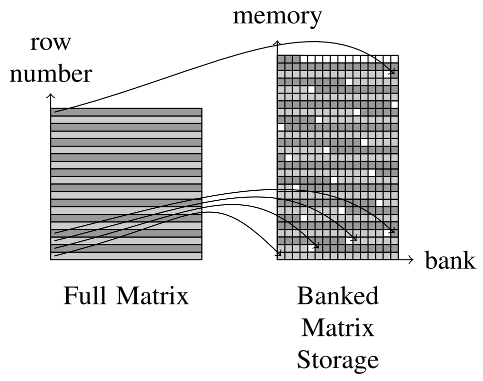
Next, observe that the matrix segments typically use less memory than the whole matrix. I can therefore reexamine the assertion that loading both matrix and fields into shared memory is not viable. Unfortunately, while the space to do so is now available, the field access bank conflicts from Section 5.3.2 spoil the idea.

One final observation is that for the typical choice of the reference element [Hesthaven and Warburton, 2007] the three differentiation matrices $D^{\partial\mu}$ are all similar to each other by a permutation matrix. Using this fact could allow for significant storage savings, but in my experiments, the added logic was too costly to make this trick worthwhile.

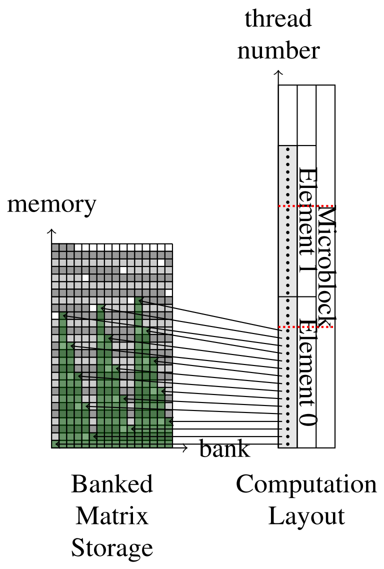
Algorithm 5.4 presents an overview of the techniques in this section. Instead of maintaining three separate local differentiation matrices, it works with one matrix in which the $D^{\partial\mu}$ are horizontally concatenated and then segmented. Shared memory limitations allow this algorithm to work at order six and below.

Algorithm 5.4 Local Differentiation with a segmented matrix in shared memory.

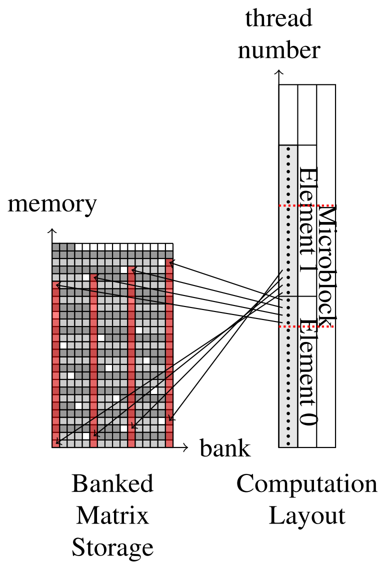
```
Require: A grid of $\lceil N_{pM}/N_R \rceil \times \lceil n_M/(w_p w_i w_s) \rceil$ blocks of size $N_R \times w_p \times 1$.
Require: Inputs: $\mathtt{u}^\mathtt{T}$, the field to be differentiated; $\mathtt{r}^\mathtt{T}$, the local-to-global differentiation
  coefficients.
Ensure: Output: $\mathtt{d}^\mathtt{G}_\nu$, the local $x, y, z$-derivatives of $\mathtt{u}^\mathtt{T}$.
  Allocate the differentiation matrix segment $\mathtt{D}^\mathtt{S} \in \mathbb{R}^{N_R \times (N_p d)}$ in shared memory.
  Load rows $[b_x N_R, b_x(N_R+1)\rangle \ (\mathrm{mod} N_p)$ of $[D^{\partial 1}, \dots, D^{\partial d}]$ into $\mathtt{D}^\mathtt{S}$.
  ———— Barrier+Memory Fence ————
  for all $\mathtt{s} \in [0, w_s)$ do
      $\mathtt{m} \leftarrow ((b_y w_s + \mathtt{s}) w_p + t_y) w_i$ { this thread's microblock number }
      $\mathtt{d}^i_\mu \leftarrow 0$ for $\mu \in \{1, \dots, d\}$ and $i \in [0, w_i\rangle$
      for all unrolled $n \in [0, N_p\rangle$ do
          $\mathtt{u}_{[0,w_i\rangle} \leftarrow \mathtt{u}^\mathtt{T}[(\mathtt{m} + [0, w_i\rangle) N_{pM} + n]$
          $\mathtt{d}^i_\mu \leftarrow \mathtt{d}^i_\mu + \mathtt{D}^\mathtt{S}[t_x, \mu N_p + n] \mathtt{u}_i$ for $\mu \in \{1, \dots, d\}$ and $i \in [0, w_i\rangle$
      end for
      $(\mathtt{d}^\mathtt{G})^{\mathtt{m}N_{pM} + [0,w_i\rangle N_{pM} + t_x}_{[0,d\rangle} \leftarrow \sum_\mu (\mathtt{r}^\mathtt{T})^{(\mathtt{m}+[0,w_i\rangle)K_M}_{[0,d\rangle d + \mu} \mathtt{d}^i_\mu$
  end for
```



(a) Assignment of matrix rows to memory banks. Alternating matrix rows are shown in two different shades of gray. They preserve their color as they move into individual 4-byte cells in the banked shared storage. Padding inserted to prevent conflicts is shown in white.



(b) Conflict-free access pattern in the first half-warp of the computation layout. The green highlighting illustrates that each of the 16 accesses lands in a unique bank.



(c) Conflicting access pattern in the second half-warp of the computation layout. The memory banks highlighted in red show 4 banks with two accesses each.

Figure 5.5. Local matrices and memory banks.

5.4 Experimental Results

In this section, I examine experimental results obtained from a DG solver for Maxwell's equations in three dimensions for linear, isotropic, and time-invariant materials. In terms of the electric field E, the magnetic field H, the charge density ρ, the current density j, the permittivity ϵ, and the permeability μ, they read

$$\epsilon\partial_t E - \nabla \times H = -j, \qquad \mu\partial_t H + \nabla \times E = 0, \tag{5.1}$$

$$\nabla \cdot (\epsilon E) = \rho, \qquad \nabla \cdot (\mu H) = 0. \tag{5.2}$$

One absorbs E and H into a single state vector

$$u := (E, H)^T = (E_x, E_y, E_z, H_x, H_y, H_z)^T.$$

If one defines

$$F(u) := \begin{bmatrix} 0 & -E_z & E_y & 0 & H_z & -H_y \\ E_z & 0 & -E_x & -H_z & 0 & H_x \\ -E_y & E_x & 0 & H_y & -H_x & 0 \end{bmatrix}^T,$$

(5.1) is equivalently expressed in conservation form as

$$\begin{bmatrix} \epsilon & 0 \\ 0 & \mu \end{bmatrix} u_t + \nabla \cdot F(u) = 0.$$

If the two equations (5.2) are satisfied in the initial condition, the equations (5.1) ensure that this continues to be the case. Remarkably, the same is also true (to the order of the scheme) of the DG discretization of the operator [Hesthaven and Warburton, 2002]. One may therefore assume a compliant initial condition and omit (5.2) from further discussion.

I label the numerical solution $u_N := (E_N, H_N)^T$ and choose the numerical flux F^* to be the upwind flux from [Mohammadian et al., 1991]:

$$\hat{n} \cdot (F_N - F_N^*) := \frac{1}{2} \begin{bmatrix} \{Z\}^{-1}\hat{n} \times (Z^+ [\![H_N]\!] - \hat{n} \times [\![E_N]\!]) \\ \{Y\}^{-1}\hat{n} \times (-Y^+ [\![E_N]\!] - \hat{n} \times [\![H_N]\!]) \end{bmatrix}.$$

I have employed the conventional notations for the cross-face average $\{u\} := (u_N^- + u_N^+)/2$ and jump $[\![u]\!] := u_N^+ - u_N^-$. For concise notation, I use the intrinsic impedance $Z := \sqrt{\mu/\epsilon}$ and admittance $Y := 1/Z$. Applying the principles of Section 2.1, I arrive at a discontinuous Galerkin scheme.

For my experiments, a solver using this scheme runs on an off-the-shelf Nvidia GTX 280 GPU with 1 GiB of memory using the Nvidia CUDA driver version 180.29. The GPU code was compiled using the Nvidia CUDA compiler version 2.1. At the time of this writing, GPUs of the same type as the one used in this test are sold for less than US$400.

I use a rectangular, perfectly conducting vacuum cavity (see [Jackson, 1998, Section 8.4]) excited by one of its eigenmodes to test the approximate solutions for accuracy. The solver works in single precision. L^2 errors observed for a sequence of grids at orders from one through nine are shown in Table 5.2. To better display the actual convergence of the method, the meshes examined were chosen to be rather coarse. Between the onset of asymptotic behavior and the saturation at the limits of single precision, the error exhibits the expected asymptotic behavior of h^{N+1} [Hesthaven and Warburton, 2002]. I observe that the solver recovers a significant part of the accuracy provided by IEEE 754 single precision floating point. It exhibited the same stability properties and CFL time step restrictions as a corresponding single- and double-precision CPU implementation. I have thus established that the discussed algorithm works and provides adequate solution accuracy if the discretization error falls above the threshold provided by the IEEE single precision

K	475	728	1187	1844	
N	$h = 0.3$	$h = 0.255$	$h = 0.21675$	$h = 0.184237$	EOC
1	$1.57 \cdot 10^{0}$	$1.19 \cdot 10^{0}$	$1.03 \cdot 10^{0}$	$6.46 \cdot 10^{-1}$	1.72
2	$4.15 \cdot 10^{-1}$	$2.84 \cdot 10^{-1}$	$1.82 \cdot 10^{-1}$	$1.19 \cdot 10^{-1}$	2.58
3	$1.61 \cdot 10^{-1}$	$9.44 \cdot 10^{-2}$	$5.56 \cdot 10^{-2}$	$2.80 \cdot 10^{-2}$	3.55
4	$4.75 \cdot 10^{-2}$	$2.52 \cdot 10^{-2}$	$1.13 \cdot 10^{-2}$	$5.03 \cdot 10^{-3}$	4.64
5	$1.54 \cdot 10^{-2}$	$6.37 \cdot 10^{-3}$	$2.55 \cdot 10^{-3}$	$9.03 \cdot 10^{-4}$	5.79
6	$3.84 \cdot 10^{-3}$	$1.42 \cdot 10^{-3}$	$4.42 \cdot 10^{-4}$	$1.32 \cdot 10^{-4}$	6.94
7	$9.89 \cdot 10^{-4}$	$2.77 \cdot 10^{-4}$	$7.36 \cdot 10^{-5}$	$1.77 \cdot 10^{-5}$	8.24
8	$1.91 \cdot 10^{-4}$	$4.76 \cdot 10^{-5}$	$1.05 \cdot 10^{-5}$	$2.55 \cdot 10^{-6}$	8.90
9	$4.25 \cdot 10^{-5}$	$8.71 \cdot 10^{-6}$	$2.10 \cdot 10^{-6}$	$1.30 \cdot 10^{-6}$	7.31

Table 5.2. L^2 errors and empirical orders of convergence (EOC) obtained by a solver for Maxwell's equations on an Nvidia GTX 280 running in single precision, at a variety of orders and for a number of rather coarse meshes.

floating point representation.

The reason for bringing DG onto a GPU was however not to show that it works there, but to show that it can be made to work extremely fast. Figure 5.6(a) portrays the speed of my solver in comparison with a CPU implementation running on a single core of a 3 GHz Intel Core2 Duo E8400 CPU, also running in single precision.

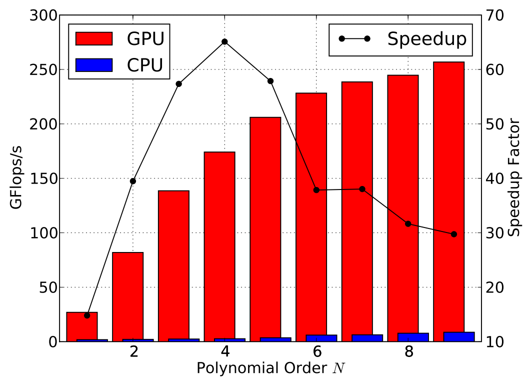
The CPU calculations are based on the solver "hedge" (see Chapter 3). The code generated by hedge is compiled on the fly using gcc 4.3.2, with optimization enabled (`-O3 -march=native -mtune=native -ftree-vectorize`). For element-local parts of the operator, the Python code adaptively chooses between a fully unrolled, machine-generated matrix multiplication kernel and a BLAS-based version that targets ATLAS 3.8.2 [Whaley et al., 2001]. The machine-generated matrix multiplication code does not explicitly use SSE or other vector instruction intrinsics, but such instructions may well be used by the compiler.

Unless otherwise specified, all performance numbers are based on the wall clock time from the beginning of one time step to the beginning of the next, including RK4 time stepping. Timings were averaged over a run of 100 (CPU) or several hundred (GPU) time steps to minimize the influence of timing transients. Timings were observed to be consistent across runs.

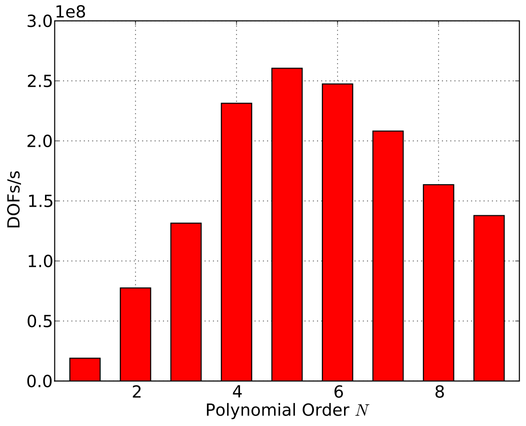
The GPU reaches a peak floating point throughput of more than 250 GFlops/s at polynomial order nine, and more than half this value at orders three and above, with a rapid increase between order $N = 1$ and $N = 5$, and a performance plateau of more than 200 GFlops/s at orders $N = 6$ and above. While I would like to argue that these immediate performance numbers in units of Flops/s are perhaps the most meaningful measure of performance and are sufficient on their own, it is natural to ask for the performance gain compared to prior code on the CPU. While such numbers are certainly helpful in putting results into perspective, the comparison is dangerous, because it introduces another variable–the quality of the CPU code–into the measurements that has no bearing on the quality of the GPU results, and they should be viewed with suspicion.

Many of my results contain CPU performance data and speed-up numbers. Above, I have tried to specify quite carefully how the CPU numbers were obtained. Regardless, I would like to remark that even for the (rather optimistic) prediction that a more highly tuned CPU-specific implementation might run twice or three times as fast as mine, the GPU maintains an advantage well above an order of magnitude, and this is the important message of this chapter.

In more detail, the GPU outperforms the CPU by factors ranging from 14 to 65. At the practically relevant orders of three and four, the speedup factors are 57 and 65, respectively. It is rather fortunate, but not entirely a coincidence that these two orders are not only the ones that see most practical use, they also exhibit some of the largest speedup factors on the GPU.



(a) Discontinuous Galerkin performance in GFlop-s/s on a GPU and a CPU. Computations were performed in single precision.

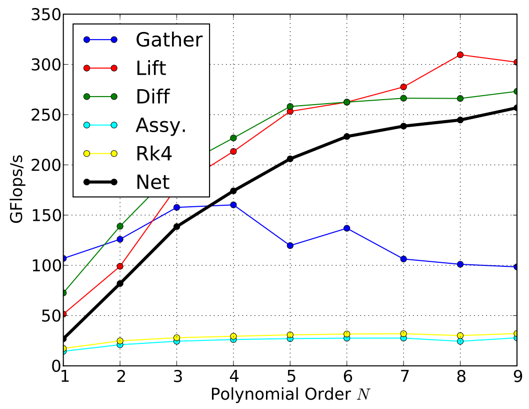


(b) Number of degrees of freedom to which my methods can apply the Maxwell operator in one second. Assuming linear scaling, this graph can be used to determine run times for larger and smaller problems. DOFs from each of the six Maxwell fields are counted separately.

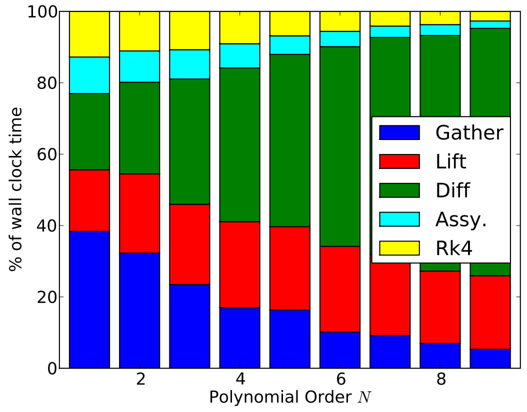
Figure 5.6. Performance characteristics of DG on Nvidia graphics hardware.

Orders three and four are particularly favorable not only for their appreciable speedups and their moderate time step requirements [Warburton and Hagstrom, 2008]. They also achieve the peak nodal value throughputs on the GPU as shown in Figure 5.6(b). Naturally, high-order approximations of solutions to partial differential equations contain much more information per DOF than do solutions obtained via low order methods. This is most apparent in the number of DOFs required to accurately represent one wavelength [Hesthaven et al., 2007]. Interestingly, I observe that despite lower computational load, the DG methods of orders one and two achieve lower overall throughput than the next higher ones, a likely result of a mismatch with the hardware's granularities. This crossover between granularity effects and the increase in floating point work with growing N makes DG methods of orders three through five the fastest DG methods on a GPU even on a per-DOF basis.

Recall now that I have split the DG operator into several parts, each of which performs distinct kinds of processing and tends to require a different strategy to map onto a GPU. It



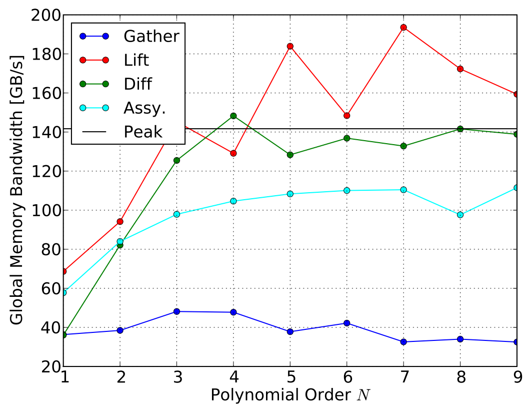
(a) Compute bandwidth in GFlops/s achieved by each part of the DG operator, at various polynomial orders. The published theoretical peak floating point performance for the hardware on which these tests were run is 933 GFlops/s [Various authors, 2008].



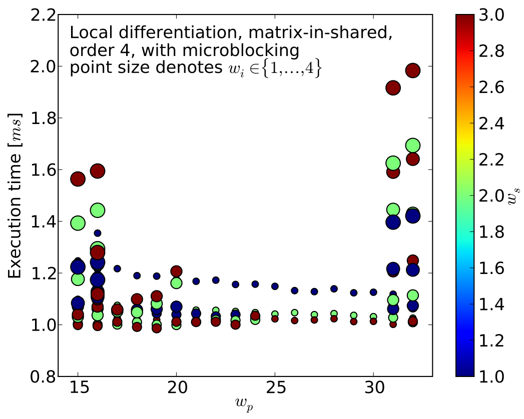
(b) Percentage of time spent in various parts of the DG operator vs. polynomial order.

Figure 5.7. Performance characteristics of DG on Nvidia graphics hardware, continued.

is therefore interesting to see what performance level is attained by each part of the operator. Figure 5.7(a) gives an indication of this performance, based again on the number of floating point operations per second. Here and wherever GPU performance is broken down by component, timings were obtained using the `cuEventElapsedTime()` call. It is reassuring that, despite different implementation strategies, the flop rates for element-local differentiation and lifting evolve almost identically as the order N is increased. These two parts of the operator are also characterized by the highest arithmetic intensity and the most regular access pattern. As an unsurprising consequence, they are able to realize the greatest performance gain as the order of the operator and therefore the access granularity grows. The flux gather, on the other hand, realizes its greatest performance at orders three and four. I suspect that the decline in performance with increasing N can be attributed to the growth of the indirect indexing information in the form of face index lists $\mathtt{J}^{\mathrm{T}}$ from Algorithm 5.3. These lists are referenced constantly throughout the whole algorithm and are therefore likely to reside in the texture cache, of which there are only a few KiB per multiprocessor. As these lists grow, their cache eviction likelihood also grows, resulting in reduced access



(a) Memory bandwidths in GB/s achieved by each part of the DG operator. The peak memory bandwidth published by the manufacturer is 141.7 GB/s. Values exceeding peak bandwidth are believed to be due to the presence of a texture cache.



(b) Sample work distribution parameter study for local differentiation on fourth-order elements with microblocking enabled.

Figure 5.8. Performance characteristics of DG on Nvidia graphics hardware, continued.

bandwidth. In addition to the above-mentioned main parts of the operator, the figure also shows performance data for the assembly of the operator and the fourth-order low-storage Runge Kutta time stepper [Carpenter and Kennedy, 1994]. Both of these operations perform linear combinations of vectors, making them much less arithmetically intense than the element-local operations. Fortunately, as the order N increases, the processing time spent in element-local operations dominates and helps decrease the influence of the latter three operations on overall performance. Figure 5.7(b) reinforces this point.

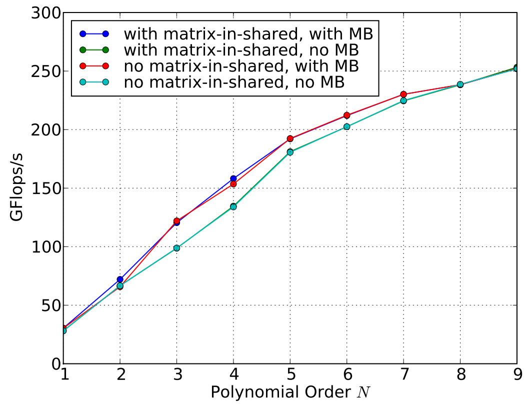
It is interesting to correlate the achieved floating point bandwidth of each component from Figure 5.7(a) with the bandwidth reached for transfers between the processing core and global memory, shown in Figure 5.8(a). I have obtained these numbers by counting the number of bytes fetched from global memory either directly or through a texture unit. The published theoretical peak memory bandwidth is 141.7 GB/s [Various authors, 2008], shown as a black horizontal line. Perhaps the most striking feature here is that the calculated memory bandwidth sometimes transcends this theoretical peak. I attribute this phenomenon to the presence of various levels of texture cache. Its occurrence is especially pronounced in the case of flux lifting, and it should perhaps be sobering that the other parts of the DG operator do not manage the same feat. In any case, flux lifting uses the fields-in-shared strategy, and therefore fetches and re-fetches the rather small matrix L, making large amounts of data reuse a plausible proposition. Aside from this surprising behavior of flux lifting, it is both interesting and encouraging to see how close to peak the memory bandwidth for element-local differentiation gets. As a converse to the above, this makes it likely that the operation does not get much use out of the texture cache in most situations. It does imply, however, that rather impressive work was done by Nvidia's hardware designers: The theoretical peak global memory bandwidth can very nearly be attained in real-world computations. Next, taking into account what was said in Section 5.3.2 about the flux-gather part of the operator, the rather low memory throughput achieved is not too surprising–the access pattern is (and, for a general grid, has to be) rather scattered, decreasing the achievable bandwidth. Lastly, operator assembly, which computes linear combination of vectors, consists mainly of global memory fetches and stores. It seems likely that ancillary operations such as index calculations, loop overhead and bounds checks drive this component's shortfall from peak memory bandwidth.

I would further like to remark that Figure 5.8(a) indicates that the element-local matrix parts of the calculation are memory- rather than compute-bound. This should not be the case–by the nature of the workload, there is much very local floating point work to be performed, which should suffice to mitigate the dependency on memory bandwidth. Unfortunately, despite the sophisticated tiling and loop splitting techniques discussed earlier, it appears that the on-chip memory on the device does not suffice to sufficiently exploit this locality. This phenomenon is also observed in the vendor's high-performance BLAS implementation, whose SGEMM (single-precision matrix-matrix multiplication) routine reaches very similar performance levels as my element-local matrix routines, albeit only on much larger matrices.

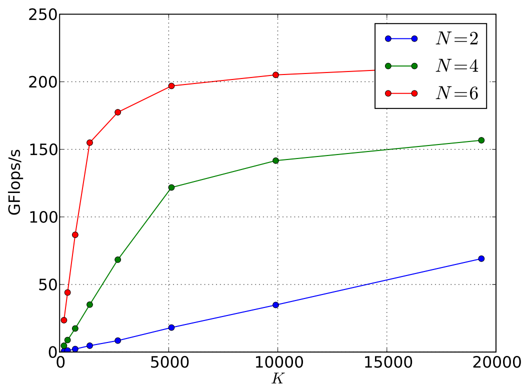
		differentiation				flux gather		flux lifting			
N	K_M	Shared	w_p	w_i	w_s	M_B	w_p	Shared	w_p	w_i	w_s
1	4	Matrix	15	2	2	2	16	Field	3	3	1
2	8	Matrix	21	1	3	1	17	Field	3	3	1
3	4	Matrix	21	1	3	1	8	Field	2	3	1
4	4	Matrix	19	2	3	1	15	Field	2	4	1
5	2	Field	1	4	1	1	9	Field	2	3	1
6	2	Field	1	4	1	1	8	Field	2	4	1
7	2	Field	2	4	1	1	5	Field	2	3	1
8	1	Field	2	4	1	1	2	Field	2	4	1
9	1	Field	2	4	1	1	3	Field	2	4	1

Table 5.3. Empirically optimal method parameters for each part of the DG operator at polynomial orders 1 through 9.

For potential implementers, it may be interesting to know which exact parameters were used to obtain the results in this section. The parameters of interest include the generic work distribution tuple (w_p, w_i, w_s) for each subtask, the microblock size K_M, the gather block size M_B, and which of the matrix- or field-in-shared approaches was used at what order. Table 5.3 presents this data. It is peculiar how little regularity there is in this data set. Despite a sequence of attempts, I have failed to come up with a heuristic that would predict performance accurately. This led me to develop an empirical optimization procedure that finds the data of Table 5.3 in an automated fashion through a sequence of synthetic and real-world benchmarks. A detailed study of the toolkit I have constructed to enable them was presented in Chapter 4. At this moment, I will restrict myself to displaying the results of one such procedure. Figure 5.8(b) displays the run time obtained for element-local differentiation employing microblocking and the matrix-in-shared strategy at order $N = 4$. The objective is to find the work distribution parameter tuple (w_p, w_i, w_s) that leads to an empirically short run time for this part of the operator. It should be stressed that all runs depicted in the figure perform the same amount of work. From Table 5.3 one can see that in this particular instance, an optimum was found at $(w_p, w_i, w_s) = (19, 2, 3)$. Undoubtedly, with better knowledge of the hardware, many of the odd-looking ups and downs in Figure



(a) Performance in GFlops/s achieved at various polynomial orders, for different simplified implementations of DG on CUDA.



(b) Mesh-dependent scaling of discontinuous Galerkin on Nvidia GPUs.

Figure 5.9. Performance characteristics of DG on Nvidia graphics hardware, continued.

5.8(b) could be understood. Given the published documentation however, one is mostly left to take the results at face value. Luckily, if one were to randomly choose a configuration from the portrayed set, in all likelihood the resulting operation would at most take about 20 per cent longer than the optimal one chosen here. On the other hand, with some bad luck one may also encounter a configuration that makes the computation take about twice as long.

From Table 5.3 one can also gather that the field-in-shared strategy with a wide variety of work distribution parameters is found to deliver the best performance at all orders for flux lifting, as well as for higher-order element-local differentiation. This is plausible behavior and was already discussed in Section 5.3.4. It is therefore reasonable to ask what would be lost if the matrix-in-shared approach were omitted from a GPU DG implementation entirely. Also, the introduction of microblocks into the method brings about some mild complications, particularly in the form of shared memory bank conflicts, so one may be compelled to ask how much is lost by ignoring microblocks and simply padding each element to the nearest alignment boundary. The remaining performance after restricting my implementation to not use one or both of these optimizations can be seen in Figure 5.9(a). Examination of this figure leads to the conclusion that the work of implementing a matrix-in-shared strategy is likely only worthwhile if one is particularly interested in running GPU-DG at a few specific low orders. The benefit of employing mircoblocking, on the other hand, is pervasive and fairly substantial. It stretches to far higher orders than one might suspect at first, given the growth of the involved operands.

Note that these conclusions apply only to the algorithms exactly as described so far. If even one simple trick is omitted from an implementation, trade-offs may shift dramatically. For example, omitting the thread ordering trick from Section 5.3.2 makes a matrix-in-shared strategy optimal for differentiation up to order six.

I would like to note that the performance results in this section depend on the size of the problem being worked on. A very small problem may, for example, not offer enough opportunity to properly occupy all the processing cores that the hardware provides. Figure 5.9(b) reveals that even relatively small problems achieve decent performance. In addition, I observe that this scaling effect is apparently not just governed by the number of elements present, but also by the order N, which influences the number of flops per DOF in the method. I conclude that as soon as there is a certain amount of floating point work to be done per time step, performance will be as expected. Further on the topic of problem size, I note that the size of the largest possible simulation is limited only by the amount of available memory. Following Figure 5.1(a), taking into account the worst-case padding overhead of 5 per cent and knowing how many scalar fields one needs for storage (30-40 is a reasonable number for a Maxwell solver), one may easily calculate the number of elements available on a given GPU. As an example following these guidelines, each gigabyte of GPU memory translates into about 200k elements at $N = 4$.

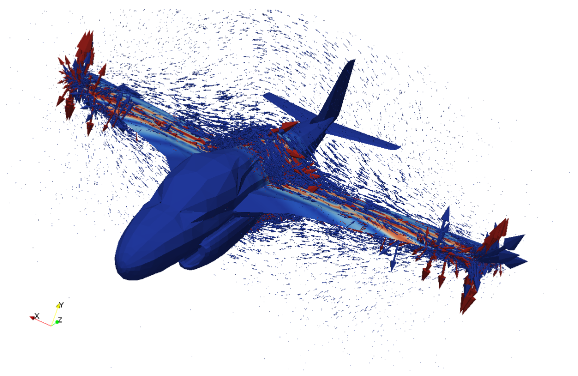
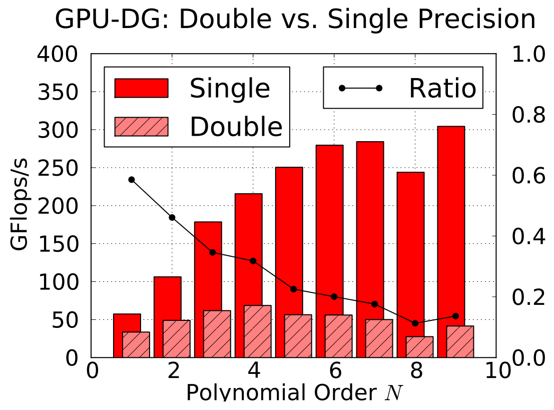


Figure 5.10. A sample scattering problem solved using the methods described in the text. The incident plane-wave electric field is shown as pseudo-color values on the scatterer, while the scattered electric field is shown as arrows. The computation was performed at order $N = 4$ on a mesh of $K = 78745$ elements using an incident-field formulation [Hesthaven and Warburton, 2002] and characteristic absorbing boundary conditions. It achieved and sustained more than 160 GFlops/s.

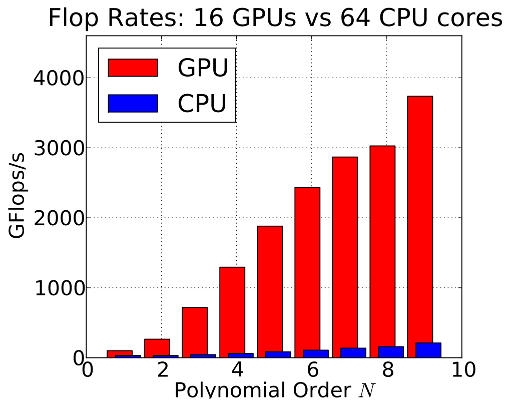
5.4.1 Further Results: Double Precision, Distributed Computation

A common charge leveled against GPU computing is that it is in some sense “a toy” or “not fit for ‘serious’ use”. Two of the cornerstones of this type of argument is the perceived lack of full support of calculation in IEEE double precision, and the other is the perceived impossibility to to treat big problems.

The purpose of this section is to present results refuting both arguments. If one only looks at the hardware specifications, it is easy to be misled to the conclusion that IEEE double precision is not a first-class citizen on a GPU: On Nvidia hardware that was current as of this writing, there are eight times fewer hardware units for performing double precision calculations than there are for single precision, which might lead one to conclude that performance would decline by a factor of eight. First, this factor of eight will be much reduced on future hardware, and second, the situation is not as dire even on present hardware, as Figure 5.11(a) shows. The data in the figure were obtained on the same Maxwell eigenmode test as above, but on an Nvidia GTX295 using a newer version of the



(a) Discontinuous Galerkin performance in GFlops/s vs polynomial order N on an Nvidia GTX295 GPU in single and double precision.

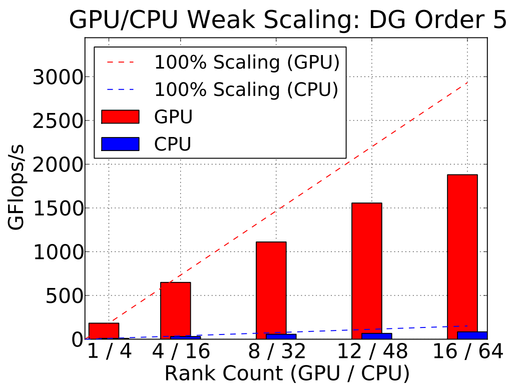


(b) Discontinuous Galerkin performance in GFlops/s vs polynomial order N on 16 GPUs, compared to 16 8-core CPUs. The computation was performed in single precision.

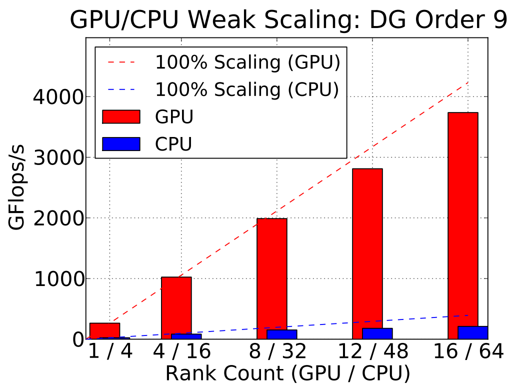
Figure 5.11. Performance of Maxwell-GPU-DG in double precision and on a parallel machine.

solver, with both factors contributing to a slight performance increase over Figure 5.6(a).

The figure shows a performance decrease from single precision to double precision of a factor between three and five, roughly, with a somewhat significant, and so far unexplained, drop at order $N = 8$. It is further remarkable that up to order $N = 4$, only a performance reduction by a factor of three is encountered. Considering Figure 5.8(a), it is not hard to explain why the full impact of the reduction in floating point hardware from single to double precision is not felt: If the computation were compute-bound, i.e. waiting for floating-point results, the performance decrease would match the decrease in hardware. As was remarked above, however, this is not the case, as significant parts of the computation are memory-bound. And since IEEE double-precision floating point numbers require only twice the amount of storage of single-precision ones, memory-bound parts of the computation should only suffer a performance penalty of a factor of two. All of the observed values for the SP/DP performance ratio are found within this range, therefore doubling as a convenient indicator of how memory- or compute-bound a certain my GPU-DG implementation is at a given order N.



(a) GPU and CPU Weak Scaling at $N = 5$.



(b) GPU and CPU Weak Scaling at $N = 9$.

Figure 5.12. Parallel scaling for distributed-memory Maxwell-GPU-DG.

In summary, while the observed performance in double precision is certainly not as impressive as in single precision, some of the Flops/s rates achieved are still beyond the *peak* (not sustained!) values of current CPUs. Therefore, performing a computation in double precision is a practical option on a GPU. However precision should be carefully chosen to make sense for a given computation. If the discretization error is greater than the precision impact of single precision, performing such a calculation in double precision would be quite unreasonable.

Next, I would like to discuss some performance results that I have obtained from for GPU-DG on a 16-GPU cluster. Throughout this chapter, it was observed that DG is very suitable to the fine-grained, shared-memory parallelism of a GPU. It is further known that DG also adapts well to the large-scale distributed-memory setting [Fischer et al., 2008]. Here, I am exploring how well DG adapts to a mixture of the two. One might imagine that it is a disadvantage that communication between GPUs involves three data transfers (GPU 1 to host 1, host 1 to host 2, host 2 to GPU 2), and hence more communication latency and perhaps reduced bandwidth when compared to direct, host-to-host communication.

Luckily it appears that, if this penalty exists, it is not very large. Figure 5.11(b) shows performance results for 16 T10 GPUs as found in Nvidia S1070-500 rack-mount GPU computing systems, which were attached to eight dual-quad-core Intel Xeon E5472 hosts running at 3 GHz. CPU results were obtained on the same machine, with one process per core. The same caveats as above about comparability of CPU and GPU results hold here, and two further issues arise hampering direct comparability. First, in a CPU, one should consider that all CPU cores share the off-chip memory interface, and hence its memory bandwidth, presenting a natural scaling impediment. Second, in this comparison, the GPU-based parallel computation had four times fewer (MPI) ranks than the CPU computation. Therefore the communication needs of both computations are not directly comparable.

Even taking these comparability concerns into account, it appears that the CPU-GPU performance gap remains larger than an order of magnitude. It is remarkable that the computation at $N = 9$ achieves nearly four Teraflops/s of sustained application compute throughput. Just a few short years ago, such numbers were firmly the territory of very large supercomputer installations, but they have here been achieved using (comparatively) very affordable hardware.

It is further interesting to review the scaling properties of the implementation as GPUs are progressively added. Figure 5.12 displays results for weak scaling (i.e. the problem size grows with the number of machines) of GPU-DG at polynomial degrees $N = 5$ and $N = 9$. It is clearly visible that at the lower order, where each element contains a larger fraction of facial degrees of freedom, and hence communication needs are greater, only about 60 per cent of the theoretical, perfectly-scaled throughput (indicated by the dashed line) is achieved. The situation markedly improves at $N = 9$, reaching the high performance results discussed above.

5.5 Conclusions

In this chapter, I have described and evaluated a variety of techniques for performing discontinuous Galerkin simulations on recent Nvidia graphics processors. I began this work by adapting a pre-existing DG code for the GPU, enabling a thorough comparison of strategies for mapping the method onto the hardware. After that, I wrote a final code that combined the insights gained from its precursor and colleagues' research. This code implements the strategies of Sections 5.2 and 5.3 and was used to obtain the results in Section 5.4.

I have shown that, using my strategies, high-order DG methods can reach double-digit percentages of published theoretical peak performance values for the hardware under consideration. DG computations on GPUs see speed-up factors just short of two orders of magnitude. This speed increase translates directly into an increase of the size of the problem that can be treated using these methods. A single compute device can now do work that previously required a roomful of computing hardware. Alternatively, a cluster of machines equipped with these cards can run simulations that were previously outside the reach of all but the largest supercomputers. This lets the size and complexity of simulations that researchers can afford on a given hardware budget jump significantly.

To make these speed gains accessible, I have provided detailed advice for potential implementers who need to achieve a trade-off between computing performance and implementation effort. The data provided in Section 5.4 will help them make informed implementation decisions by allowing them to predict the computational speed achieved by their implementations.

Many-core computing presents a rare opportunity, and I feel that discontinuous Galerkin methods have a number of unique characteristics that make them unusually suitable for many-core platforms. In the past, users have chosen low-order methods because of the perceived expense involved in running simulations at a high order of accuracy. While this perception was questionable even in the past, I feel that many-core architectures disproportionately *favor* high order and significantly drive down its relative cost. Moreover, unlike most other numerical schemes for solving partial differential equations, DG methods make the order of accuracy a tunable parameter. These factors combine to give the user a maximum of control over both performance and accuracy.

CHAPTER SIX

Viscous Shock Capturing in a Time-Explicit Discontinuous Galerkin Method

6.1 Introduction

In the previous chapter, it was seen that graphics processors can accelerate DG solvers for linear systems of hyperbolic conservation laws by a significant factor of more than an order of magnitude. Given this advance, it is a tempting and rather obvious extension to ask what the same technology can do for nonlinear systems. If the solution of the system remains smooth for the entire time under consideration, and if thereby the decay of modal coefficients is fast enough, the method from the previous chapter be used as-is for a so-called “nodal approach”. Optionally, aliasing error in the computation of integrals for stiffness and mass matrices can be avoided by the introduction of quadrature schemes of sufficient order [Hesthaven and Warburton, 2007].

If however the solution does not stay smooth for long enough periods of time, the arising discontinuities pose a number of problems which have been the subject of intense study since the early days of scientific computation and numerical analysis. The most grave such problems are *Gibbs phenomena*, which manifest themselves as an unphysical oscillation near a discontinuity. Gibbs phenomena were first observed in the context of Fourier expansions, but occur just as much in the polynomial spaces I am employing here. The phenomenon can lead to many undesirable effects such as the occurrence of negative values for inherently positive quantities (such as density or pressure) or the premature crossing of thresholds in systems with strong nonlinearities. A vast body of literature on this subject of *shock capturing* exists, and it is not my goal here to give a full overview of the approaches that have been tried. Instead, my goal here is to seek out, based on immediately related literature, a method that is able to control the occurrence of Gibbs phenomena in the context of the discontinuous Galerkin method (as introduced in Section 2.1 and discussed in other previous chapters). In doing so, I will engineer the method to be suited to leveraging the graphics processor-based solver technology presented in the previous chapter.

I choose to base my approach to the problem on *artificial viscosity*, a purposefully introduced, carefully designed, and entirely unphysical diffusion term whose sole purpose it is to selectively damp out high frequency solution components encountered wherever Gibbs phenomena are present. The technique itself is based on the smoothing character of diffusive processes, and thereby obvious enough. It dates back to von Neumann and Richtmyer [1950] and was, as most numerical techniques, first used in the context of finite difference methods [Lapidus, 1967], and then spread into finite element literature (see, e.g., the study by John and Schmeyer [2008] for a review) and was also applied to time-dependent discontinuous Galerkin methods very early on [Bassi and Rebay, 1994]. Within the DG community, the method has enjoyed continuing popularity [e.g. Burman, 2007].

There has been a recent resurgence of interest in the method based on publications by researchers at the Aerospace Computational Design Laboratory at MIT [Barter and Darmofal, 2010, Persson and Peraire, 2006]. The methods in this chapter aim to improve on these latter schemes and make them suitable for a GPU-DG setting. As I justify the construction of my methods in Section 6.4, I will provide further context and comparison about the methods cited in this paragraph.

Many more authors have proposed methods to capture shocks within a high-order discontinuous Galerkin setting, by different methods. *Flux limiting*, which has been both successful and popular with Finite Volume practitioners, was combined with DG immediately in conjunction with the resurgence of interest in the method in the late 1980s. A vast body of literature has emerged that proposes a large variety of limiters for use with DG methods, and nearly every method that has enjoyed success in a Finite Volume setting has been tried with DG, ranging from early TVB limiters [Cockburn and Shu, 1989, 1998, Cockburn et al., 1989, 1990], through a variety of more recent developments [Burbeau et al., 2001, Dolejsí et al., 2003, Krivodonova, 2007, Kuzmin et al., 2005, Tu and Aliabadi, 2005, Xu et al., 2009]. A common theme to limiting is that the solution is modified in some way to retain desirable properties such as positivity and freedom from spurious oscillation, and in doing so, reaches various (often low) orders of accuracy.

Although limiting has been tremendously successful and prevalent in the literature, I am suspicious of the–to my mind–often somewhat brutal modifications to the approximate solution performed by limiters, and I prefer the simplicity of artificial viscosity methods. These methods take the position that the only hope of resolving a discontinuity by a high-order approximation lies in smoothing it out. The method of *Spectrally Vanishing Viscosity* [e.g. Kirby and Sherwin, 2006, Tadmor, 1989] is similar in spirit, but tries to restrict its smoothing action to high-frequency solution components.

One final, if expensive, approach of dealing with discontinuities is that of adapting the mesh and adding resolution. It is generally thought that 'shocks', i.e. actual discontinuities, do not exist in nature [Woodward and Colella, 1984], and thereby, if only enough resolution were available, the problem of shock capturing would vanish by itself. While nature may obey this statement, mathematical models of it often do not, and so one needs to "help a little"–for example by adding an artificial viscosity [e.g. Hartmann, 2006]. Further, while adaptivity certainly is a useful technique in capturing shocks [Flaherty et al., 1997, Kirby et al., 2000, Warburton et al., 1999, Xu et al., 2010], it depends on a detector that reliably tells the method where refinement is necessary. If this detector is just a bit late in detecting oscillation or underresolved discontinuities, adaptivity by itself is unlikely to be able to salvage the solution.

It has been noticed that many methods have been proposed which "perform well when applied to one-dimensional flow problems but which encounter major difficulties in two dimensions." [Woodward and Colella, 1984] Since Finite Volume methods solve an assembly of essentially one-dimensional discontinuous interface problems (i.e. *Riemann problems* [Toro, 2009]), they manage to retain a one-dimensional character, even in multiple dimensions. The component enabling this is the representation of the solution by cell averages. Conversely, as soon as significant element-local structure (such as local polynomial spaces in DG) is present, the transition to two and more dimensions can be particularly treacherous. To help avoid falling into this trap, I will aim to base my method only on concepts which have a simple generalization to multiple dimensions. Ambiguities arising in this generalization are discussed in Section 6.4.3.

In constructing my method, I will proceed as follows: I will begin in Section 6.2 by explaining a few basic design considerations for the method, in particular in relation to time integration. In Section 6.3, I will give a brief overview of the hyperbolic conservation laws that I am targeting, and whose solution theory allows the existence or emergence of discontinuities and shocks. In this section I will also clarify how the artificial viscosity term is added to each of the conservation laws, in each case depending on a parameter ν. It is of course not wise to use a homogeneous, non-zero viscosity ν all across the solution domain, as this would unduly diffuse even smooth (and well-resolved) solution features. One therefore needs a detector whose output is a spatially dependent measure of smoothness s that alerts one to those areas where under-resolution and oscillation are occurring. The careful construction of a robust detector of this kind (and its justification) is the main contribution of this chapter, to be found in Section 6.4. The subsequent Section 6.5 explains how measured smoothness may be turned into a space-dependent viscosity parameter $\nu(x)$. Section 6.6 then represents my attempt to convince the reader that the detector and the viscosity generator work as designed, through a comprehensive series of tests of increasing complexity. Finally in Section 6.7, I will comment on what was achieved, what remains to be done, and further point out directions for future investigation.

6.2 Basic Design Considerations

I have already stated that, as in the previous chapter, I am targeting massively parallel throughput-oriented computer architectures with the shock-capturing scheme that this chapter describes. I have described a method to quickly compute the vector $A(x)$ for a (then linear) discontinuous Galerkin operator A and a state vector x using graphics hardware.

On wide-SIMD, parallel architectures such as those of the previous chapter, where memory is at a premium and scattered memory access is particularly expensive, such *matrix-free* methods, if they can be implemented efficiently, will always hold a significant performance advantage over approaches that have to build, keep in memory, and constantly access a pre-built sparse matrix, because such a computation is necessarily bound by the speed at which matrix entries can be streamed into the core, where they are then used exactly once and discarded. [Bell and Garland, 2008] A matrix-free approach, as shown, has far more freedom to exploit local structure and re-use data. I will therefore focus my investigation on matrix-free methods.

This choice has important ramifications. One consequence of it affects the trade-off by which one chooses between implicit and explicit time stepping. Consider the case of implicit time integrators, in which one must constantly solve large linear systems of equations. Direct, factoring solvers for sparse matrices are as yet unavailable on massively parallel hardware, and even if they were, they would doubly suffer from the issues that sparse matrices encounter. One therefore naturally looks towards iterative methods for solving large sparse systems. For the complicated linearized systems arising from the nonlinear hyperbolic conservation laws I am targeting in this chapter, these methods generally need help in the form of a preconditioner in order to be efficient. This is the next implication of the choice of matrix-free methods: One automatically chooses to not use the substantial body of literature showing how a preconditioner may be built from a known sparse matrix. Instead, one needs to invest further work designing and testing preconditioner (using e.g. multi-grid or domain-decomposition methods), and, in addition to the design time spent, these preconditioners may carry significant additional computational expense, typically through their communication needs. Their suitability for massively parallel computer architectures is as yet undetermined. In addition, Krylov methods in particular involve global reductions (in the form of inner products) which are known to not achieve peak performance on graphics processors [Harris, 2007]. Worse, the nonlinear systems I am targeting in this chapter require a nonlinear system to be solved (likely by Newton iteration, which in turn requires Jacobians to be evaluated).

This collection of drawbacks and uncertainties in the application of implicit time integration on massively parallel hardware makes it seem opportune to examine the use of explicit time steppers, which were already used with good success in the previous chapter, with the goal of finding out if the single big disadvantage of explicit methods, namely their small time step restriction, can be offset by the judicious choice of methods combined with the advantages conferred by the hardware.

Since the scheme I am aiming to design involves the use of artificial viscosity, the scaling of the explicit time step is typically given by

$$\Delta t \sim \frac{1}{\lambda_{\text{max}} \dfrac{N^2}{h} + \|\nu\|_{L^\infty} \dfrac{N^4}{h^2}}, \tag{6.1}$$

where λ_{max} is the largest characteristic velocity and ν is the magnitude of the viscosity, h is the local mesh size and N is the approximation's polynomial degree [Hesthaven and Warburton, 2007]. Within (6.1), the numerical diffusion time scale $N^4\|\nu\|_{L^\infty}/h^2)$ can be rather damaging, as it contains mesh-dependent factors at high exponents.

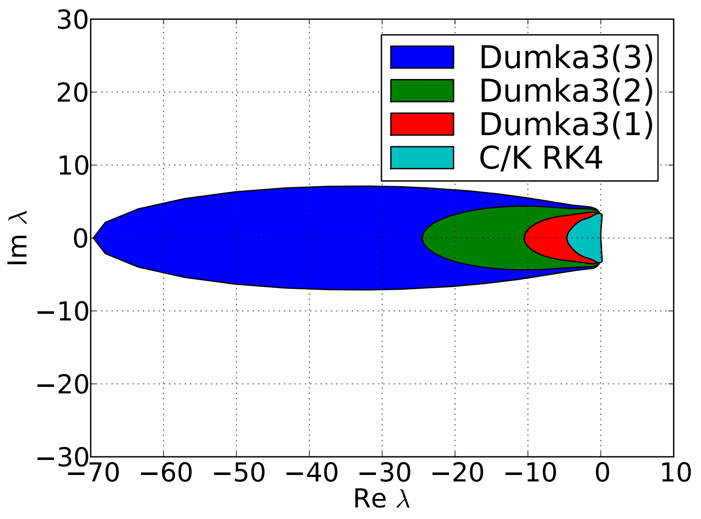


Figure 6.1. Stability regions of various DUMKA3 time integrators Medovikov [1998] and a fourth-order low-storage Runge-Kutta method by Carpenter and Kennedy [1994]. As in Chapter 8, stability information was obtained by the power method applied to the time stepper for the test system $\dot{y} = \lambda y$.

Luckily, (6.1) does not tell the entire story. First of all, I expect the occurrences of high viscosity ν to be localized in both space and time. Spatial localization could conceivably be dealt with using local time stepping (cf. Chapter 8). Temporal localization is easily dealt with by the use of adaptation in time [e.g. Dormand and Prince, 1980]. Adaptivity in time is particularly important for explicit time stepping of artificial-viscosity-enhanced PDE solvers. While some points to the contrary are made below, a detected shock and the resulting spike in viscosity do change the time step restriction of the method. Perhaps the temporal variation in the time step requirement is not quite as drastic as (6.1) might suggest, however there may be solution events requiring very small time steps. In my experience, these events are relatively rare, and therefore it would be a tremendous waste to only ever make progress at the smallest Δt required throughout the entire computation. Section 6.6 will further support this point with empirical observations.

(6.1) embodies a tacit assumption, namely that the stability region of the time stepping method has the same extent in both the imaginary direction, (roughly) responsible for the convective scale $h/(\lambda_{\max} N^2)$, and the negative real direction, responsible for the diffusive time scale. This is not necessarily true, and Runge-Kutta-Chebyshev-type methods have been designed, for example by Medovikov [1998], whose stability region grows along the negative real direction proportional to the square of the number of their stages . Figure 6.1 on the preceding page shows an example of such stability regions, compared to that of a "conventional" equal-aspect Runge-Kutta method [Carpenter and Kennedy, 1994]. I am therefore confident that the smallness of the time step due to viscosity can be controlled.

One further aspect of the time discretization should be considered: Much of the effort in this chapter is targeted at mitigating the effect of oscillations in the spatial discretization of a conservation law that trace their roots back to the polynomial expansions used for them. Time discretizations, however, are equally based on polynomials, and a total-variation-diminishing (TVD) family of time steppers has been developed to mitigate oscillations caused by them [Shu, 1988]. Since in the case of this chapter, the need for adaptivity in time is greater than perfect control of oscillation, which I deem not achievable just through the use of artificial viscosity, I am forgoing the use of TVD time discretizations for now, but I would like to remark that an embedded Runge-Kutta method, whose higher-order component is TVD, would be likely be the most appropriate choice if it were available.

In summary, the emergence of massively parallel hardware along with the use of non-mainstream time discretizations may help explicit methods be competitive with implicit methods for the integration of large-scale nonlinear systems, a few of which I will introduce next.

6.3 Applications and Equations

I will be testing my artificial-viscosity-based shock capturing scheme on a number of different hyperbolic conservation laws, ranging from the very simple to the rather complicated.

6.3.1 Advection Equation

At the very simple end of the spectrum, the *advection equation*

$$\partial_t u + \partial_x u = 0$$

transports its initial condition along its one characteristic, described by the velocity vector v. I will apply artificial viscosity to this PDE as

$$\partial_t u + v \cdot \nabla_x u = \nabla_x \cdot (\nu \nabla_x u).$$

Here, and in all further equations, it is important to write the viscosity in "conservation" form $\nabla_x \cdot (\nu \nabla_x u)$. The desired consequence of this is that the resulting DG method will be conservative [Arnold et al., 2002].

In DG discretizations of this equation, I use an upwind flux

$$\hat{n} \cdot F_N^* := (\hat{n} \cdot v) \begin{cases} u^- & \hat{n} \cdot v \geq 0, \\ u^+ & \hat{n} \cdot v < 0 \end{cases}$$

in a strong-form DG formulation. The diffusion term $\nabla_x \cdot (\nu \nabla_x u)$ is discretized by a first-order ("dual") *interior penalty method* [Arnold et al., 2002], with the gradient being computed in strong form, and the divergence computed in weak form. The diffusive fluxes are given by

$$u_N^* := \{u_N\}, \qquad \sigma_N^* := \{\nu \nabla_{x,h} u_N\} - \frac{N^2}{h}\nu \left[\!\left[u_h \right]\!\right],$$

where σ_N is the discretization of $\nu \nabla_x u$.

6.3.2 Second-Order Wave Equation

Upon adding another, opposite characteristic to the advection equation, one obtains the second order wave equation $\partial t^2 u + c^2 \triangle u = 0$, which may be rewritten as a first-order system of conservation laws as

$$\partial_t u + c\nabla_x \cdot v = 0, \tag{6.2a}$$

$$\partial_t v + c\nabla_x u = 0. \tag{6.2b}$$

I will apply artificial viscosity to this system in the form

$$\partial_t u + c\nabla_x \cdot v = \nabla_x \cdot (\nu \nabla_x u), \tag{6.3a}$$

$$\partial_t v + c\nabla_x u = \nabla_x \cdot (\nu \nabla_x v), \tag{6.3b}$$

where I have again been careful to use the conservative form of the diffusive term. The vector diffusion term $\nabla_x \cdot (\nu \nabla_x v)$ is to be read as the diffusion ν being applied to each component separately.

The wave equation is valuable for testing artificial viscosity methods because it is the simplest system where the effects of two coupled characteristics may be observed. In particular, since I am choosing to use a single artificial viscosity ν that applies to both components of the system, this system enables me to observe whether this simple choice entails any undesired consequences. The discontinuity sensor to be described below operates on the component u.

In DG discretizations of this equation, I use an upwind flux

$$\hat{n} \cdot F_N^* := c \begin{pmatrix} \hat{n} \cdot \{v\} - \frac{1}{2}(u^- - u^+) \\ \hat{n}\left(\{u\} - \frac{\hat{n}}{2} \cdot (v^- - v^+)\right) \end{pmatrix}$$

in a strong-form DG formulation. The diffusion terms $\nabla_x \cdot (\nu \nabla_x u)$ and $\nabla_x \cdot (\nu \nabla_x v)$ (collectively $\nabla_x \cdot (\nu \nabla_x q)$) are again discretized by a first-order ("dual") *interior penalty method* [Arnold et al., 2002], with the gradient being computed in strong form, and the divergence computed in weak form. The diffusive fluxes are given by

$$q_N^* := \{q_N\}, \qquad \sigma_N^* := \{\nu \nabla_{x,h} q_N\} - \frac{N^2}{h} \nu \, [\![q_h]\!] \,,$$

where σ_N is the discretization of $\nu \nabla_x q$ and q varies through u and each of the components of v.

6.3.3 Burgers' Equation

While the linear hyperbolic conservation laws discussed so far will (in one dimension) only propagate discontinuities already present in their initial condition, *Burgers' equation* is a nonlinear conservation law whose solution will spontaneously develop discontinuities. This simple fact makes the equation valuable as a testing prototype for more the subsequent, more complicated Euler equations.

The equation is given by

$$\partial_t u + \partial_x \left(\frac{u^2}{2} \right) = 0. \tag{6.4}$$

As in Section 6.3.1, I apply the artificial viscosity simply as

$$\partial_t u + \partial_x \left(\frac{u^2}{2} \right) = \partial_x (\nu \partial_x u). \tag{6.5}$$

In DG discretizations of this equation, I use a *local Lax-Friedrichs* (or *Rusanov*) flux

$$\hat{n} \cdot F_N^* := \hat{n} \cdot \frac{F(u^+) + F(u^-)}{2} - \frac{\lambda_{\text{max}}}{2}(u^+ - u^-),$$

where λ_{max} is the maximum characteristic speed, in a weak-form DG formulation. The diffusion term is discretized as in Section 6.3.1. In multiple dimensions (see Section 6.6.5), the nonlinear inner products arising in the Galerkin formulation of (6.4) and (6.5) are integrated using the simplicial quadrature formulas by Grundmann and Möller [1978], which provide equivalent accuracy at a somewhat lower point count than the schemes used by Hesthaven and Warburton [2007]. The chosen quadrature is exact to degree $3N$, where N is the polynomial degree of the approximation.

6.3.4 Euler's Equations of Gas Dynamics

Lastly, the system of conservation laws that justifies the effort spent on this study, *Euler's equations of gas dynamics*, broadly applies to compressible, inviscid flow problems. It is given by

$$\partial_t \rho + \nabla_x \cdot (\rho \mathbf{u}) = 0, \tag{6.6a}$$

$$\partial_t (\rho \mathbf{u}) + \nabla_x \cdot (\mathbf{u} \otimes (\rho \mathbf{u})) + \nabla_x p = 0, \tag{6.6b}$$

$$\partial_t E + \nabla_x \cdot (\mathbf{u}(E + p)) = 0. \tag{6.6c}$$

As in Section 6.3.2, I am again choosing to use a single artificial viscosity ν that applies to all components of the system, such that I get the viscosity-endowed system

$$\partial_t \rho + \nabla_x \cdot (\rho \mathbf{u}) = \nabla_x \cdot (\nu \nabla_x \rho), \tag{6.7a}$$

$$\partial_t (\rho \mathbf{u}) + \nabla_x \cdot (\mathbf{u} \otimes (\rho \mathbf{u})) + \nabla_x p = \nabla_x \cdot (\nu \nabla_x (\rho \mathbf{u})), \tag{6.7b}$$

$$\partial_t E + \nabla_x \cdot (\mathbf{u}(E + p)) = \nabla_x \cdot (\nu \nabla_x E). \tag{6.7c}$$

The discontinuity sensor to be described below operates on the component ρ.

Persson and Peraire [2006] suggest that a Navier-Stokes-like physical viscosity may provide sufficient control of jumps and will not unduly smooth out contact discontinuities. On the other hand, it is obvious that such a system is effectively unable to control initial discontinuities (and therefore oscillations) in ρ. I therefore deem such a viscosity application unfit for my purpose.

In DG discretizations of this system, I use a *local Lax-Friedrichs* (or *Rusanov*) flux as in Section 6.3.3 in weak-form DG. The diffusion term is discretized as in Section 6.3.2. As above, a quadrature exact to degree $3N$ is used to integrate the nonlinearity.

6.4 A Smoothness-Estimating Detector for the Selective Application of Artificial Viscosity

6.4.1 Detection Methods in the Literature

Detectors for the selective application of artificial viscosity have been built in a large variety of ways. The most popular, perhaps, is sensing on the L^2 norm of the residual of the variational form [Bassi and Rebay, 1994, Jaffre et al., 1995]. Hartmann [2006] employs a similar indicator that includes sensing of the primary orientation of the discontinuity and performs anisotropic mesh refinement based on this data.

Other detectors in the literature employ information gathered not on the whole volume of the domain, but only on element faces [Bassi et al., 1997]. Specializing further, some methods use the magnitude of the facial inter-element jumps as an indicator of how well-resolved the solution is and to what degree it has converged [Barter and Darmofal, 2010, Feistauer and Kučera, 2007].

A further approach to shock detection repurposes entropy pairs, objects from the solution theory for scalar conservation laws, for the purposes of shock detection [Guermond and Pasquetti, 2008].

My approach most directly traces its lineage to work by Persson and Peraire [2006], which addresses one crucial shortcoming in much of the above work: scaling. Many of the quantities discussed clearly relate directly to how well-resolved (and smooth) the approximate solution of the system is. It is however rarely clear how large a value of the quantity in question indicates that a problem exists, and a variety of ad-hoc scaling choices are proposed, often by the maximum of the quantity found across the domain, or by the element-local norm, but without assigning an explicit meaning to the scaled quantity.

The method by Persson and Peraire [2006] also performs scaling by the element-local L^2 norm $\|q_N\|_{L^2(\mathsf{D}_k)}$ of the discretized value of the quantiy q_N to be sensed on. On each element D_k, it obtains a value

$$S_k := \frac{(q_N, \phi_{N_p-1})^2_{L^2(\mathsf{D}_k)}}{\|q_N\|^2_{L^2(\mathsf{D}_k)}}, \tag{6.8}$$

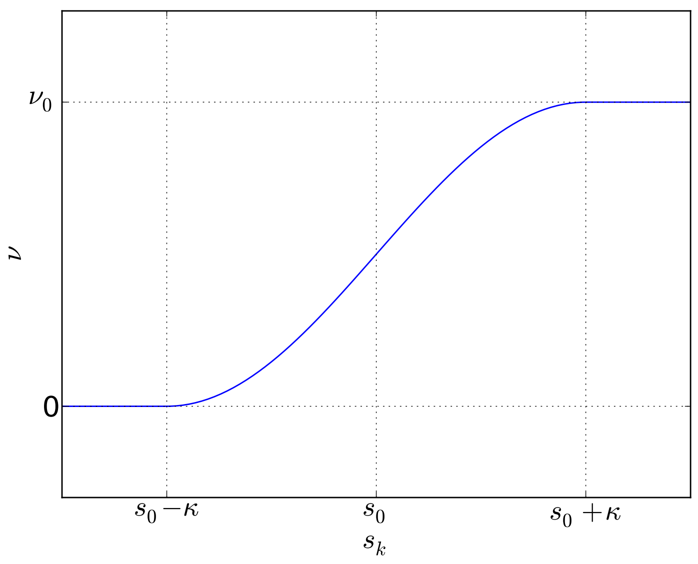


Figure 6.2. Viscosity activation map for the sensor of Persson and Peraire [2006].

where $\{\phi_n\}_{n=0}^{N_p-1}$ is an orthonormal basis for the expansion space [see e.g. Dubiner, 1991, Koornwinder, 1975] numbered from 0. Simply put, S_k reflects the (squared) fraction of q_N's mass contained in the highest mode of the expansion, relative to all mass present on the element. Persson and Peraire [2006] then invoke an analogy to Fourier expansions, where a continuous function (roughly) can be recognized by having Fourier expansions in which the nth mode's magnitude scales at most as $1/n^2$. In doing so, they have conveniently solved the issue of scaling–it is now understood what S_k measures and what value it is supposed to take on for which degree of smoothness. Based on this analogy, they argue that S_k should have a magnitude of $1/N^4$ for q_N to be continuous, or, alternatively, that smoothing by artificial viscosity should activate if $S_n > 1/N^4$.

They achieve this activation through a sequence of mapping steps. First, they take the logarithm

$$s_k := \log_{10} S_k$$

to obtain a quantity that scales linearly with the decay exponent, which they put in relation to a quantity s_0 that they claim should scale as $1/N^4$. I believe this is a typographical error in their paper, because for proper comparability, s_0 should scale with the *logarithm* of

$1/N^4$. Through the application of a mapping function pictured in Figure 6.2, they obtain the final per-element viscosity

$$\nu_k = \nu_0 \begin{cases} 0 & s_k < s_0 - \kappa, \\ \frac{1}{2}\left(1 + \sin\frac{\pi(s_k - s_0)}{2\kappa}\right) & s_0 - \kappa \le s_k \le s_0 + \kappa, \\ 1 & s_0 - \kappa \le s_k \le s_0 + \kappa, \end{cases} \tag{6.9}$$

where ν_0 is the maximum viscosity, which Persson and Peraire [2006] suggest to scale with h/N and κ is the spread of the activation ramp in Figure 6.2 on the previous page.

The focus of the remainder of this chapter is to identify a number of issues and make a number of improvements to this method of finding an artificial viscosity. For example, as it stands, the method requires choosing s_0, ν_0, κ–a multitude of parameters, many of which are to be found empirically. One of my goals will be to reduce the number of parameters significantly. Secondly, since (6.8) focuses on the very last mode of the expansion, it does not treat every direction in space equally, and it may be more sensitive in one direction (depending on the element's local-to-global map) than in another.

6.4.2 Estimating Solution Smoothness

Before I begin my discussion of the refinements to the method, let me set the stage by discussing the type of numerical method at which the to-be-designed artificial viscosity is aimed. As was already discussed, for methods of low approximation order (and polynomial degrees $N \lessapprox 2$), the flux limiting literature provides plenty of alternatives for shock capturing, and therefore will not be the main target area for my work. Very few serviceable shock capturing schemes are available for polynomial degrees $N \in \{3, 4\}$. Since my method, like the work of Persson and Peraire [2006] will try to extract smoothness information from the modal expansion of the solution, it is my hope that the expansion at these degrees already contains enough smoothness information to be viable as a basis for an artificial viscosity, and whether this is actually so will be briefly discussed in Section 6.6. Lastly, at degrees $N \gtrapprox 5$, there is guaranteed to be sufficient smoothness information, though the time step restriction (6.1) may make these approximations somewhat impractical.

I begin my deconstruction and rebuild of the Peraire-Persson estimator by examining the assumption that, like for Fourier series, smoothness can be estimated by modal decay. In Fourier series, this can be justified by viewing what happens if a derivative of an expanded function is taken (and hence smoothness is reduced)–the nth coefficient's magnitude gets multiplied by n. This results in the identity

$$\left\| \frac{d}{dx} e^{inx} \right\|_{L^p((-\pi,\pi))} = n \left\| e^{inx} \right\|_{L^p((-\pi,\pi))} \qquad \text{for } p \in [1,\infty]. \tag{6.10}$$

An polynomial analog for (6.10) is provided by Bernstein's inequality [Borwein and Erdélyi, 1995, Warburton and Hagstrom, 2008]

$$\left| \frac{d}{dx} P(x) \right| \leq \frac{n}{\sqrt{1-x^2}} |P(x)| \qquad \text{for } P \in P^n([-1,1]), x \in [-1,1]. \tag{6.11}$$

While it conveniently exhibits the same scaling as its Fourier counterpart, unfortunately, this estimate breaks down near the domain boundaries. Markov's inequality [ibid.]

$$\left\| \frac{d}{dx} P(x) \right\|_{L^\infty([-1,1])} \leq n^2 \left\| P(x) \right\|_{L^\infty([-1,1])} \qquad \text{for } P \in P^n([-1,1]). \tag{6.12}$$

extends the estimate out to the domain boundary, at the expense of a larger scaling. Further, it may be argued that if one wants to transfer the knowledge gained from (6.12) to a modal setting, L^∞ is the wrong norm, and one should consider the L^2 norm instead to be able to benefit from Parseval's identity. Fortunately, an L^2 analog of (6.12) is available [Warburton and Hagstrom, 2008, and references therein]

$$\left\| \frac{d}{dx} P(x) \right\|_{L^2([-1,1])} \leq \sqrt{3} n^2 \, \|P(x)\|_{L^2([-1,1])} \qquad \text{for } P \in P^n([-1,1]), \tag{6.13}$$

known as an *inverse inequality*. Taking into account (6.11) and (6.13), the polynomial analogy to the Fourier case is therefore expected to carry over well for non-smoothness occurring on the interior of each finite element, whereas for non-smoothness at the domain boundary, the smoothness measure will likely differ.

Having examined the viability of modal decay as an estimator for smoothness, I seek to make the notion of modal decay more precise than (6.8). I presume that, for the modal coefficients $\{\hat{q}_n\}_{n=0}^{N_p-1}$ of a member q_N of the L^2-orthonormal approximation space spanned by $\{\phi_n\}_{n=0}^{N_p-1}$, modal decay is approximately representable as

$$|\hat{q}_n| \sim c n^{-s}. \tag{6.14}$$

Taking the logarithm of the relationship (6.14) yields

$$\log |\hat{q}_n| \sim \log(c) - s \log(n),$$

an affine relationship whose coefficients s and $log(c)$ may be found through least-squares fitting, satisfying

$$\sum_{n=1}^{N_p-1} \left| \log |\hat{q}_n| - (\log(c) - s \log(n)) \right|^2 \to \min! \tag{6.15}$$

Observe that the decay rate of (6.14) has rather little to do with the presumed magnitude of the remainder term of an expansion, on which most a-priori error estimates for finite element solutions are based–these *start* with an assumption of sufficient smoothness. There is a connection, however. Mavriplis [1994], in the context of mesh adaptation, has used a

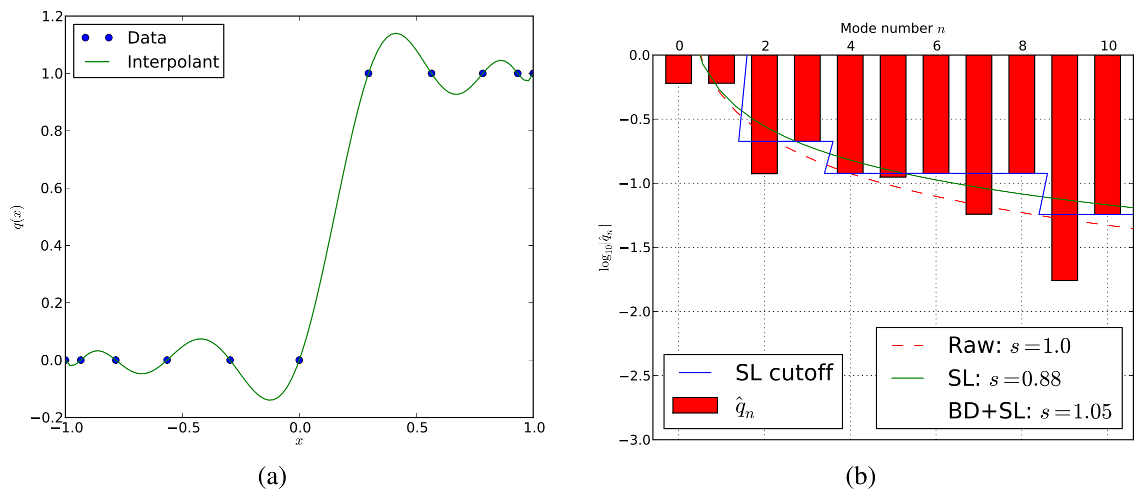


Figure 6.3. Modal portrait for an approximant of a (discontinuous) Heaviside jump function. Subfigure (a) shows the nodal data and its unique polynomial interpolant. Subfigure (b) shows the modal coefficients of a Legendre expansion of the function in (a), the processing of these coefficients, and the unprocessed and postprocessed smoothness estimates.

similar least-squares fit to the modal decay, defining a continuous function $\hat{q}(n)$ through the found fit. She then proceeds to estimate the remainder term of the expansion as

$$\|q - q_N\|^2_{L^2(\mathsf{D}^k)} \approx \left(\frac{\hat{q}_N^2}{\frac{2N+1}{2}} + \int_{N+1}^{\infty} \frac{\hat{q}(n)^2}{\frac{2n+1}{2}} \,\mathrm{d}n \right).$$

This remark aside, the least-squares procedure (6.15) yields an estimate s of the decay exponent. If the analogy with Fourier modal decay holds water, one would then expect $s \approx 1$ for a discontinuous q, $s \approx 2$ for $q \in C^0 \setminus C^1$, $s \approx 3$ for $q \in C^1 \setminus C^0$, and so forth. Figure 6.3 shows a first attempt at determining whether this is really the case by examining an interpolant of a Heaviside jump function as shown in Figure 6.3(a). Figure 6.3(b) shows the magnitudes of the first ten modal coefficients along with the fitted curve (the dashed red line). The obtained decay exponent s, shown in the legend next to the dashed red line, matches the expectation rather well, giving a value of exactly 1.

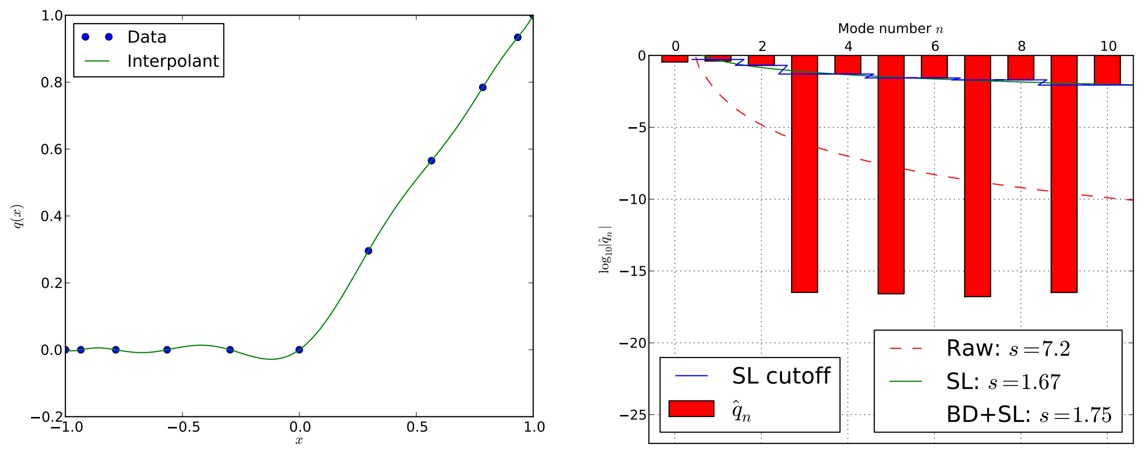
Before moving on from this first successful test, I would like to comment on two important features of (6.15) that deserve some extra attention. Notice that although throughout this chapter I have started numbering nodes at zero, the sum in (6.15) starts at one. This latter choice is easy to justify: The goal of this procedure is to estimate smoothness. For every common definition of smooth, added constants do not matter–$q(x) + c$ for a real constant c is considered just as (non)smooth as q itself. It would therefore run counter to the stated goal if the zeroth mode (which exactly represents an additive constants) was included in the modal fit. Note that, in disregarding the zeroth mode, one is discarding potentially useful information. Further below, this problem will make itself felt, and the now-discarded information will be reintegrated into the estimate in a different form.

The other important feature of (6.15) is that the numbering of nodes starts at zero at all, which is not immediate. Further, if the zeroth mode had not been eliminated above, this numbering choice would have caused the use of a logarithm of zero in the specification of the fit. So why is a zero-based numbering natural for modes, as, because of the logarithm, shifted numberings are not equivalent? The reason for this goes back to (6.10) and (6.13), which I have used as an anchor for the entire construction of my estimator. These formulas are only valid if modes are numbered starting from zero–any other numbering makes them false.

Continuing this line of experimentation, I would like to move on to an interpolant of a “kink” function

$$q(x) := \begin{cases} 0 & x < 0, \\ x & x \geq 0. \end{cases}$$

The same observations as for the Heaviside function are shown in Figure 6.4 on the next page. Unfortunately, the figure reveals a rather powerful shortcoming of the modal fit method as developed so far. An odd-even effect draws the coefficients for the odd modes of number three and greater to zero, leading to machine zeros ($\approx 10^{-15}$) in those approximate coefficient numbers. These “fully converged” coefficients fool the estimator into thinking

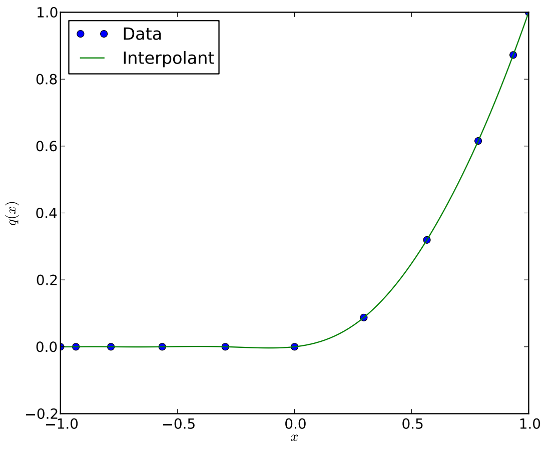


(a) (b)

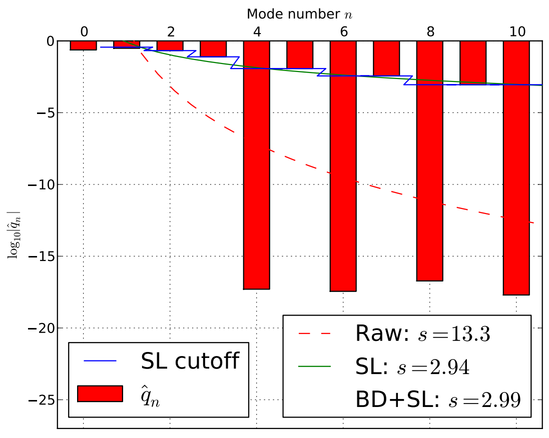
Figure 6.4. Modal portrait for an approximant of a C^0 non-differentiable "kink" function. Subfigure (a) shows the nodal data and its unique polynomial interpolant. Subfigure (b) shows the modal coefficients of a Legendre expansion of the function in (a), the processing of these coefficients, and the unprocessed and postprocessed smoothness estimates.

that far more smoothness is present than is actually the case, leading to an estimated decay exponent of about seven–far too high.

It is unfortunate that the fit can be misled that easily, but a close look at Figure 6.4(b) will have already revealed to the attentive reader that this is an easily recoverable issue. Realize that the fit tries to model modal decay, i.e. the shrinking of modal coefficient magnitudes $|\hat{q}_n|$ as n increases. The model (6.14) that is fitted to the decay only generates monotone modal decays. Figure 6.4(b) is characterized by a strongly non-monotone mode profile, and this is precisely what is misleading the estimator. Consider this: Given a mode n with a small coefficient $|\hat{q}_n|$, if there exists another coefficient with $m > n$ and $|\hat{q}_m| \gg |\hat{q}_n|$, then the small coefficient $|\hat{q}_n|$ was likely spurious, just like the near-zero coefficients in Figure 6.4(b) were spurious. These spurious coefficients should hence be eliminated from the fit, and this is what a new procedure, termed *skyline pessimization*, achieves. From the modal coefficient magnitudes $\{|\hat{q}_n|\}_{n=0}^{N_p-1}$, it generates a new set of



(a)



(b)

Figure 6.5. Modal portrait for an approximant of a C^1 truncated polynomial. Subfigure (a) shows the nodal data and its unique polynomial interpolant. Subfigure (b) shows the modal coefficients of a Legendre expansion of the function in (a), the processing of these coefficients, and the unprocessed and postprocessed smoothness estimates.

modal coefficients by

$$\bar{q}_n := \max_{i \in \{\min(n, N_p - 2), \dots, N_p - 1\}} |\hat{q}_i| \qquad \text{for } n \in \{1, 2, \dots, N_p - 1\}. \tag{6.16}$$

The effect of the procedure is that each modal coefficient is raised up to the largest higher-numbered modal coefficient, eliminating non-monotone decay. Since odd-even effects in modal portraits (such as the one of Figure 6.4(b) are a common phenomenon, there is a slight modification in (6.16) accounting for the last mode, which is forced to also be larger than the second-to-last mode. This would become an issue if, for example, only the first nine modes of Figure 6.4(b) were used, in which case the smallness of the last coefficient would again cause an artificially high smoothness exponent. Once skyline pessimization has been performed, decay estimation (6.15) is applied to them in the same fashion as above, yielding a corrected decay estimate.

The effect of skyline pessimization is shown in the modal portrait of Figure 6.4(b) as a zig-zagged blue line that appears to “truncate” the bars representing modal coefficients

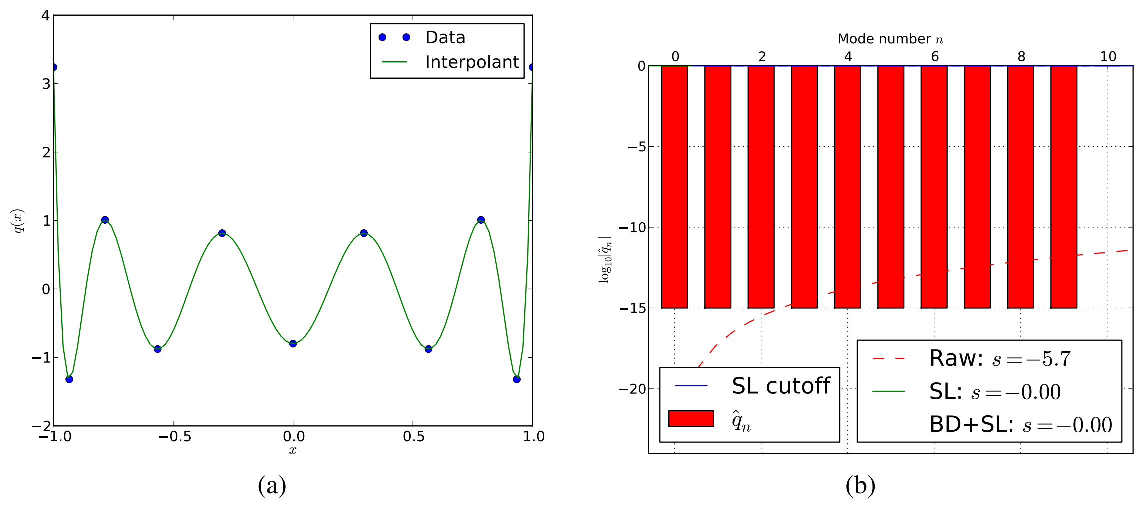


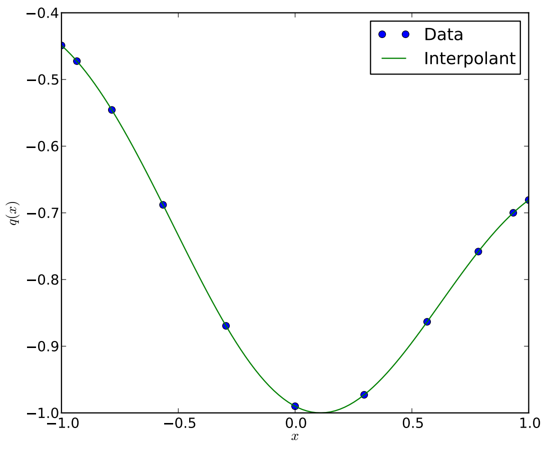
Figure 6.6. Modal portrait for a function consisting of only the highest representable Legendre mode ϕ_{N_p-1} in an expansion of length 10. Subfigure (a) shows the nodal data and its unique polynomial interpolant. Subfigure (b) shows the modal coefficients of a Legendre expansion of the function in (a), the processing of these coefficients, and the unprocessed and postprocessed smoothness estimates.

at the level of the largest higher-numbered coefficient. Further, the fitted decay curve is shown in green, along with the resulting estimated decay exponent, labeled as "SL". With skyline pessimization in place, the estimated smoothness exponent for the "kink" example becomes 1.67–reasonably close to the expected value of 2.

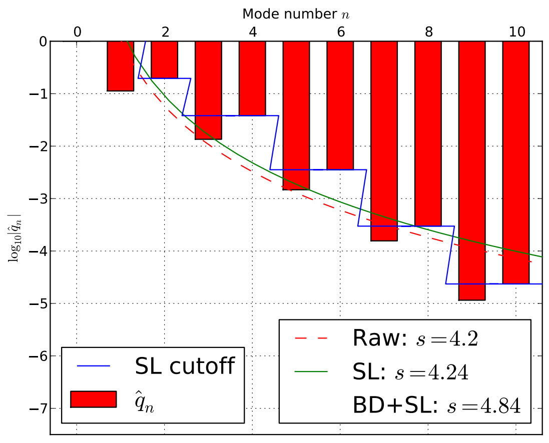
Figure 6.5 on the preceding page shows the next-smoothest test of the estimator, a truncated polynomial

$$q(x) := \begin{cases} 0 & x < 0, \\ x^2 & x \geq 0. \end{cases}$$

Obviously, $q \in C^1 \setminus C^2$. As in the "kink" case, Figure 6.5(b) shows a pronounced odd-even discrepancy, that leads to spuriously high "raw" smoothness exponent estimate of about 13. After skyline pessimization, the estimate assumes nearly exactly the expected value, three. The three artificial tests conducted so far confirm the premise on which the estimator is built, namely that the smoothness of a function represented by a Legendre expansion can be accurately estimated solely by examining its coefficients.



(a)



(b)

Figure 6.7. Modal portrait for the function $\cos(3 + \sin(1.3x))$, as an example of a very smooth function. Subfigure (a) shows the nodal data and its unique polynomial interpolant. Subfigure (b) shows the modal coefficients of a Legendre expansion of the function in (a), the processing of these coefficients, and the unprocessed and postprocessed smoothness estimates.

By presenting a number of further tests, I hope to clarify the behavior of the estimator as designed so far. A particularly interesting case is shown in Figure 6.6 on the previous page, which shows the estimator applied to the highest mode present in the Legendre expansions of length 10 which I have been considering. In a sense, this is the most oscillatory, and thereby the least smooth, function that the expansion can express. After skyline pessimization, this function is assigned a smoothness exponent of zero–which in a Fourier setting would correspond to white noise.

The next two tests, of Figures 6.7 and 6.8 on the next page, are concerned with very smooth functions and confirm that the estimator recognizes them as such. While the smoothness values (both around four) assigned to them are not as meaningful as the results in the low-smoothness examples, this is not necessarily a problem. As long as the estimator can sharply pick up non-smoothness on a reliable scale (and keep the smooth examples clear of this area), it is performing satisfactorily for its purpose.

The second-to-last test that I am portraying, shown in Figure 6.9 on page 138, highlights

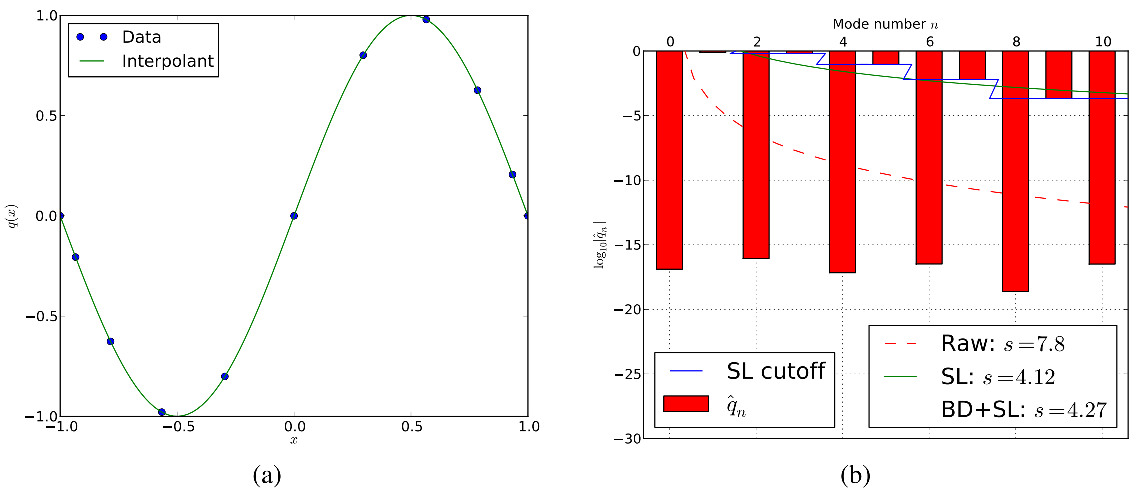


Figure 6.8. Modal portrait for the function $\sin(\pi x)$, as an example of a smooth, odd function. Subfigure (a) shows the nodal data and its unique polynomial interpolant. Subfigure (b) shows the modal coefficients of a Legendre expansion of the function in (a), the processing of these coefficients, and the unprocessed and postprocessed smoothness estimates.

a behavior of the detector that could be considered a failure mode. Shown is a constant–1 is this case–perturbed by white noise of a much smaller scale–in this case 10^{-3}. The graph also shows a number of further diagnostics that I will further explain below. As discussed above, the detector ignores the constant one, and thus all it sees is white noise, of a largely constant modal makeup, and therefore both unaided and skyline-pessimization-assisted decay estimation–correctly–yield a smoothness value of about zero. Unfortunately, this is often "wrong" from an application point of view. Consider the case where the solution of the PDE under consideration has extended areas where the solution is constant. Invariably, these areas will be contaminated by floating point noise in the least significant digits of the solution. Ignoring the constant, the estimator will look for smoothness in the floating point noise, and it will not find any. This would lead to spurious activations of the artificial viscosity in areas that are smooth (constant!) by all conventional definitions.

This problem is rooted in the (correct) removal of constant-mode information from the estimation process, causing the estimator to not have a "sense of scale", i.e. keeping it from noticing that the noise is "small" compared to the remainder of the solution. In the following, I present one (somewhat ad-hoc) way to re-add this “sense of scale” by distributing energy according to a “perfect modal decay”, which is defined as

$$|\hat{b}_n| \sim \frac{1}{\sqrt{\sum_{i=1}^{N_p-1} \frac{1}{n^{2N}}}} \frac{1}{n^N} \tag{6.17}$$

for N the polynomial degree of the method, where the normalizing factor ensures that

$$\sum_{n=1}^{N_p-1} |\hat{b}_n|^2 = 1.$$

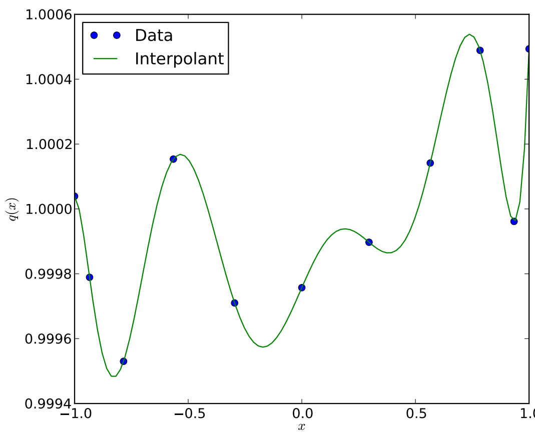
The idea is to consider the coefficients

$$|\tilde{q}_n|^2 := |\hat{q}_n|^2 + \|q_N\|^2_{L^2(\mathsf{D}_k)} |\hat{b}_n|^2 \qquad \text{for } n \in \{1, \dots, N_p - 1\} \tag{6.18}$$

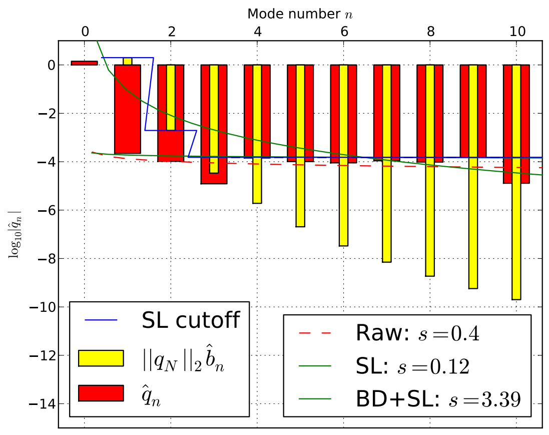
as input to skyline pessimization instead of the “raw” coefficients $|\hat{q}_n|^2$. The right way of viewing this modification is as adding a *baseline decay*, scaled by the element-wise norm. The desired effect of this change is to control coefficients that are spuriously small compared to the element-wise norm. Baseline decay will not generally make measured smoothness worse. To see this, consider

$$\log(a^2 + b^2) \geq \max\{\log(a^2), \log(b^2)\}$$

with $a = |\hat{q}_n|$ and $b = \|q_N\|_{L^2(\mathsf{D}_k)} |\hat{b}_n|$ in the context of (6.18). This is relevant because decay estimation operates on a logarithmic scale, i.e. it operates on the logarithm of the sum of squares in (6.18). In adding the baseline decay, one is setting a “baseline” minimum coefficient magnitude, below which small coefficients should not contribute to a poor smoothness measurement. The baseline decay therefore precisely addresses the problem motivating it.



(a)



(b)

Figure 6.9. Modal portrait of the constant 1, perturbed by white noise of magnitude 10^{-3}. Subfigure (a) shows the nodal data and its unique polynomial interpolant. Subfigure (b) shows the modal coefficients of a Legendre expansion of the function in (a), the processing of these coefficients, and the unprocessed and postprocessed smoothness estimates.

The effect of the baseline decay can be seen in Figure 6.9, where the yellow bars indicate the magnitude of the scaled baseline decay. Unlike the flat “white-noise” fit seen above, the small coefficients at the start of the expansion are increased to baseline level, resulting in a smoothness estimate of about three, which better matches the expectations set forth above.

The crucial ingredient that makes the baseline decay work is the knowledge of the entire element-wise norm of the measured quantity q_N. Conversely, it cannot help if it loses its sense if $\|q_N\|_{L^2(\mathsf{D}_k)}$ is exactly or nearly zero. The case of exact zeros may be handled specially and is easy to catch, but near-zeros are more difficult. Here, q_N consists *entirely* of floating point noise. This is the only case known to me in which the detector fails, and I am not aware of a usable automatic recovery. If defined behavior is desired in this case, the user might supply a defined minimum value for element-wise norm scaling in (6.18). In many practical cases, such as $q_N = \rho_N$ in the Euler equations, this issue is fortunately entirely irrelevant, because the density only takes positive values.

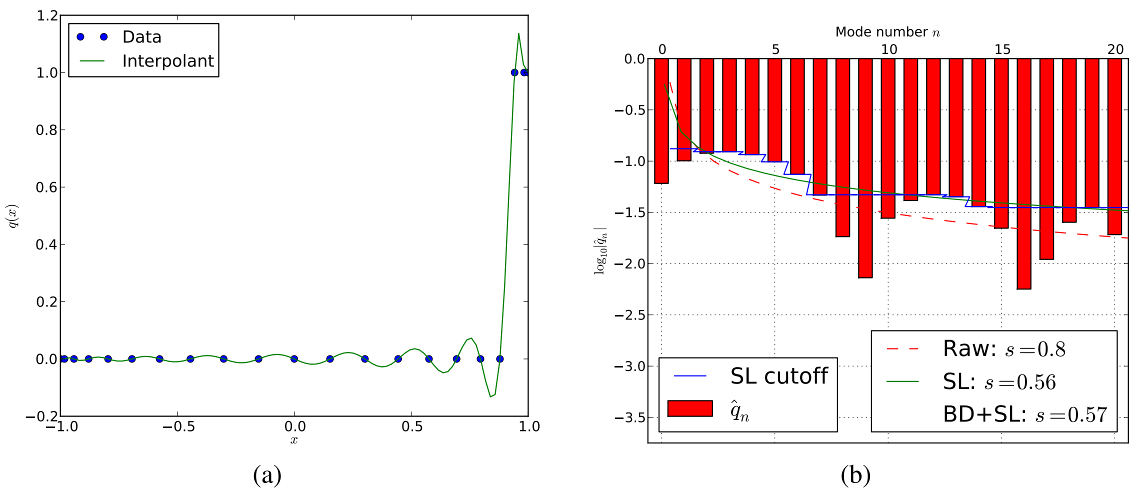


Figure 6.10. Modal portrait for an approximant of a (discontinuous) jump function, offset from the center of the element. Subfigure (a) shows the nodal data and its unique polynomial interpolant. Subfigure (b) shows the modal coefficients of a Legendre expansion of the function in (a), the processing of these coefficients, and the unprocessed and postprocessed smoothness estimates.

For the sake of exposition, baseline decay was not introduced upfront, but only once the need for it arose. It is obviously not wise to first spend significant time and effort convincing oneself that a method works as designed, only to reach back and modify that method, potentially voiding the results of past efforts. Fortunately, this criticism does not apply to the present situation, as the results shown so far are barely changed by the addition of the baseline decay. The reader may convince himself of this fact by examining the estimated decay exponents given as "BD+SL" in the past graphs and comparing to the pure-skyline values given as "SL". In particular, observe that the baseline decay has not changed the smoothness measurement for the case of Figure 6.6 on page 134, even though small modal coefficients occur at the start of the expansion. In summary, the addition of baseline decay does not invalidate any of the statements made in the text so far.

This completes the discussion of the design of the detector. Now might also be a good time to point out a known shortcoming in its design that was already anticipated in the motivating discussion. The issue relates to the discussion of mode scaling with decreasing smoothness initiated earlier in this section. Consider Figure 6.10, which shows decay estimation data for the same Heaviside jump function as Figure 6.3 on page 130, but shifted to the element's edge. The data in the figure confirms the earlier conjecture that a function of the smoothness might result in modal decay exponents that differ by up to a factor of two, depending on where the non-smoothness is located inside the element–the measured smoothness exponent for the shifted Heaviside function is only 0.57, compared to 1.05 after all corrections above. Additional confirmation comes from the fact that the final smoothness estimates for boundary-shifted versions of the kink and the C^1 spline are $s = 1.19$ and $s = 2.24$ respectively (not shown, original versions in Figures 6.4 on page 132 and 6.5 on page 133). This relates in striking ways to the scaling of the DG CFL condition (6.1), and like in its case, a remedy for this issue is not yet known.

Based on the shown examples, it should be clear that even the unassisted decay fit is a more robust smoothness estimator than the single-mode indicator (6.8), if only for the simple reason that it considers a much broader set of modal data. But I have shown that even this fairly robust indicator can give poor results in surprisingly common cases. I feel that this strongly supports the statement that the decay fit indicator *with* skyline pessimization and added baseline decay represents a more practical–if more expensive–way of obtaining smoothness information on a numerical solution.

6.4.3 Ambiguities in Two and More Dimensions

As hinted in the introduction, all of the construction features of the smoothness indicator discussed so far generalize seamlessly to multiple dimensions, except for skyline pessimization, which depends on an ordering of mode indices to increase modal coefficients according to

$$\bar{q}_n := \max_{i \le n} |\hat{q}_i| \qquad \text{for } n \in \{1, 2, \ldots, N_p - 1\}, \tag{6.19}$$

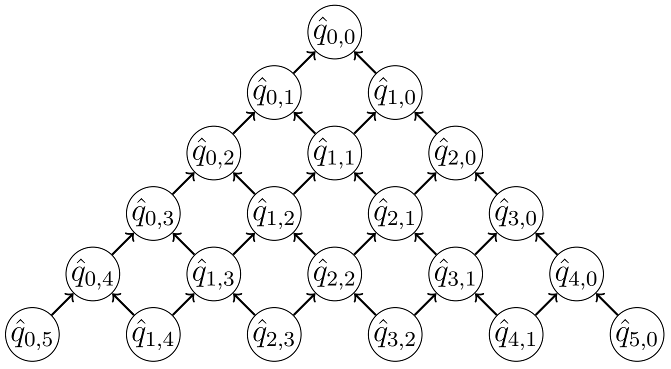


Figure 6.11. Modal adjacency ordering for skyline pessimization in the case of a triangle (i.e. a "2D simplex").

where "$\leq$" is given by the ordering. If the modal indices are captured in a tuple $i = (i_1, \dots, i_d) \in \mathbb{N}_0^d$ that one may imagine as monomial orders along each of the axes, then a number of different orderings are plausible:

Ordering by total degree $i \leq j :\Leftrightarrow \sum_{k=1}^{d} i_k \leq \sum_{k=1}^{d} j_k$,

Ordering by maximum degree $i \leq j :\Leftrightarrow \max_{k=1}^{d} i_k \leq \max_{k=1}^{d} j_k$,

Ordering by adjacency. This ordering arises as the transitive closure of the relation

$$i \prec j :\Leftrightarrow \exists k \in \{1, \dots, d\} : i + e_k = j,$$

where e_k is the kth unit vector, (This ordering is depicted for a triangle in Figure 6.11.)

and probably many more. A further, more ad-hoc possibility, which was used in the few two-dimensional experiments carried out in Section 6.6, is to sum up the squares of the modal coefficients in two dimensions along their total degree and reuse the one-dimensional skyline procedure.

All of these orderings can of course be modified to eliminate even-odd effects in the top modes like one-dimensional skyline pessimization. Which of these is the "right" one (or at least practically advantageous) is a subject of current study. In Section 6.6, I will show promising initial results with a simple ordering by total degree, with the even-odd fix for the top modes.

Ordering for skyline pessimization is further not the only ambiguity potential ambiguity that arises in multiple dimensions. Since there is now significantly more modal data at high orders (as is also shown by Figure 6.11), it is not clear that all of these modes should receive the same weighting in the least-squares fit. That is, instead of finding c and s to minimize

$$\sum_{m+n\leq N} |\hat{q}_{m,n} - c(m+n)^{-s}|^2,$$

one could minimize

$$\sum_{m+n\leq N} |\omega_{m,n}(\hat{q}_{m,n} - c(m+n)^{-s})|^2$$

instead, with the weights $\omega_{m,n}$ determined in some way. Preliminary experiments carried out with $\omega_{m,n} = 1$ and $\omega_{m,n} = 1/\sqrt{m+n}$ showed no measurable benefit to using such a weighting.

6.5 From Smoothness to Viscosity

6.5.1 Scaling the Viscosity

This section assumes that the output of the indicator is an estimated decay exponent s, approximating the decay of the solution's modal coefficients as $|\hat{u}_n| \sim n^{-s}$. I am seeking to design an activation function $\nu(s)$ whose value is the viscosity coefficient.

For the interpretation of the decay exponent s, recall the targeted scaling of the smooth-

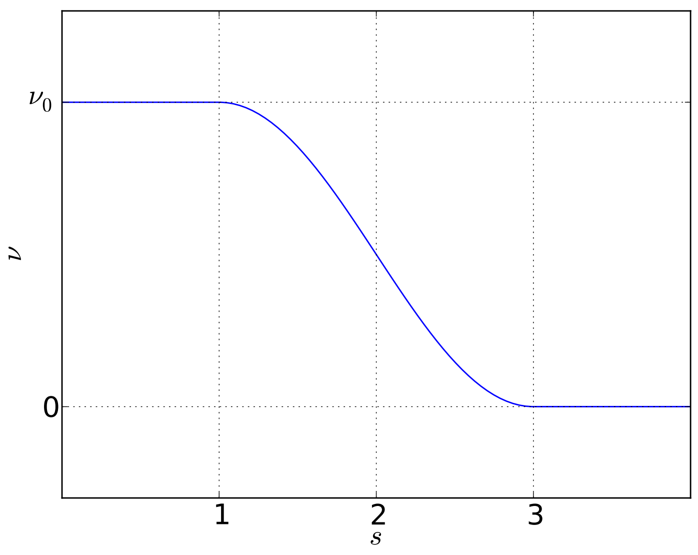


Figure 6.12. Viscosity activation map for the sensor of Section 6.4.2.

ness exponent s, where (roughly) $s = 1$ would indicate a discontinuous solution, $s = 2$ would indicate a C^0 solution, $s = 3$ a C^1 solution, and so forth. Among the chief nuisances of polynomial approximations that this work seeks to remedy is the Gibbs phenomenon, which occurs for discontinuous solutions ($s = 1$). I therefore expect to have $\nu(1) = \nu_{\text{max}}$, where ν_{max} is the maximum value of ν and dictates its scaling. Merely continuous functions still pose somewhat of a problem for polynomial approximation, so I arbitrarily fix $\nu(2) = \nu_{\text{max}}/2$, and finally I fix $\nu(3) = 0$, as I prefer that C^1 solutions should not be modified by viscosity.

In complete analogy to the activation map (6.9) by Persson and Peraire [2006], the following function provides a C^1 ramp between these values:

$$\nu(s) = \nu_0 \begin{cases} 1 & s \in (-\infty, 1), \\ \frac{1}{2}(1 + \sin(-(s-2)\pi/2)) & s \in [1, 3], \\ 0 & s \in (3, \infty). \end{cases}$$

Note that because of the close attention paid to precise scaling of the smoothness s, I was able to eliminate the ramp location and width parameters κ and s_0. The resulting activation function is shown in Figure 6.12 on the previous page.

To find an appropriate value ν_0, the behavior of the diffusion term needs to be investigated. To this end, I examine the fundamental solution of the diffusion equation $u_t = \triangle u$, the *heat kernel*. Adopting the probabilistic standard deviation σ as a measure of width, the heat kernel after time t has a width of $\sigma = \sqrt{2\nu t}$. Considering some unit t of time, the conservation law will propagate information to a distance of λ, where λ is some local characteristic velocity. Observe that viscosity propagates the bulk of its mass at a non-linear square-root pace, while the conservation law observes a linear speed. One therefore needs to pick a reference time scale t as well as a reference distance at which the two propagation distances are to coincide.

Choosing $\sigma = h/N$ after $t = (N/2)\Delta t$, and approximating $\Delta t \approx h/(\lambda N^2)$, one obtains

$$\nu_0 = \frac{\sigma^2}{2t} = \lambda \frac{h}{N}. \tag{6.20}$$

This reproduces the value of Barter and Darmofal [2010] and simultaneously provides some more detailed insight into its meaning. I would like to note that $\sigma = h/N$ is probably too ambitious a goal, as this would only smooth discontinuities to a with of about the distance between two nodal points–likely too little as Figure 6.3 on page 130 shows. A choice of $\sigma = 3h/N$ has proven to be more realistic.

For a system of conservation laws, there remains the question of which characteristic velocity should be chosen for λ. This choice has important implications as, e.g. in the Euler system, contact discontinuities propagate with stream velocity, whereas shocks propagate at sonic speeds. In a one-dimensional setting, Rieper [2010] convincingly argues that the best course of action is to perform smoothing in characteristic variables, so that each wave receives the amount of smoothing specified by the scheme, e.g. as given in (6.20). Observe that doing so fits the mold of the inapplicable strategy portrayed in Section 6.1: It works well in one-dimension and for low-order multi-D finite volume schemes, but it is less clear how it might be applied in a genuinely multidimensional situation. A simple and functional strategy is to choose λ to be the maximum characteristic velocity λ_{max}. The simplicity of this strategy comes at a price, however: returning to the example of the Euler equations, contact discontinuities have their ν_0 set higher than would be necessary from this analysis, and my numerical experiments will reflect this.

Note that the $\lambda_{\max}$-based scaling is not perfect. It works, in the sense that all test examples run successfully using it, but some can benefit from an additional ‘fudge factor’. For example, while Burgers’ problems (Section 6.3.3) work well with an unmodified scaling in a ’picture norm’ sense (little oscillation, least smoothing), most subsonic Euler problems benefit from the application of an additional factor of $1/2$. This is not entirely unexpected, given the above discussion.

Connection with the Reynolds number

Further insight can be gained from working out the relationship of the scaling of ν_0 with the Reynolds number. To that end, I consider an advection-diffusion equation

$$u_t + \lambda u_x = (\nu u_x)_x.$$

I fix characteristic length and time scales L, T and let $u(x,t) = \bar{u}v(x/L, t/T)$. Then

$$\frac{\bar{u}v_t}{T} + \lambda \frac{\bar{u}v_x}{L} = \left(\frac{\nu \bar{u} v_x}{L^2}\right)_x.$$

Setting $T = \bar{u} = L/\lambda$, I obtain

$$v_t + v_x = \left(\frac{\nu}{\lambda L} v_x\right)_x.$$

This gives a rough analog of the Reynolds number,

$$\mathrm{Re} = \frac{\lambda L}{\nu}.$$

Now consider that $\nu \in [0, \nu_0]$ with the value for ν_0 obtained above. Then the Reynolds number enforced by the scheme is in the range

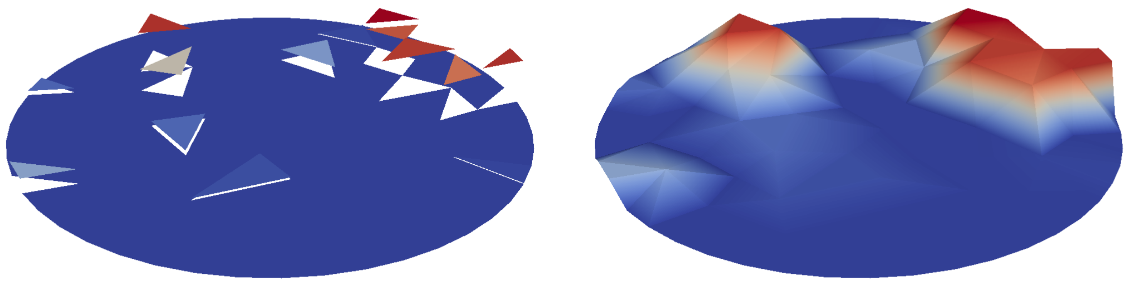
$$\mathrm{Re} \in [\frac{\lambda L}{\nu_0}, \infty).$$

If one assumes that the scheme actually uses ν up to ν_0, then the above expression gives a natural upper bound to the Reynolds numbers whose corresponding flows the scheme can resolve at a certain resolution and scaling. As a final note, observe that for the Euler equations, the above Re needs to be multiplied by ρ to match its conventional definition.

6.5.2 Smoothing the Viscosity

The artificial viscosity $\nu(x)$ obtained so far is a per-element quantity, with no guarantees on how it might vary across the domain. In particular, since the viscosity is constant on each element, it will invariably be discontinuous. Figure 6.13(a) on the next page shows a 2-dimensional surface plot of what the output viscosity $\nu(x)$ might look like.

Now observe how the viscosity is employed in the equations of Section 6.3. In particular, observe that in order to maintain conservativity, the viscosity occurs *inside* a derivative. Great care is required in the correct numerical solution of a diffusion equation with dis-



(a) A discontinuous viscosity as could be the output of the methods of Sections 6.4.2 and 6.5.1.

(b) A version of the viscosity smoothed by the vertex-wise P^1 maxima described in Section 6.5.2.

Figure 6.13. The viscosity parameter $\nu(x)$ before and after smoothing.

continuous viscosities using discontinuous Galerkin methods. Ern et al. [2009], Lörcher et al. [2008], Proft and Rivière [2009] describe various precautions that need to be taken to avoid non-conservativity and non-consistency. In my experience, however, even if appropriate methods (according to these references) are used, discontinuous viscosities (or "diffusivities" in these references) introduce numerical noise into the solution, whose removal is the declared goal of this chapter.

Feistauer and Kučera [2007] also notice the issues caused by localized, discontinuous viscosities and propose an adapted flux term to "strengthen the influence of neighbouring elements and [improve] the behaviour of the method". Barter and Darmofal [2010], through numerical experiment, also arrive at the conclusion that a discontinuous viscosity causes issues and show a marked decrease in H^1 error for smooth viscosities. Since one is at considerable liberty to choose the viscosity $\nu(x)$, I agree that it is best to choose a ν that does not include discontinuities, to avoid this entire complex of issues.

Therefore, given that the detection infrastructure built up so far works in an element-by-element fashion, one needs to introduce a post-processing step that generates a smoother variant of the generated ν. In doing so, one again has a wide array of choices. Barter and Darmofal [2010] propose a diffusion equation (effectively “diffusing the diffusivity”) with time-relaxation to obtain a viscosity that is smooth in both time and space. Unfortunately, this choice is unsuitable given the design choices laid out in Section 6.2–to achieve sufficient smoothing of the viscosity, one needs to choose a large diffusivity for it, which results in a very stiff system of ODEs. This may not pose much of a problem in a time-implicit setting, however for explicit time integration as chosen here, this would lead to gross inefficiency. One further concern is that whatever system is chosen to smooth the viscosity should not introduce under- or overshoots of its own, as an inverse diffusion equation ($\partial_t u = -\triangle u$), as might arise locally if ν undershoots zero, is not well-posed, and hence should be avoided. Unfortunately, the PDE-based viscosity of Barter and Darmofal [2010] is unable to guarantee this.

One important question in the design of a successful smoothing method is, precisely how smooth must the result of the smoothing be? In computational experiments relating to the problem of artificial viscosity, I have found that there does not seem to be an advantage to having the viscosity $\nu \in C^k$ for $k > 0$. In other words, it appears that a continuous viscosity suffices. More smoothness necessitates more sophisticated methods, so this is an important datum for the design process–especially since with higher smoothness and higher-order polynomials, the risk of oscillatory behavior increases, and undershoots in the viscosity become an issue, as mentioned above.

Another potential issue is the over- or under-response to locally clustered viscosity requests. Assume a method that, based on the requested viscosity in each element, sums element-wise smooth ‘stencils’, weighted by the requested viscosities. These stencils necessarily overlap and can, at high-degree vertices, result in a smooth viscosity that is far higher than requested by the detector. If, on the other hand, partition-of-unity-like weights are introduced to counter this effect, then the response to a viscosity request on a single element might result in a far smaller viscosity than intended.

Based on these design criteria, the successful method employed in the experiments in the next section proceeds as follows:

1. At each vertex, collect the maximum viscosity occurring in each of the adjacent elements.
2. Propagate the resulting maxima back to each element adjoining the vertex.
3. Use a linear (P^1) interpolant to extend the values at the vertices into a viscosity on the entire element.

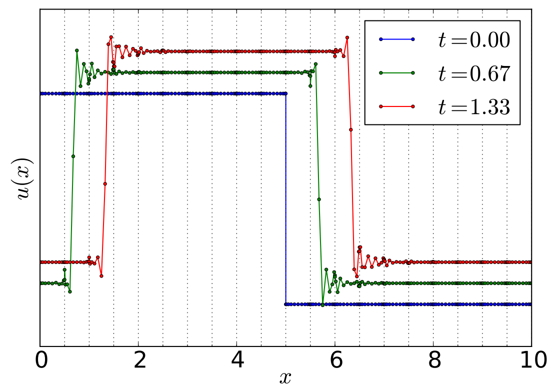
In my experience, this method is cheap, easy to implement, and it satisfies the design requirements set forth above. Figure 6.13(b) on page 147 shows the effect of this smoothing procedure on an example of a discontinuous viscosity on a disk.

6.6 Experience with and Evaluation of the Scheme

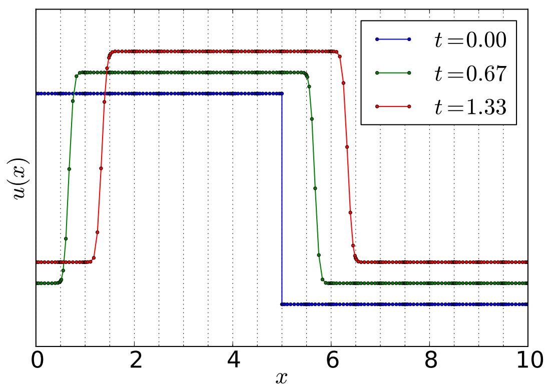
I would like to make one introductory remark regarding the results shown in this section: The normalizing factor of (6.17) was omitted in the implementation with which the experiments were carried out, which is somewhat regrettable, but because of

$$1 \le \sqrt{\sum_{i=1}^{N_p-1} \frac{1}{n^{2N}}} \le 1.01 \qquad \text{for all } N_p \ge 1,\, N_p = N+1,$$

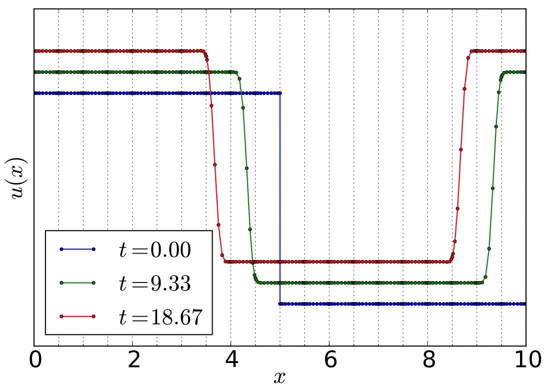
the introduced error is negligible.



(a) Solution of the advection equation without artificial viscosity.



(b) Solution of the advection equation with artificial viscosity, after short amounts of time.



(c) Solution of the advection equation with artificial viscosity, after one and two round-trips.

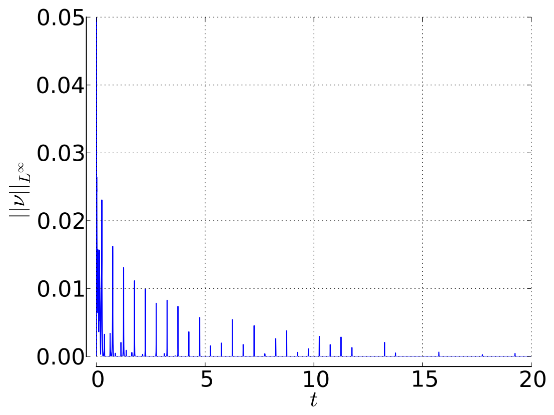
Figure 6.14. Spatial shock capturing behavior of the artificial viscosity scheme on an advection equation.

6.6.1 Advection: Basic Functionality, Interaction with Time Discretization

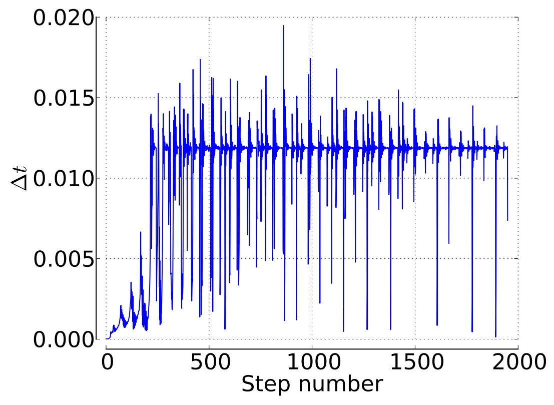
The first set of results I would like to discuss relates to the advection equation (Section 6.3.1). The examples in this section examine the advection of the function

$$u_0(x) := \begin{cases} 1 & x < 5, \\ 0 & x \geq 5 \end{cases}$$

over an interval $(0, 10)$.



(a) Artificial viscosity activations vs. time, in a discontinuous advection calculation.



(b) Adaptively found time step vs. step number, in a discontinuous advection calculation.

Figure 6.15. Interaction of the shock-capturing artificial viscosity with the time discretization.

Kuzmin et al. [2005] suggest that the advection equation is particularly suited to testing shock capturing schemes for two reasons: First, because it is the simplest PDE that can sustain a discontinuous solution, so that the behavior of the method can be observed in a well-understood setting, isolated from other characteristics and nonlinear effects. Second, because discontinuities in it are not self-steepening, in analogy to contact discontinuities in the Euler equations, it makes a challenging example to be treated with artificial viscosity: Once a discontinuity is unduly smeared by viscosity, nothing will return it to its former, sharp shape.

Figure 6.14(a) on the previous page displays the behavior of the unmodified discontinuous Galerkin method as described in Section 6.3.1. As expected, a strong Gibbs-type overshoot is observed, although it is worth noting that the used upwind fluxes already provide enough dissipation of high-frequency modes to prevent the solution from becoming useless. This example, and all examples that follow in this subsection, were run at polynomial degree $N = 10$ on a discretization using $K = 20$ elements. Further note that different time levels are vertically offset from each other in the figure for better visual discrimination. This offset is not part of the solution itself.

Next, Figure 6.14(b) on page 150 displays the result of the same calculation once the artificial viscosity machinery as described above is enabled. Discontinuities are resolved within eight points, i.e. within less than one element (containing $N_p = 11$ points) and have no visible overshoots. Element boundaries are shown as dashed lines for orientation. Figure 6.14(b) displays the solution after only a brief amount of simulation time has passed. It is naturally interesting to see whether discontinuity profiles change much during further time evolution. Figure 6.14(c) answers this question after one and two round-trips, respectively. Visually, the steepness of the solution is retained, and the number of points that are required to resolve the discontinuity has also remained stable. As an expected consequence of the clustering of the nodes towards element edges, points appear spaced closer together where the discontinuity touches an element boundary.

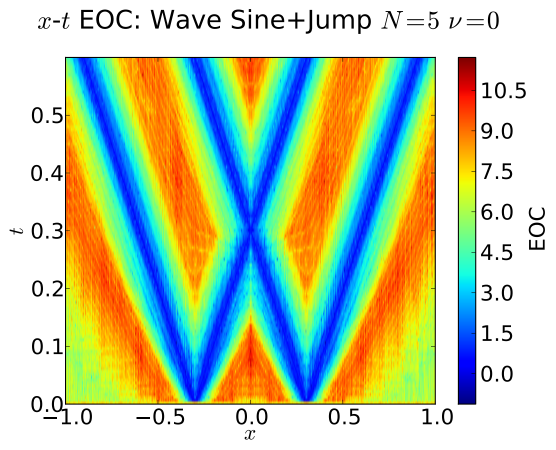
Figures 6.14(b) and 6.14(c) on page 150 appear to indicate that after a brief "settling" period the profile of the solution remains unchanged for the remainder of the calculation. Figure 6.15(a) on the preceding page sheds a new light on this observation and the observed increased sensitivity of the detector near element boundaries that was discussed above. It shows the maximum viscosity $\|\nu\|_{L^\infty}$ found anywhere on the domain, graphed versus simulation time. If the observation of "brief-settling-then-steady-state" were entirely true, then one would observe no sensor activations whatsoever after "settling" has occurred. This is not what is observed here. Instead, one sees a slowly decaying train of viscosity activation spikes. It turns out that each of these spikes coincides with a discontinuity crossing an element boundary. This again confirms the observation that the detection scheme is inhomogeneous in space, i.e. it judges solution smoothness differently depending on whether a discontinuity is located in the interior of an element or at its boundary. Since the sensor is only exposed to the non-smoothness for very short periods at a time, according to Figure 6.15(a) it takes considerable time ($t \gtrapprox 12$ in the example) and a number of viscosity "spikes" until a profile is achieved that does not trip even the overly sensitive version of the detector. It is to be expected that the profile is twice smoother than would be required if the oversensitivity did not exist.

As a last observation on the behavior of the method on this exceedingly simple problem, I would like to examine its interaction with the adaptive time stepper. The examples were computing using the well-known embedded Runge-Kutta method of third order by Bogacki and Shampine [1989] ("`ode23`" in Matlab). 6.15(b) on page 151 shows the adaptively-chosen time step Δt as a function of the step number. The stable advective time step is clearly visible, as is the initial "settling" period discussed above, along with a variety of time step reductions occurring along the way. Some of these coincide with element transitions of discontinuities, but the situation is more ambiguous (and noisier) than in the case of viscosity activations. The figure does make one thing amply clear, however: an artificial-viscosity-based shock capturing scheme using explicit time stepping must use time step adaptivity, or it will not be competitive.

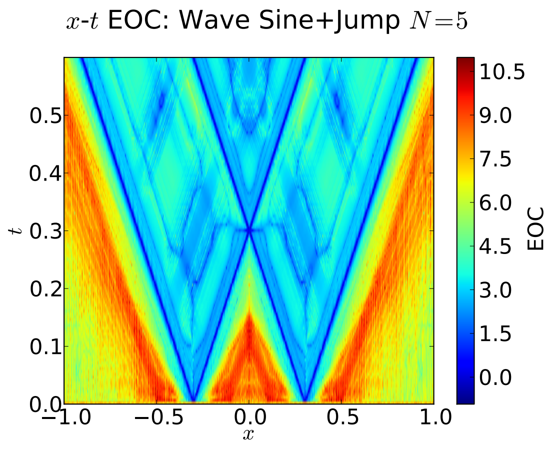
6.6.2 Waves: Shock Spreading and Spurious Coupling

The next, more complicated problem for which I examine the behavior of the proposed artificial viscosity is the wave equation, described in Section 6.3.2.

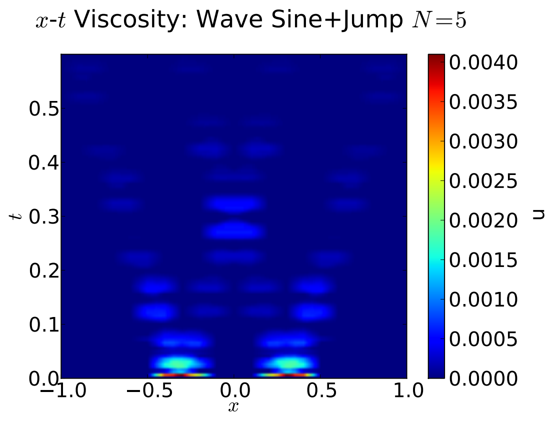
I would like to set the stage for my experimental results by considering the context of recent work by Cockburn and Guzmán [2008], who show (under a number of additional assumptions) that for a DG computation of a linear advection equation at second order using a second-order total-variation-diminishing (TVD) time discretization, pollution of the numerical solution by the shock by time T stays localized to an area of size $O(\sqrt{hT})$ ahead of and an area of size $O(\sqrt[3]{Th^2})$ behind the discontinuity. Although they only show this for a scalar advection equation, the wave equation (6.2) and its discretization may be



(a) EOC for the wave equation with a discontinuous initial condition *without* artificial viscosity.

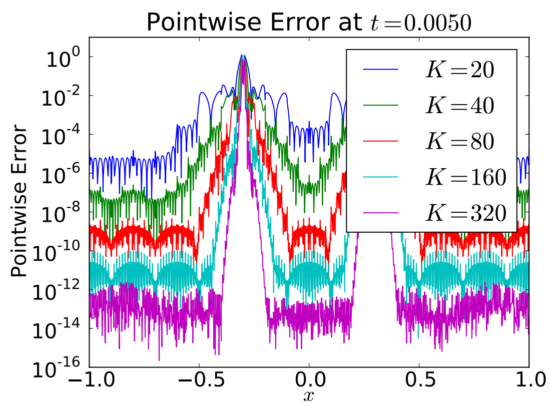


(b) EOC for the wave equation with a discontinuous initial condition *with* artificial viscosity.

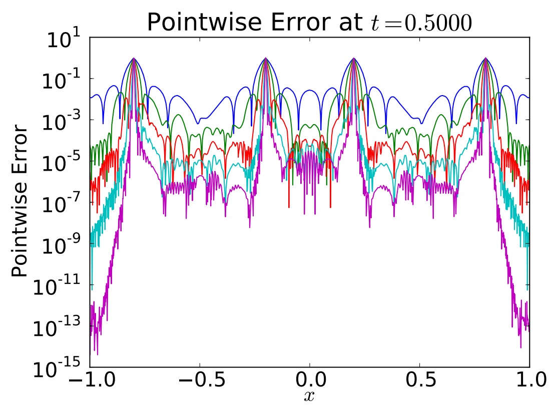


(c) Applied artificial viscosity for the example of Figure 6.16(b) in space and time.

Figure 6.16. Empirical order of convergence for the wave equation with discontinuous initial conditions.



(a) Pointwise error in space for the wave equation with artificial viscosity at a near-initial time–compare the bottom of Figure 6.16(b) on the previous page.



(b) Pointwise error in space for the wave equation with artificial viscosity at a near-final time–compare the top of Figure 6.16(b) on the previous page.

Figure 6.17. Spatial pointwise error for the wave equation.

transformed into two decoupled advection equations, and hence the result applies in this case as well.

I will study the pollution of the solution by examining its pointwise empirical order of convergence to the known analytic solution in space and time, starting from the initial condition

$$u(x,0) = 2 + \cos(5\pi x) + 4 \cdot \mathbf{1}_{[-0.3,0.3]}(x), \qquad v(x,0) = 0,$$

subject to Neumann boundary conditions, on a domain $\Omega = (-1,1)$ up to a final time $T = 0.6$, with a wave speed $c = 1$.

Figure 6.16 on the preceding page shows the resulting convergence plots, obtained with and without artificial viscosity. As expected through the work of Cockburn and Guzmán [2008], the inviscid DG scheme of Figure 6.16(a) achieves full convergence away from the discontinuities, but also shows a slowly-growing zone of non-convergence near the discontinuities, again matching predictions.

Unfortunately, results are not as favorable once artificial viscosity starts to act on the scheme. Outside the region that interacts with the discontinuities, convergence is roughly as before. However inside the interacting regions, convergence does improve again away from the discontinuity, but it does not recover the full order of the scheme. This underscores the importance of the wave equation as a test example for shock capturing schemes. Once the PDE is rewritten in as a system of first-order conservation laws (6.2), the single added viscosity of (6.3) induces a cross-coupling that appears to destroy accuracy. The plot of Figure 6.16(c) on page 154, showing the amount of viscosity applied in time and space, shows that very little viscosity suffices to degrade convergence. (The temporal oscillations in the figure stem from the fact that the solution is a standing wave by nature and therefore oscillatory in time.)

Note that such behavior *cannot* be observed in the advection equation, or, generally, any purely scalar conservation law, since these equations have only one characteristic wave, and hence the pollution caused by the artificial viscosity cannot spread, but propagates along with the solution. This might leads one to suggest an obvious “fix” for the issue: (6.2) can easily be transformed into characteristic variables, where it takes the form of two advection equations that only couple at the boundary, such that the issue disappears [Rieper, 2010]. As I have already discussed, proposing this is as a general remedy is however a bit disingenuous, as it cannot work properly in multiple dimensions. Another idea that one might have to try and avoid the reduction in accuracy is to use separate viscosities for each of the variables. According to my experiments, this does not help, as the cross-coupling of the system persists.

Next, it seems unlikely that this problem is specific to the artificial viscosity constructed in this chapter, or to discontinuous Galerkin methods, for that matter. It should be investigated whether *all* artificial viscosity schemes proposed so far in the literature suffer from this shortcoming.

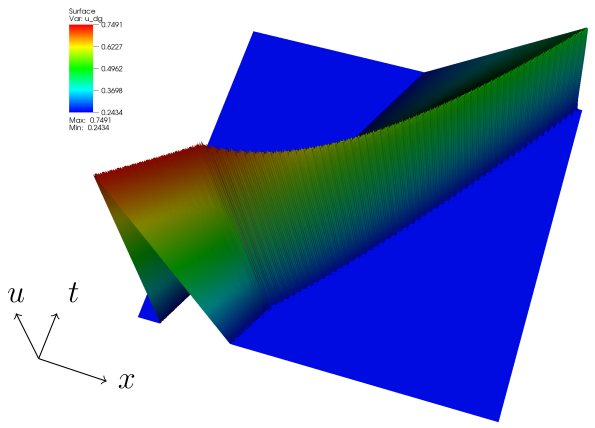


Figure 6.18. Space-time plot of the solution of Burgers' equation in 1D from the initial condition (6.21).

Further insight into the issue is available through Figure 6.17 on page 155, which shows the pointwise error in the numerical solution at an early (Figure 6.17(a)) and a late point in the time evolution of the solution (Figure 6.17(b)), for all mesh resolutions that were used to obtain the convergence information of Figure 6.16. In the near-initial situation, the influence of the discontinuities is clearly visible, but away from them, one observes convergence of the full order, where, as also mentioned by Cockburn and Guzmán [2008], the error is highly oscillatory. This good convergence is retained in the outer regions of the late-time graph, but in the center, post-interaction regions, it has broken down. It is nonetheless encouraging that convergence is still happening, albeit at a much-reduced rate.

6.6.3 Burgers' Equation

Moving on to the first nonlinear example of this sequence, Burgers' Equation, stated in Section 6.3.3, avoids the trouble described in the previous section by virtue of being a scalar conservation law. The purpose of introducing it here is to demonstrate the performance of the method on a simple, scalar, nonlinear example. It seems appropriate to mention at this point that the detector of this chapter, unlike other detectors proposed in the literature, such as the jump detector of Barter and Darmofal [2010], does not require positivity and can seamlessly treat zero crossings in the solution, if the underlying conservation law permits them.

I test the scheme on the domain $\Omega = (0, 150)$, with periodic boundary conditions and the initial condition given by

$$u(x,0) = \begin{cases} \frac{1}{4} & x \in \mathbb{R} \setminus (-10, 20), \\ \frac{x}{20} + \frac{3}{4} & -10 \leq x < 0, \\ -\frac{x}{40} + \frac{3}{4} & 0 \leq x < 20. \end{cases} \tag{6.21}$$

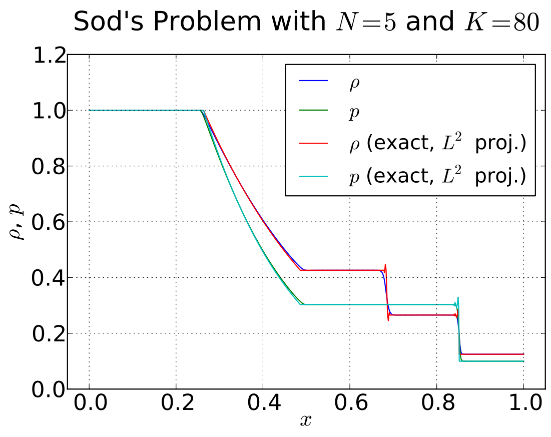
Figure 6.18 on the previous page shows the resulting numerical solution in space and time for polynomial degree $N = 5$ on $K = 80$ elements and demonstrates the good control the method exerts over spurious oscillations.

6.6.4 Euler's Equations in One Dimension

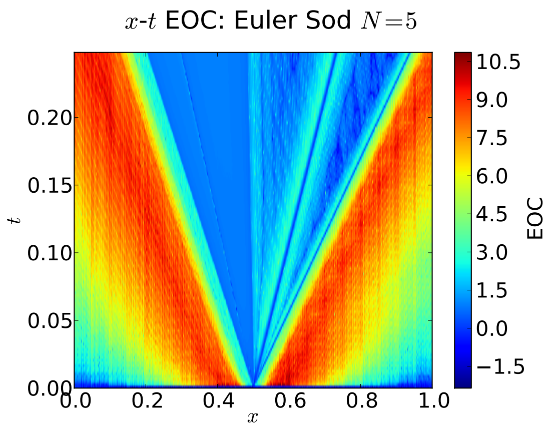
In this section, I will carefully examine the behavior of the artificial viscosity method introduced above on Euler's equations of gas dynamics, starting with the classical exact solution of the Riemann problem given by Sod [1978] as the first example.

Figure 6.19(a) on the following page shows computational results, again at polynomial degree $N = 5$ on $K = 80$ elements, in direct comparison with the (L^2 projection of) the exact solution, for the density ρ and the pressure p, at the final time $T = 0.25$ of the computation.

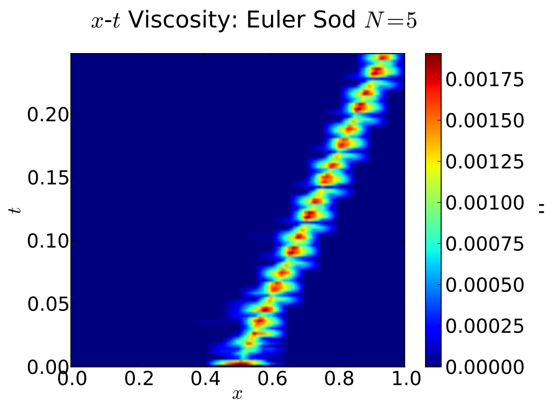
While the figure above gives an impression of the desired solution and a first impression



(a) L^2-projected exact and approximate numerical solutions of Sod's problem for polynomial degree $N = 5$ in $K = 80$ elements.

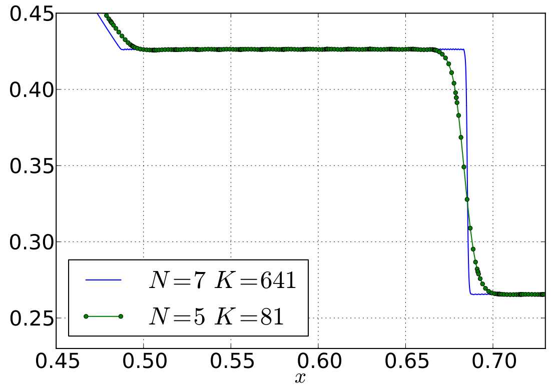


(b) Space-time diagram of the empirical order of convergence for Sod's problem, computed with artificial viscosity.

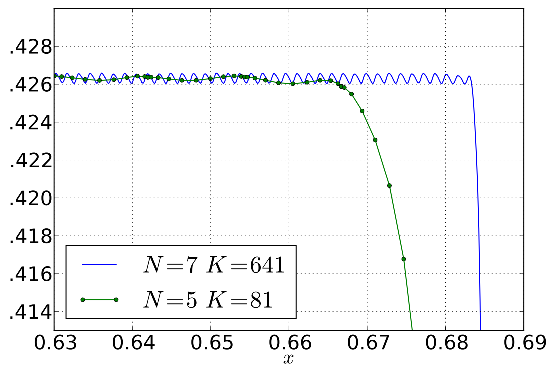


(c) Applied artificial viscosity for the example of Figure 6.19(b) in space and time.

Figure 6.19. Sod's problem with artificial viscosity: solution and x-t convergence.



(a) Close-up view of the contact discontinuity in Figure 6.19(a) on the previous page at low and high numerical resolutions. Interpolation nodes for the low-resolution case are shown as dots.



(b) Extreme close-up view of the tip of the contact discontinuity in Figure 6.20(a), at low and high numerical resolutions. Both cases exhibit spurious oscillation of the scale of one element. Interpolation nodes for the low-resolution case are shown as dots.

Figure 6.20. Element-scale oscillation exhibited by the artificial viscosity scheme.

of the performance of the method, it is perhaps more enlightening to examine an analog to the the convergence in space and time of Figure 6.16 on page 154 in the gas dynamics setting. Figure 6.19(b) on the preceding page provides this. As above, the computation was carried out at polynomial degree $N = 5$, at a variety of mesh resolutions ranging from $K = 20$ to 320 elements across the domain. Like in the linear case, convergence away from the shock region is good, while in the central, shock-interacting 'fan', it hardly exceeds order 1. In particular, it is worth noting that convergence along the profile of the smooth rarefaction wave is also no better than order 1. Given the results obtained for the wave equation, this is not very surprising, and it confirms that the issues observed on linear problems persist in the nonlinear case.

Figure 6.19(c) on the previous page, in analogy to Figure 6.16(c) for the wave equation, provides a view of the detector's reactions in space and time. Two observations may be made: First, only the genuine (self-steepening) shock in the example actually activates the detector, the contact discontinuity does not. This further confirms the 'settling' hypothesis pursued earlier. Second, the activation of the detector is non-constant in time. The presumed reason for this is, again, the detector's inhomogeneity in space.

	$N=4$	$N=5$	$N=7$	$N=9$	EOC
$h/1$	$9.982 \cdot 10^{-3}$	$7.934 \cdot 10^{-3}$	$6.522 \cdot 10^{-3}$	$5.567 \cdot 10^{-3}$	0.70
$h/2$	$5.442 \cdot 10^{-3}$	$4.231 \cdot 10^{-3}$	$3.395 \cdot 10^{-3}$	$2.921 \cdot 10^{-3}$	0.75
$h/4$	$2.945 \cdot 10^{-3}$	$2.219 \cdot 10^{-3}$	$1.778 \cdot 10^{-3}$	$1.568 \cdot 10^{-3}$	0.76
$h/8$	$1.548 \cdot 10^{-3}$	$1.166 \cdot 10^{-3}$	$9.488 \cdot 10^{-4}$	$8.329 \cdot 10^{-4}$	0.74
$h/16$	$8.087 \cdot 10^{-4}$	$6.006 \cdot 10^{-4}$	$5.121 \cdot 10^{-4}$	$4.598 \cdot 10^{-4}$	0.66
$h/32$	$4.207 \cdot 10^{-4}$	$3.111 \cdot 10^{-4}$	$2.806 \cdot 10^{-4}$	—	0.69
EOC	0.93	0.95	0.92	0.92	

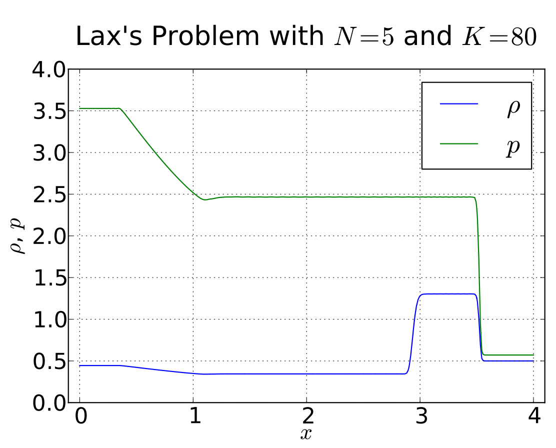
Table 6.1. L^1 error and convergence data for the Sod problem of the Euler equations of gas dynamics. "EOC" stands for the empirical order of convergence, obtained as a least-squares fit to the data.

A closer look at the numerical solutions in the poorly-converged region of 6.19(b) offers a revealing insight, shown in Figure 6.20 on the preceding page for a high-resolution case ($N = 5$, $K = 81$) and a low-resolution case ($N = 5$, $K = 81$). On the constant parts of the solution to the Riemann problem, I observe small "wrinkles". Figure 6.20(a) provides a sense of scale, while the extreme close-up of Figure 6.20(b) shows the phenomenon in detail. In both the high- and the low-resolution case, the oscillation's wave length roughly agrees with the size of an element. Further, it is remarkable that the magnitude of the oscillation appears to grow, rather than shrink, with increased resolution, which seems to indicate that convergence below the margin provided for by the oscillation might not occur. (Convergence will be examined in some detail below.) The phenomenon is observed on all constant areas that are inside the fan of characteristics emanating from the shock at time $t = 0$. So far, I do not understand the cause of this phenomenon, nor is it known whether there is a connection between these wrinkles and the reduced convergence observed in Section 6.6.2. One might speculate that, again, the spatial detector's spatial inhomogeneity is to blame.

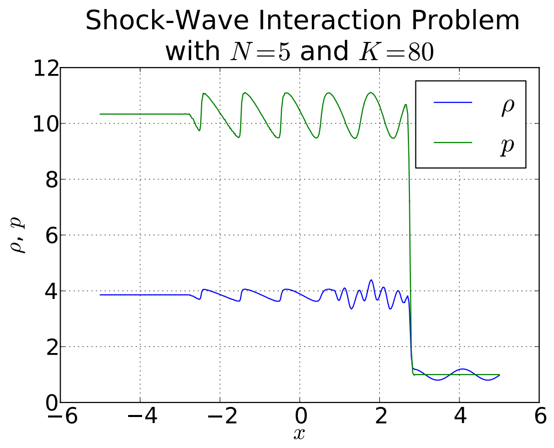
Beyond the spot testing conducted so far, I have also carried out a more comprehensive convergence study on the Euler equations applied to the Sod problem. The raw L^1 error data as well as empirical convergence order results obtained from least-squares fits are shown in Table 6.1. The data was gathered at a variety of polynomial degrees N and with $K = 20$ elements at the coarsest level, with uniform refinements thereafter. The data seems to support about a full order of convergence in $h = 1/K$. No improvement in convergence occurs as the order is increased. Further, the data supports less than a full order of convergence in N, indicating that an addition of elemental resolution at present is a more effective way of getting a more accurate solution than increasing the size of the local approximation spaces, especially considering that the computational complexity grows superlinearly in N. At the resolutions examined, the influence of the oscillations ("wrinkles") observed above does not appear to have contributed a significant part of the error–given their observed behavior in response to resolution changes, they would likely have represented a "bottom" to convergence at some fixed error magnitude. That issue aside, the observed convergence data appears to be as good as one might reasonably expect. While convergence of higher order would course be desirable, the method as it presently stands is not designed to be able to achieve this. Through some experiments on polynomials, I have reason to believe that convergence of order one in N is achievable and thereby a goal for future research.

In addition to the problem of Sod [1978], which has furnished the basis for all tests so far, I have also conducted tests using other available solutions for the Euler equations. One such solution that is rather similar to the Sod problem is that of Lax [1954] in that it also originates from a Riemann problem. Figure 6.21(a) on the following page demonstrates that the scheme can successfully compute a correct solution to the problem. Lax's problem prominently features a contact discontinuity, which is prone to smearing, as was discussed above. The contact discontinuity in the figure appears somewhat more smeared than the Sod contact discontinuity at a similar scale.

A further basic benchmark test for the method applied to the one-dimensional Euler equations was proposed by Shu and Osher [1989, Example 8] to highlight the need for high-order methods in properly capturing the interaction of shocks with smooth wave-like



(a) Approximate numerical solution for density and pressure of Lax's problem for polynomial degree $N = 5$ in $K = 80$ elements.



(b) Approximate numerical solution for density and pressure of a shock-wave interaction problem for polynomial degree $N = 5$ in $K = 80$ elements.

Figure 6.21. Solutions of classical test problems for the Euler equations using the artificial viscosity scheme.

features. Considering the gathered convergence data, I cannot claim that the method is of high order away from discontinuities once such areas enter the domain of influence of a location where artificial viscosity was applied. Nonetheless, it is still instructive to see that the method is capable of keeping the computation stable and delivering a correct result at least in the "picture norm", as evidenced by Figure 6.21(b). This example is commonly considered challenging, and it is encouraging that the method is able to stabilize the computation and give a meaningful result without excessive smearing.

As a final validation of the detector's design on the Euler equations, it is important to examine whether it will recognize smooth solutions and leave them untouched, preserving high-order accuracy. I have tested this using the smooth isentropic vortex test case of Zhou and Wei [2003] with the result that as soon as sufficient resolution is available, the detector does not activate anywhere at any time during the solution process.

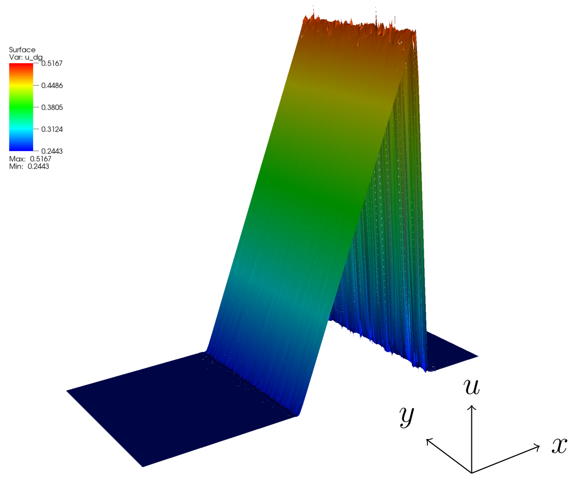


Figure 6.22. Solution of the *one-dimensional* Burgers equation on a two-dimensional computational domain at polynomial degree $N = 5$ on $K \approx 600$ triangles.

6.6.5 Initial Experience in Two Dimensions

One of the core goals in the design of the smoothness detector was to be seamlessly generalizable to multiple dimensions. This is the case, however as discussed in Section 6.4.3, several ambiguities arise in this generalization that must be settled among a number of choices. Nonetheless, as planned, the scheme can be run even with ad-hoc choices made in the two-dimensional application of the scheme, and provides results that are comparable, if somewhat worse, than those in one dimension. Figure 6.22 shows the result of a two-dimensional treatment of the one-dimensional Burgers' problem set forth in Section 6.6.3. As can be seen in the figure, the method does not yet dampen oscillations as effectively as it does in one dimension, making it challenging to keep computations based on the Euler equations stable as solutions experience undershoots below zero in pressure and density. Such issues typically express themselves in a step size underflow of the adaptive time stepper, which detects that no progress can be made without advancing the solution into an invalid state. A proposed fix for this is the detection of this situation and the insertion of "pure smoothing" cycles into the solution process to circumnavigate such issues, however no experimental data is yet available to determine whether this provides the needed "breathing room" to stabilize gas dynamics computations.

6.7 Conclusions and Future Work

What sets the shock capturing method of this chapter apart is its focus on explicit, local, GPU-suited calculation in the context of discontinuous Galerkin methods, as explained in Section 6.2. Despite Section 6.6's focus on issues that still exist, I would contend that in this niche the method is already reasonably successful and merits further study. Its construction introduces several new concepts, such as a more precise interpretation of the correspondence between polynomial decay and smoothness, as well as methods like skyline pessimization, baseline decay, and P^1 viscosity smoothing. I should note that most of these ideas arose arose from discussions I had with Tim Warburton over the course of a few months.

The study of the method's behavior on simple problems (such as linear waves and transport) was–in my opinion–quite revealing, and it should be investigated in how far other shock capturing methods are susceptible to the same problems. It would further be interesting to see if some of the issues observed (such as the high sensitivity of the detector at element transitions) can be remedied in some fashion.

On more complicated nonlinear problems, results were, in my estimation, encouraging. For example, the method manages to stabilize the computation of the shock-wave-interaction example and other important benchmarks, without introducing excessive smoothness. Further investigation, using the rich pool of tests available in the shock capturing literature [ASC Flash Center, 2009, Slater et al., 2009, Stone, 2009, Woodward and Colella, 1984] will doubtlessly give further insight into the method's strengths and weaknesses as well as help to further improve it.

CHAPTER SEVEN

The Vlasov-Maxwell System and DG

7.1 Introduction

The Vlasov-Maxwell system describes the evolution of collisionless plasmas and is applicable in, e.g., the modelling of particle accelerators, laser-matter interaction, and certain regimes occurring in nuclear fusion. Fundamentally, it models the interaction between moving charge carriers and the electromagnetic field.

Let Ω be a bounded, polyhedral, open domain of interest. The Vlasov-Maxwell system consists of two parts. First, Maxwell's equations describe the evolution of the electromagnetic field. In the formulation used here, Maxwell's equations describe the time evolution of the electric field $E(x,t)$ and the magnetic field $H(x,t)$ for $x \in \Omega$:

$$\partial_t E - \frac{1}{\epsilon}\nabla \times H = -\frac{j}{\epsilon}, \tag{7.1a}$$

$$\partial_t H + \frac{1}{\mu}\nabla \times E = 0, \tag{7.1b}$$

$$\nabla \cdot E = \frac{\rho}{\epsilon}, \tag{7.1c}$$

$$\nabla \cdot H = 0. \tag{7.1d}$$

ϵ is the (electric) permittivity of the material under consideration, and μ is the (magnetic) permeability. In this case, both will assume their known vacuum values $\epsilon = \epsilon_0$ and $\mu = \mu_0$.

In addition to the EM fields, the system models the evolution of a *number density* $f(x,p,t)$ on a phase space $\Omega \times \mathbb{R}^3$ in time t, where the variable p represents particle momentum. (x,p)-space viewed as one domain is also called *phase space*. The behavior of f is governed by a transport equation termed the *Vlasov Equation*:

$$\partial_t f + v \cdot \partial_x f + L \cdot \partial_p f = \langle \text{Sources} \rangle - \langle \text{Sinks} \rangle. \tag{7.2}$$

The Vlasov equation is similar in appearance to the *Boltzmann Equation*, which accounts for the occurrence of collisions by source and sink terms on the right hand side of the equation. Conversely, (7.2) does not account for collisional effects.

The quantity L in (7.2) represents a force acting on the particles. Here, this is the *Lorentz force*

$$L(E,H) = q(E + v \times \mu H). \tag{7.3}$$

Here and throughout this chapter, I choose to work in terms of momentum p rather than particle velocity v to facilitate the inclusion of relativistic effects. One important consequence of this choice is that the second component of phase space is truly unbounded, rather than bounded by the speed of light $c = 1/\sqrt{\mu\epsilon}$. When particle momentum and species rest mass m_0 are known, the particle velocity is found as

$$v(p) = \frac{cp}{\sqrt{p \cdot p + c^2 m_0^2}}.$$

The Lorentz force establishes a coupling from the electromagnetic fields to the number density. The reverse coupling from number density to the EM field is accomplished through the source terms j and ρ already present in (7.1). These can be computed from the known number density f by using the particle species charge q.

$$\rho(x,t) = q \int_{B(O,c)} f(x,p)\,\mathrm{d}v, \tag{7.4a}$$

$$j(x,t) = q \int_{B(O,c)} v f(x,p,t)\,\mathrm{d}v, \tag{7.4b}$$

where $B(O,c)$ represents the open ball around the origin O with radius c.

Observe that, at least as far as the formulation is concerned, it is trivially possible to account for the motion of multiple species of particles simply through the introduction of additional number densities f. One common case is the simulation of an electron cloud under the assumed existence of a homogeneous, neutralizing background of ions with

$$\int_\Omega \rho_{\text{ion}}\, dx = \int_\Omega \rho_{\text{electron}}\, dx.$$

Because of the enormous weight difference between ions and electrons, it is safe to model the ions as immovable. They can then be accounted for in (7.4) by setting

$$\rho(x,t) = q \int_{B(O,c)} f_{\text{el}}(x,p)\,\mathrm{d}v + \rho_{\text{ion}}, \tag{7.5}$$

$$j(x,t) = q \int_{B(O,c)} v f_{\text{el}}(x,p,t)\,\mathrm{d}v, \tag{7.6}$$

where I note that (7.6) is unchanged from the case of pure electron transport.

7.1.1 Boundary Conditions

Both the electromagnetic fields E and H and the density f must be endowed with appropriate boundary conditions to form a well-posed system.

For Maxwell's equations, one may for example adopt one of the following common field boundary conditions on various subsets of the boundary $x \in \partial\Omega$:

- Perfect Electrical Conductor, $\hat{n} \times E = 0$, $\hat{n} \cdot H = 0$, or
- Perfect Magnetic Conductor, $\hat{n} \cdot E = 0$, $\hat{n} \times H = 0$, or
- Absorbing boundary conditions, realized as outgoing characteristic BCs or a perfectly matched layer.

In what follows, I will approximate momentum space as truly infinite, so far-field momentum boundary conditions are not needed. Near geometric obstacles or domain boundaries, one does however need joint boundary conditions in x and p for the density f in the Vlasov equation at points where $x \in \partial\Omega$ and $\hat{n} \cdot p < 0$, where $\hat{n}$ is the unit surface normal. A few common examples include

- Specular Reflection: $f(x, p, t) = f(x, p - 2(p \cdot \hat{n})\hat{n}, t)$,
- Absorption: $f(x, p, t) = \alpha f(x, p - 2(p \cdot \hat{n})\hat{n}, t)$, $0 \le \alpha < 1$,
- Emission: $f(x, p, t) = g(x, p, t)$, $g \ge 0$,

and combinations thereof.

7.2 Discretizing the Electromagnetic Field

One of the main motivations for this work is that the discontinuous Galerkin method (see Section 2.1) provides a very practical discretization for the system of Maxwell's equations (7.1).

I refer to 2.1 for an introduction to discontinuous Galerkin discretizations and restrict myself here to noting that I use a 'strong-form' DG formulation [Hesthaven and Warburton, 2002, 2007] with an upwind flux due to Mohammadian et al. [1991]:

$$\hat{n} \cdot (F_N - F_N^*) := \frac{1}{2} \begin{bmatrix} \{Z\}^{-1}\hat{n} \times (Z^+ [\![H_N]\!] - \hat{n} \times [\![E_N]\!]) \\ \{Y\}^{-1}\hat{n} \times (-Y^+ [\![E_N]\!] - \hat{n} \times [\![H_N]\!]) \end{bmatrix}. \tag{7.7}$$

I have employed the conventional notations for the cross-face average $\{u\} := (u_N^- + u_N^+)/2$

and jump $[\![u]\!] := u_N^+ - u_N^-$. For concise notation, I use the intrinsic impedance $Z := \sqrt{\mu/\epsilon}$ and admittance $Y := 1/Z$. Applying the principles of Section 2.1, I arrive at a discontinuous Galerkin scheme.

7.3 Discretizing the Density

While the discretization of the electromagnetic fields as discussed above follows a known standard procedure, the main challenge in a computational treatment of the Vlasov-Maxwell system lies in dealing with its transport part: Because the plasmas under consideration are so rarefied that collisions do not play a significant role, the Vlasov-Maxwell system as a continuum model allows for particles with differing momenta at a single spatial location, as evidenced by the presence of separate momentum axes in the pre-image space of f.

7.3.1 The Eulerian approach

If one chooses a conventional Eulerian fixed-mesh approach to the discretization of the density f, one is automatically faced with the problem of high dimensionality. There are three spatial and three momentum dimensions to be discretized, resulting in a total of six dimensions. Many three-dimensional problems require the world's largest supercomputers to achieve adequate resolution. As a result, a six-dimensional mesh can be quite prohibitive. Even a spatially 2D problem still has effectively four dimensions, which severely limits the resolution.

Eulerian approaches are faced with two further problems: First, occupancy of phase space by particles is often low except for a localized (in space and momentum) region of interest, and hence f mostly evaluates to zeros or near-zeros. Second, compared with the Boltzmann equation [see, e.g., Narayan and Klöckner, 2009], the Vlasov equation lacks a collision term. While the collision term contributes its own set of issues that a discretization has to solve, it does add a certain amount of regularity. This is helpful because the Vlasov equation tends to form many fine features in its solution (a phenomenon known as "filamentation"), each of which has sharp gradients at its boundaries. Sharp solution gradients on the other hand can cause instabilities and spurious Gibbs-type oscillations, which must be controlled by some method. (see Chapter 6 for somewhat related work)

These effects and, particularly, their strength, makes it seem unlikely that a non-adaptive Eulerian scheme can yield a successful discretization of the Vlasov-Maxwell system. Together with Akil Narayan, I have attempted such a discretization, using a tensor product of regular discontinuous Galerkin with his generalized Wiener rational functions [Narayan and Hesthaven, 2009] as a basis. We quickly encountered the issues mentioned above and have yet to move past them in our efforts.

7.3.2 Particles and the Lagrangian approach

High dimensionality and lack of smoothness make direct (Eulerian) simulation of particle transport demanding in terms of both processing power and memory. Lagrangian methods offer a compelling alternative that may help overcome the issues explained in the previous section, but they again come with their own set of disadvantages.

The first and perhaps most obstructive such issue is that since the scheme for Maxwell's equations is in all likelihood an Eulerian one, one is necessarily faced with the problem of mapping back and forth between two representations. Second, one is faced with a choice of object with which to approximate the density f. Objects having a true, time-dependent spatial extent invariably encounter some trouble in the Lagrangian-Eulerian remapping as the Vlasov equation may badly deform them over time. To avoid having to 're-normalize' the Lagrangian object, spatially point-shape particles have been a dominant choice for the Lagrangian approximants of f. Attempts to discretize plasma physics as described by the Vlasov-Maxwell system through particle methods date back to the early days of computing [Birdsall and Langdon, 1984, Hockney and Eastwood, 1988]. To emphasize the mixed Lagrangian-Eulerian character of such methods, they are often termed '*Particle-in-Cell*' (or *PIC*) methods.

Particle discretizations present perhaps the fewest immediately insurmountable computational obstacles, but they are by far not free from problems. First, a finite sum of point-shape objects is not particularly suited to approximating a density given in a continuum description–one is forced to sample from the distribution, where the analogy to the theory of probability is not spurious, but actually desired. Second, the sampling introduces sampling noise. Third, the sampling introduces a question of resolution–which parts of f should be sampled with how many particles? Fourth, most methods for the approximation of PDEs (and in particular the high-order DG methods I am targeting here) depend on some smoothness in their approximated fields. Particles are less smooth and thereby worse than the discontinuities one might have faced in an Eulerian discretization and thereby introduce a further form of noise into the method.

The remainder of this chapter summarizes my efforts to overcome these issues in the described context of a DG method. The challenge is to find a particle model and a coupling that works with high-order, unstructured DGTD field solvers. This is an unsolved problem, consisting of two parts:

- computing (or "depositing") a suitable DG-discretized current density based on modeled particle movement, and

- computing (or “interpolating”) the Lorentz force for each particle location based on the computed fields.

7.3.3 Rationale and Strategy for High-Order Unstructured PIC

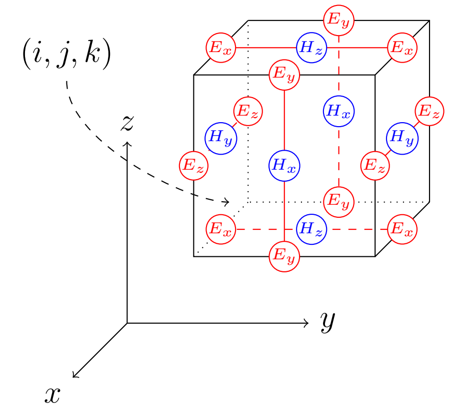
As Section 7.4 will show, most existing work on PIC focuses on low-order methods on Cartesian grids. This section aims to explain why I seek to adapt particle-in-cell methods to the discontinuous Galerkin method, which I intend to operate at high order on unstructured grids.

The main advantage of a high-order-accurate numerical scheme is that, for a small given desired accuracy, they are more work-efficient than corresponding low-order schemes [Hesthaven et al., 2007], assuming a certain amount of solution smoothness. Given the noise and non-smoothness issues inherent in particle-based methods, it is not obvious that it makes sense to ask for the small accuracies at which high-order methods excel, and some work will have to be invested below to control these issues. But even aside from this, a clear benefit becomes apparent for a certain, significant subset of applications. Suppose there are well-separated regions of (mainly) electromagnetic and (mainly) particle-driven activity. At some distance from the particle-focused region, accurate long-range wave propagation will become more relevant to simulation success than noisy fine-scale interactions. In such a scenario, I hope to reap the full benefit of high-order DG: accurate approximation by relatively few points per wavelength, and less introduced phase error and wave discretization noise.

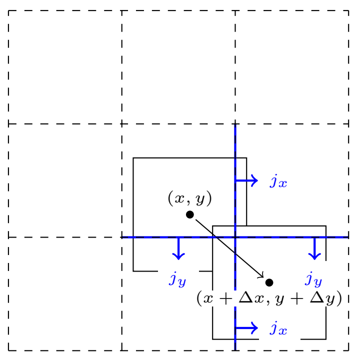
In comparison to the use of high-order methods, the case for particle-in-cell methods on unstructured grids is much clearer. In many problems of interest, such as in the simulation of particle accelerator cavities (cf. Section 7.10), the somewhat complicated and curvilinear surrounding geometry is *the* decisive factor in how electromagnetic waves propagate and therefore needs to be modeled accurately. It will become apparent below that non-Cartesian structured meshes, in addition to being much harder to obtain than, e.g. simplicial unstructured ones, encounter many of the same problems of the latter. It is therefore safe to consider only fully unstructured and Cartesian structured grids as alternatives. The latter type of grid approximates complicated, curved geometry with at best first order in the local mesh size, and thus makes accurate boundary representation very expensive in multiple dimensions, unless some sort of cut-cell scheme is employed, which may in turn introduce problems such as prohibitively small time step limitations. Unstructured (and potentially curvilinear) meshes manage to decouple time step and local mesh size from the requirement of accurate boundary representation, and thus promise a significant cost savings.

In addition to facilitating accurate representation of boundary geometry, unstructured grids have a further advantage. It was mentioned above that one crucial factor to the success of a PIC simulation is proper management of resolution. Unstructured grids make it comparatively easy to increase mesh resolution where it is needed. This can be achieved a-priori with great ease if the regions of interest are known ahead of time or, with little more effort, adaptively during the run time of the method.

As the next section will show, only limited work has so far been done on bringing PIC to high-order methods on unstructured meshes. I therefore pursue a mostly exploratory strategy, focused on finding and evaluating the choices available for both deposition and force interpolation. I further aim to investigate how they interact with a chosen density representation. Once a number of choices are available, one encounters the question of how to choose between them. Since convergence theory for particle-in-cell methods is rudimentary even in the structured, low-order case [Victory and Allen, 1991], I will first pursue an evaluation of methods based on their performance on a number of test problems. Since there are very few known, exact solution that exercise the full complexity of the



(a) Mesh arrangement for Yee's Finite-Difference Time-Domain scheme. [Yee, 1966]



(b) Villasenor-Buneman assignment scheme for current densities in FDTD-PIC [Villasenor and Buneman, 1992]

Figure 7.1. Aspects of FDTD-PIC.

Vlasov-Maxwell system, in many cases I will be evaluating the performance of a scheme by how well it agree with known facts about the physics of the problem, as I will discuss in Section 7.9. The further strategy will then be to choose a promising algorithm and then, if necessary, try to improve its figures of merit by fine-tuning its details. The method obtained from this procedure is obviously only as good as the tests I perform on it, and therefore I will spend significant effort on assembling a reliable evaluation capability.

7.4 A Brief, Incomplete Survey of Prior Work

Traditionally, the most successful PIC methods have been those that combine Yee's [1966] explicit time-domain, staggered-grid centered-difference method for Maxwell's equations (see Figure 7.1(a)) with a charge deposition scheme by Villasenor and Buneman [1992] that assigns current densities based on the amount of charge passing through a grid boundary (see Figure 7.1(b)). This scheme is simple, fast, well-studied and reasonably well-validated. It achieves perfect conservation of energy and charge (see Section 7.5 for a description of the latter issue), is, aside from sampling error, first-order accurate in j and ρ and second-order accurate in its field approximation. In the setting of accelerator simulation (see Section 7.10), one particularly damaging consequence of the low-order approximation and hence the mismatch between particle and wave speeds causes a phenomenon termed *numerical Cherenkov radiation* [see, e.g., Zagorodnov and Weiland, 2005, and references therein]. In addition, even this low-order scheme can suffer from the noise generated by the non-smooth particle approximations it is often used with, resulting in "*grid heating*" [Rambo, 1997]. In summary, FDTD-PIC's applicability is significantly hampered by its geometric inflexibility and its poor approximation properties. Both shortcomings provide motivation for my high-order unstructured work.

One step towards mitigation of phase error and towards the run-time adaptation of mesh resolution was made by Gjonaj et al. [2006], who have generalized the FDTD scheme above to include high-order discontinuous Galerkin approximations on non-conforming meshes, retaining exact charge conservation in the process. Their article provides compelling evidence for the use of high-order schemes in PIC methods, but does not manage to shed the geometric restriction to Cartesian meshes.

The similar bodies of work by Campos Pinto et al. [2008] and Candel et al. [2009] are perhaps closest in spirit to my declared goal of high-order unstructured PIC, however with one important difference: In their work, the particles interacting with a high-order unstructured discretization are represented as point-shape entities. While they do manage to maintain good conservation of charge (again, see Section 7.5), the noise emanating from the point particles is significant and endangers the validity of the simulation unless some noise control is applied.

My various approaches to the problem of unstructured high-order PIC as detailed below build upon the work of Jacobs and Hesthaven [2006] and Jacobs et al. [2006]. I let

$$f(x,p,t) = \sum_{n=1}^{P} \frac{1}{r_n^d} S\left(\frac{x-x_n}{r_n}\right) \delta(p-p_n), \tag{7.8}$$

and in order to mitigate the generated noise, I following their work in deviating from the common choice $S \equiv \delta$ and giving each particle a non-zero extent expressed by the *shape function*

$$S(x) := \begin{cases} \frac{1}{\mathcal{N}}\left(1-||x||^2\right)^\alpha & ||x|| < 1, \\ 0 & \text{otherwise,} \end{cases} \tag{7.9}$$

This choice is justified by findings of Jacobs and Hesthaven [2006], who compared this and many other shapes in an empirical study. The normalization $\mathcal{N}$ in S is chosen such that $\int_{\mathbb{R}^d} S = 1$. In addition, this shape allows the choice of an exponent parameter α allowing a continuum of particle shapes with varying locality and smoothness. Figure 7.2 shows a view of S for $\alpha = 2$. A few features of S are summarized below:

- S is a (truncated) polynomial, which is reasonable in a polynomial approximation of j_N and ρ_N.
- $S \in C^{\alpha-1}$ for $\alpha \geq 1$.
- S is compactly supported.
- High values of the exponent α improve particle localization, but make the particle more difficult to resolve in both interpolatory and L^2-projection methods.

Further observe that S (and in general any scheme admitting a nonzero particle extent) requires a choice of width r_n of the nth particle. It is non-obvious what should be chosen for this particle width, as it is imperative for proper approximability that r_n scale with the local mesh size. On the one hand, a particle is almost guaranteed to encounter cells of

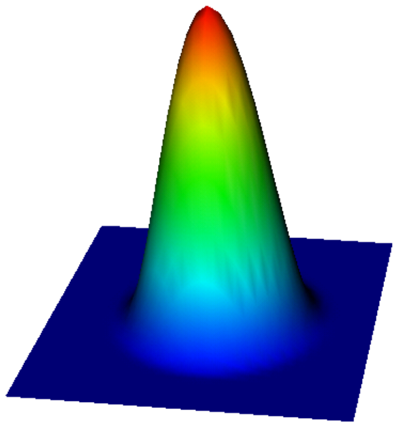


Figure 7.2. A three-dimensional view of the polynomial particle shape function of (7.9) with $\alpha = 2$.

various sizes throughout its life. On the other hand, the size of a particle cannot be changed during simulation without violating the consistency of the simulation (see, again, Section 7.5).

7.5 Ensuring "Charge Conservation"

In PIC literature, the somewhat unsound name "*charge conservation*" is attached to the non-occurrence or near-non-occurrence of inconsistencies between the charge density as represented in simulation state by the particle discretization and the charge density as obtained by (7.1c), i.e. the non-violation of the equality

$$\epsilon \nabla \cdot E(x,t) = q \int_{\mathbb{R}^d} f(x,p,t)dv \quad \text{for all } x, t. \tag{7.10}$$

Taking the divergence of Ampére's law (7.1a) yields the continuity equation

$$\partial_t \rho + \nabla \cdot J = 0. \tag{7.11}$$

By taking the time derivative of (7.10), substituting the Vlasov equation in the right hand side, and using (7.11) on the left hand side, it is easy to see that a solution to the Vlasov-Maxwell system satisfies (7.10) for all time if (7.10) is satisfied by the initial condition.

Available charge-conserving algorithms include conventional FDTD-PIC [Villasenor and Buneman, 1992], higher-order DG variants thereof [Gjonaj et al., 2006], methods on Cartesian grids having particles with extent [Esirkepov, 2001] and the point-shape unstructured PIC methods [Campos Pinto et al., 2008, Candel et al., 2009]. For particles with extent on unstructured or high-order discretizations (or both), methods with this property are, to the best of my knowledge, unavailable at the time of this writing.

Charge conservation is a critical property. If it is not satisfied, pollution in the propagating and non-propagating parts of the electric field may be left behind in the path of the particle. The non-propagating parts are perhaps the most damaging, as they can add up in place and add a significant spurious contribution in the Lorentz force (7.3). The propagating parts of charge conservation error are somewhat less damaging–so much less so, in fact, that turning the non-propagating parts into propagating parts has been proposed in the literature as a fix for non-charge conserving methods. I will discuss this fix in Section 7.5.2.

Among the schemes discussed below, marked differences exist in how well (7.10) is preserved. Based on my experience, I find it unlikely that an exactly charge conserving method of the kind of Villasenor and Buneman [1992] will be proposed for extent-bearing particles on unstructured grids. For high-order methods, in particular in view of Section 7.7.2, obeying (7.10) to the order of the scheme is conceivable, however. In absence of a scheme that natively obeys (7.10), various fixes have been conceived which have been cast in a common framework by Munz et al. [1999, 2000]. Out of the fixes yielded by their framework, the projecting elliptic and the hyperbolic ones are likely to be the most relevant in a time-explicit setting. Both are briefly explained in the two following subsections.

7.5.1 Divergence Cleaning by Helmholtz Projection

Suppose the current state of the solver involves a "polluted" electric field $\tilde{E}$, and view E to be an as yet unspecified nearby "clean" field obeying $\nabla \cdot E = \rho/\epsilon$. One seeks to "clean up" non-propagating components of the electric fields, which, ignoring the effects of discretization, are irrotational and thereby the gradient of a potential ϕ. One therefore writes the polluted field as

$$\tilde{E} = E + \nabla\phi, \tag{7.12}$$

where ϕ is a potential belonging to the 'misplaced' charge. Taking the divergence of this identity and solving for ϕ yields a Poisson Equation for ϕ:

$$\nabla^2\phi = \nabla \cdot \tilde{E} - \rho/\epsilon, \tag{7.13}$$

which is solved assuming Dirichlet conditions $\phi = 0$ on $\partial\Omega$. To conclude, (7.12) can be rearranged to find the "nearby" field $E = \tilde{E} - \nabla\phi$ which obeys $\nabla \cdot E = \rho/\epsilon$.

This correction technique has the distinct advantage that, whenever it is applied, it produces an electric field exactly obeying the divergence constraint. However, in doing so, it adds to the fully hyperbolic (and thereby parallel-computation-friendly) Vlasov-Maxwell system a distinctly less parallelization-friendly elliptic component (7.13). It should also be considered that the step from $\tilde{E}$ to E is a relatively violent modification. Thus if magnetic effects are to play a significant role in the problem under consideration, this method is not suitable. Also observe that fields surrounding fast-moving (and especially relativistic) charges gain a significant magnetic component after they are Lorentz-transformed to the laboratory frame [Jackson, 1998], marking another case where this divergence correction method is not suitable.

This technique is one out of a few proposed for divergence cleaning in the work of Munz et al. [1999, 2000].

7.5.2 Hyperbolic Divergence Cleaning

If one subscribes to the point of view that propagating disturbances in the electromagnetic field are "less bad" than stationary ones, one might stumble on the idea of turning the former into the latter by adding a characteristic to the Maxwell's system. Using a fixed wave speed multiplier $\chi > 1$, the following hyperbolic system achieves just that:

$$\partial_t E - \frac{1}{\epsilon}\nabla \times H + \chi \nabla \phi = -\frac{j}{\epsilon} \tag{7.14a}$$

$$\partial_t H - \frac{1}{\mu}\nabla \times E = 0 \tag{7.14b}$$

$$\partial_t \phi + \chi(\nabla \cdot E - \rho/\epsilon) = -\kappa\phi \tag{7.14c}$$

This hyperbolic system admits the characteristic velocities c $(2\times)$, $-c$ $(2\times)$, χc $(1\times)$, $-\chi c$ $(1\times)$, and 0 $(1\times)$. In comparison with the unmodified Maxwell's system (7.1), two characteristics of speed $\pm\chi c$ have appeared, and one non-propagating mode has disappeared. An alternate point of view is that an additional "cleaning wave" state variable ϕ has been joined to the system, and a wave equation in E and ϕ has been added.

The essence of (7.14) is captured by (7.14c): A divergence error $\nabla \cdot E - \rho/\epsilon$ present in the solution state acts as a source term for the "cleaning" variable ϕ and is turned into a wave propagating with speed χc, achieving what one set out to do–turning static divergence error into fast-propagating noise. In addition to propagation, the right hand side of (7.14c) includes a decay term with a coefficient $\kappa \geq 0$, whose declared goal it is to dissipate out the divergence error as it propagates. The decay term is used in this form by Jacobs and Hesthaven [2006].

(7.14) as a hyperbolic system is amenable to discretization by a discontinuous Galerkin method, in particular an upwind flux in analogy to (7.7) can be derived by standard methods [see Jacobs and Hesthaven, 2006].

Unlike Helmholtz-projection-based cleaning, this technique is usable even in the case of relativistically fast particles, and it does not lose the hyperbolic character of the system. It does come at a significant cost: To compute the right-hand side of (7.14c), a deposition of ρ is necessary, which is not the case for the ordinary Vlasov-Maxwell system. Further, since the system's largest characteristic velocity is multiplied by a factor of $\chi > 1$, the largest time step that can be taken in an explicit setting is reduced by just this factor.

Again, this technique is one out of a few proposed for divergence cleaning in the work of Munz et al. [1999, 2000].

7.6 Time Discretization

If one inserts a single-particle density $f(x,p,t) = \delta(x - x_n(t))\delta(p - p_n(t))$ into (7.2), multiplies by a test function, and integrates over all of phase space, one obtains

$$\begin{aligned}
0 =& \int \partial_t(\delta(x - x_n)\delta(p - p_n))\phi + v \cdot \partial_x(\delta(x - x_n)\delta(p - p_n))\phi \\
&\quad + L \cdot \partial_p(\delta(x - x_n)\delta(p - p_n))\phi \,\mathrm{d}x\,\mathrm{d}p \\
=& \int -\partial_t x_n \delta'(x - x_n)\delta(p - p_n)\phi - \partial_t p_n \delta(x - x_n)\delta'(p - p_n)\phi \\
&\quad + v \cdot \delta'(x - x_n)\delta(p - p_n)\phi \\
&\quad + L \cdot \delta(x - x_n)\delta'(p - p_n)\phi \,\mathrm{d}x\,\mathrm{d}p \\
=& - \partial_t x_n \partial_x \phi(x_n, p_n) - \partial_t p_n \partial_p \phi(x_n, p_n) + v(p_n) \cdot \partial_x \phi(x_n, p_n) \\
&\quad + L \cdot \partial_p \phi(x_n, p_n).
\end{aligned}$$

Sorting the last equation by coefficients of $\partial_x \phi(x_n, p_n)$ and $\partial_p \phi(x_n, p_n)$, one obtains the equations of particle motion

$$\partial_t x_n = v(p_n), \tag{7.15a}$$

$$\partial_t p_n = L(E(x_n), H(x_n)). \tag{7.15b}$$

If a method-of-lines approach is followed for the discretization of the field part of the Vlasov-Maxwell system, then the resulting system together with (7.15), one obtains a large system of ordinary differential equations describing the evolution of the semi-discrete particle-in-cell system. This system can then be discretized using a standard ODE integrator, resulting in a flow of data as depicted in Figure 7.3. In particular, notice that I have not employed operator splitting in time and thereby retain the full order of accuracy of whatever time integrator I have chosen to use.

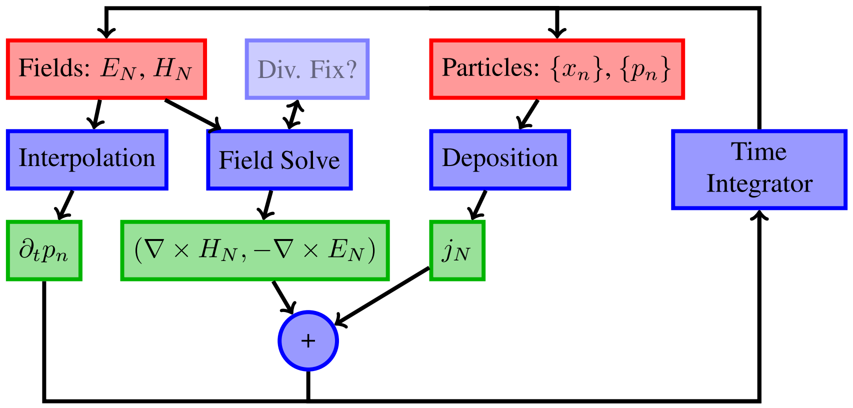


Figure 7.3. Data flow graph for DG-PIC.

PIC methods based on the scheme by Yee [1966] are typically used in conjunction with Leapfrog time integrators that allow them to achieve exact conservation of total (electromagnetic and kinetic) energy. Leapfrog integrators require a partitioning of the system into two parts, natural candidates being each of the two staggered meshes of the Yee scheme.

While standard discontinuous Galerkin can of course be used with Leapfrog integrators, such use (also in conjunction with order-increasing generalizations) is, to the best of my knowledge, relatively recent in the literature [Diaz and Grote, 2009]. Further, since the typically employed upwind fluxes (e.g. (7.7)) add a small amount of dissipation in exchange for excellent control of spurious modes, exact energy conservation (i.e. conservation down to floating point accuracy) is often perceived as less valuable and Runge-Kutta integrators have been preferred [Hesthaven and Warburton, 2007]. On the more extreme end of the damping spectrum, fully implicit methods have recently been proposed to entirely avoid resolving high-order modes in the electromagnetic fields [Drouin et al., 2010].

7.6.1 Multi-rate Time-Stepping for PIC

The time scales on which the sub-systems (7.15) and (the DG discretization of) (7.1) evolve may differ significantly. While it is common in particle-in-cell simulations to see stiff behavior in the particle part of the system, the Courant-Friedrichs-Lewy time step limit of the DG-discretized Maxwell's system scales as

$$\Delta t \sim \frac{h}{\lambda_{\mathrm{max}} N^2}, \tag{7.16}$$

where λ_{max} is the largest characteristic velocity, h is the local mesh size and N is the approximation's polynomial degree [Gottlieb and Tadmor, 1991, Hesthaven and Warburton, 2007]. Whatever the precise nature of each time step restriction may be at a given point in time, it is likely that both restrictions do not agree and may in fact vary by as much as an integer multiple. Observe that a mismatch of time step requirements becomes even more likely if hyperbolic cleaning of Section 7.5.2 is employed, as the additional characteristic wave introduce propagates a velocity that is typically a multiple of the speed of light that naturally occurs in Maxwell's equations.

Depending on the exact method parameters (and the presence of hyperbolic cleaning), it is likely that the time step restriction (7.16) required by the DG-based field approximation will be the more stringent of the two. Jacobs and Hesthaven [2009] present one way of dealing with this problem by using an implicit-explicit (IMEX) Runge-Kutta method, treating the discretized DG system in an implicit fashion by means of a sparse-factorization-based, direct solver. This subsection presents an alternative approach.

As originally suggested by Warburton [priv. comm.] and further examined by Stock [2009], it may make sense to employ multi-rate time stepping to save work on the component of the system permitting the larger time step. Multi-rate time stepping methods, and

To \ From	Particles		Fields	
Particles	$v(p)$	(7.15a)	$L(E(x_n), H(x_n))$	(7.15b)
Fields	$j = q\int vf\,\mathrm{d}v$	(7.4b)	$(\nabla \times H_N, -\nabla \times E_N)$	(7.1)
	$(0, 0, \chi(\rho/\epsilon))$	(7.14c)	$(-\chi\nabla\phi, 0, -\chi(\nabla \cdot E))$	(7.14a)

Figure 7.4. Overview of particle-field coupling for particle-in-cell simulation. The very last row of the table is only relevant if hyperbolic cleaning (see Section 7.5.2) is used.

in particular multi-rate linear multi-step methods, are discussed in Chapter 8. Most of the work in PIC is in the particle-to-field and field-to-particle coupling terms, as illustrated in Figure 7.4. One of the key contributions of Chapter 8 is to allow more control over when and how often the expensive coupling (i.e. off-diagonal) terms in Figure 7.4 are evaluated.

Stock [2009] discusses the combination of DG-PIC with multi-rate Adams-Bashforth methods at length, presenting comprehensive data on accuracy and application performance impact.

7.7 Deposition methods for DG-PIC

As discussed in Section 7.3.2, once a field discretization has been fixed, the main components of a PIC scheme are the off-diagonal coupling terms of Figure 7.4. This and the following section discuss various choices for each of these two coupling terms, beginning with the computation of charge and current densities ρ and j from particle quantities.

Note that, without loss of generality, this section will only discuss deposition for the current density j. In each case below, a deposition method for ρ can be obtained by simple analogy.

7.7.1 Element-wise Deposition

One of the first, and simplest, schemes that was proposed for DG-PIC is that of Jacobs and Hesthaven [2006]. Inserting the particle discretization (7.8) into (7.4b) yields

$$j(x,t) = \sum_{n=1}^{P} v_n q_n \frac{1}{r_n^d} S\left(\frac{x - x_n}{r_n}\right). \tag{7.17}$$

Given this expression, the main question is how to accurately obtain a representation of it in terms of a DG expansion. The simple solution proposed by Jacobs and Hesthaven [2006] involves, on each element D_k, the point evaluation of (7.17) at a set of well-conditioned nodal points $\{\xi_{k,i}\}$ and the use of these nodal values as coefficients of an expansion in Lagrange polynomials $l_{k,i}$ on D_k

$$j_N^{\text{intp}}|_{\mathsf{D}_k}(x,t) := \sum_{i=1}^{N_p} l_{k,i}(x) j(\xi_i, t). \tag{7.18}$$

This purely interpolatory scheme is, as illustrated in Figure 7.5, crucially dependent on the local availability of sufficient nodal resolution near the particle center. If one chooses particle radii r_n to match local mesh sizes, and if typical edge-clustering nodal sets (such as the one by Warburton [2006]) are used, then the resolution with which (7.17) is discretized is not homogeneous in space, which can lead to location-dependent (and often time-oscillatory) pollution of the obtained value of j.

In trying to avoid this issue, one is led to investigate alternate possibilities for the evaluation of (7.17) on a DG discretization, for example the exact or approximate numerical evaluation of the L^2 projection

$$j_N^{\text{proj}}|_{\mathsf{D}_k}(x,t) := \sum_{i=1}^{N_p} \phi_i(x) \int_{\mathsf{D}_k} j(\xi,t)\phi_i(\xi)\,\mathrm{d}\xi \tag{7.19}$$

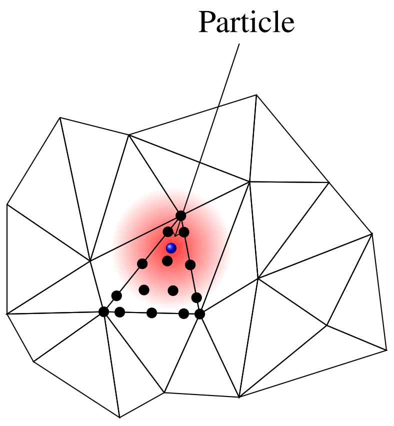


Figure 7.5. Element-wise deposition. The compactly supported particle is shown in red, along with the nodal points of the mesh with which it is interacting.

for each element D_k, where $\{\phi_n\}$ is some suitable orthonormal basis on D_k that spans the finite element space used for E_N [see e.g. Dubiner, 1991, Koornwinder, 1975]. Unfortunately, the use of (7.19) brings its own set of issues and is not necessarily superior to (7.18). An exact evaluation of (7.19) is complicated by the presence of truncation to zero in the expression for S, cf. (7.9), and likely not feasible within reasonable cost constraints. Alternatively, an approximate evaluation of the integrals in (7.19) by some quadrature scheme might provide an advantage through its adjustable level of additional resolution, however it depends on nodal evaluations just like (7.18). And even if one managed to evaluate (7.19) exactly, the failure mode of (7.19) in the case of under-resolved particles is potentially less benign than that of (7.18): Instead of losing mass and adding local oscillations, (7.19) will generate non-local oscillations and a discontinuous current density. In summary, while interpolatory deposition is certainly less expensive, it is not clear that the results obtained by the more expensive projection method are necessarily computationally preferable.

In this work, I use the basic element-wise deposition scheme (7.18) as a published [Jacobs and Hesthaven, 2006] baseline against which to evaluate other schemes, both with regard to computational efficiency and figures of merit derived from the physics. I will further discuss a few important implementation details of the element-wise deposition scheme in the following subsections.

Should $\int \rho_N = q$ be enforced?

An entirely interpolatory method like (7.18) will not manage to ensure that

$$\int_\Omega j_N^{\text{intp}}(x,t)\,\mathrm{d}x \overset{?}{=} \int_\Omega j(x,t)\,\mathrm{d}x.$$

This is even clearer when broken down further

$$\sum_{k=1}^{K} \int_{\mathsf{D}_k} \sum_{i=1}^{N_p} l_{k,i}(x)\rho(\xi_{k,i},t)\,\mathrm{d}x \overset{?}{=} \sum_{n=1}^{P} q$$

and stated for a single particle:

$$\sum_{k=1}^{K} \int_{\mathsf{D}_k} \sum_{i=1}^{N_p} l_{k,i}(x)\frac{1}{r^d} S\left(\frac{\xi_{k,i}-x_0}{r}\right)\mathrm{d}x \overset{?}{=} 1, \tag{7.20}$$

for some x_0 and r such that $B(x_0,r) \subset \Omega$.

Because $S \in C^{\alpha-1}$ for the usually chosen $\alpha \in \{2,3,4,\dots\}$, interpolation error estimates ensure that asymptotically (for a 1D example), as $h \to 0$, the L^∞ error in (7.20) decreases at least as $h^{\min(\alpha-1,N+1)}$. However given that the approach is targeted at about $N \approx 4$ and $h \sim r$, it is safe to assume that the approximation will be in the pre-asymptotic regime of any such estimate and hence (7.20) will be far from equality. It is obvious that this fact alone is enough to create large divergence errors, and it is tempting to "fix" at least the zeroth moment by numerically evaluating the left hand size of (7.20) and rescaling each particle's shape such that its integral matches its desired value.

However it is not just the moments of j_N^{intp} that are wrong, shape and location are approximated equally poorly. As such, one can expect to actually *worsen* various figures of merit by this well-intentioned “fix”. The experimental data of Section 7.9 confirms this expectation.

Element Finding by Mesh Connectivity

A further, more algorithmic problem encountered by element-wise deposition is finding exactly which elements intersect with the spherical support of each particle’s shape. Because of the scale of the problem in both the number of particles and the number of elements, exhaustive search is prohibitive, and simple heuristics such as intersection-of-circumspheres lead to excessive numbers of “false positives”, i.e. elements tagged for overlap with the particle when in fact no such overlap is occurring.

I have designed a dimension-independent, fast, greedy procedure aimed at finding all simplicial elements intersecting a spherical particle with very few “false positives”, i.e. elements tagged for overlap with the particle when in fact no such overlap is occurring. The procedure is illustrated in Figure 7.6(a), and it is based on an assumed prior knowledge of the element containing each particle’s center point. It is focused on quickly making face-based element-to-element transition decisions, based on the two geometric objects whose intersection tests with a sphere are trivial: that of another sphere and that of a (hyper)plane. Algorithm 7.2 exhibits the details.

Algorithm 7.2 and many other parts of a particle-in-cell scheme rely on knowledge of which element contains each particle’s center. Therefore, this center point element membership information, once initially established by perhaps an octree or global search, needs to be maintained in a data-local and efficient fashion. Elaborate schemes such as the one by Haselbacher et al. [2007] have been devised to perform this task, however in my

Algorithm 7.1 Maintenance of information on element membership of each particle's center point.

```
Require: Present particle position $x$
Require: Present particle momentum $p$
Require: $k' \in \{1, \dots, K\}$ such that $\mathsf{D}_{k'}$ is the last element known to contain $x$
Ensure: Find a $k$ such that $k \in \mathsf{D}_k$
  Compute barycentric coordinates $\lambda^{(k')}$ of $x$ relative to $\mathsf{D}_{k'}$
  if $\lambda_i^{(k')} \geq 0$ and $\sum_i \lambda_i^{(k')} \leq 1$ then
      $k \leftarrow k'$, i.e. $x \in D_{k'}$, return.
  end if
  Find face $F \subset \partial\mathsf{D}_{k'}$ maximizing $\hat{n}_F \cdot p > 0$, where $\hat{n}_F$ is $F$'s outside unit normal vector
  if there is such an $F$ then
      if $F$ is a boundary face then
          Treat particle boundary condition for $x$ and $F$, return.
      end if
      Find index $k$ such that $\mathsf{D}_k \cap \mathsf{D}_{k'} = F$.
      Check for $x \in \mathsf{D}_k$ as above, if found, return.
  end if
  Find vertex $\xi$ of $\mathsf{D}_{k'}$ minimizing $\|x - \xi\|_2$
  for all elements $\mathsf{D}_k$ adjoining $\xi$ do
      Check for $x \in \mathsf{D}_k$ as above, if so, return.
  end for
  { none of the heuristics worked out }
  Find $\mathsf{D}_k$ by exhaustive search.
```

case the simple strategy of Algorithm 7.1 has proven to be entirely sufficient.

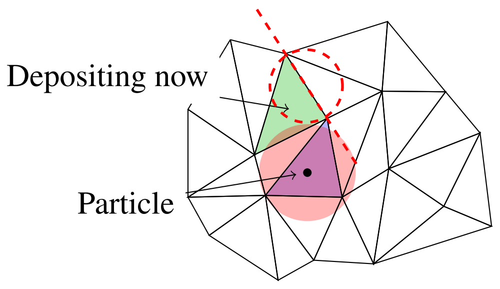
Algorithm 7.2 `find_dep`(k'): Recursive procedure for element finding by mesh connectivity.

```
Require: A particle center point $x$ and radius $r$
Require: A set $K_p$ of indices $k$ such that $\mathsf{D}_k$ has already been deposited, initially empty.
Require: An element index $k'$ known to satisfy $\mathsf{D}_{k'} \cap B(x,r) \neq \emptyset$, initially the element
  containing $x$.
Ensure: $K_p = \{k : \mathsf{D}_k \cap B(x,r) \neq \emptyset\}$
  $K_p \leftarrow K_p \cup \{k'\}$
  for all faces $F \subset \partial\mathsf{D}_{k'}$ do
      Find index $k$ such that $\mathsf{D}_k \cap \mathsf{D}_{k'} = F$. If no such $k$ exists, skip this face.
      if $k \in K_p$ then
          skip this face.
      end if
      if $d(P,x) > r$, where $P$ is the (infinite) (hyper)plane containing $F$ then
          skip this face.
      end if
      if $d(C,x) > r$, where $C$ is a minimal sphere containing $F$ then
          skip this face.
      end if
      find_dep($k$)
  end for
```

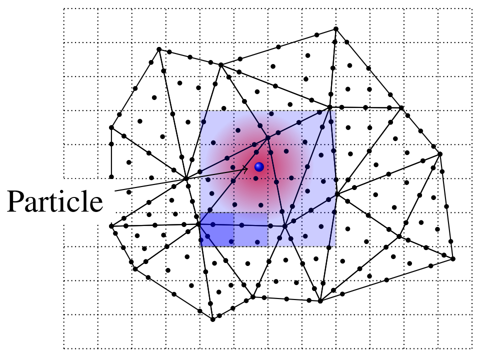
Deposition by Cartesian Node Binning

If one is content with a nodal interpolation scheme for deposition, such as (7.18) or a quadrature version of (7.19), and if the geometry under consideration is simple enough and has not much variation in mesh resolution, the expense of Algorithm 7.2 can be decreased somewhat, at the expense of having to build and retain an auxiliary data structure.

Figure 7.6(b) illustrates the idea. During pre-processing, one creates a Cartesian mesh that matches the resolution of the target interpolation mesh in such a way that a constant, low number of interpolation nodes fall within each Cartesian grid cell. In essence, one views the interpolation nodes as a “sea of points”, ignoring the “element” level in the structure of the discretization. One then builds a lookup table from each Cartesian cell to



(a) Element Finding by Mesh Connectivity.



(b) Node Finding by Cartesian Binning.

Figure 7.6. Various ways of implementing element-wise deposition, as described in Section 7.7.1

.

the interpolation nodes contained within.

Then, to evaluate (7.18), one follows the simple Algorithm 7.3. In my experience, Algorithm 7.3 can be up to twice faster than Algorithm 7.2 on suitable meshes in three dimensions. The key to this speed increase is that lookups in a Cartesian spatial lookup table are very cheap. There are however geometries where the method is less suitable and can lead to either excessive memory use or poor run-time performance. These include shapes which do not fill a significant fraction of their Cartesian bounding box, or meshes which are very inhomogeneously refined. Some of these issues could potentially remedied with an octree lookup, but since the margin over true element-wise deposition is only a factor of two, it is somewhat doubtful whether this strategy would pay off. Also realize that the algorithm deposits square supersets of the actual spherical shape of the particle, which leads to some inefficiency.

Algorithm 7.3 Element-wise deposition by Cartesian node binning.

```
Require: A Cartesian grid $o + \sum_i \Delta x_i \alpha_i e_i$ with finite extents $0 \le \alpha \le \alpha_{\max}$ and $e_i$ the
    $i$th unit vector. Further let $C_\alpha$ be the right-half-open grid cell at index $\alpha$.
Require: A look-up table $T : \alpha \mapsto I$, a list of node indices such that $\xi_i \in C_\alpha$.
Ensure: $j_N^{\text{dep}} \equiv j_N^{\text{intp}}$
  $j_N^{\text{dep}} \leftarrow 0$
  for all particle indices $n \in \{1, \dots, P\}$ do
      Compute the index range $\alpha_- < \alpha_+$ such that $B(x_n, r_n) \subset \bigcup_{\alpha_- \le \alpha < \alpha_+} C_\alpha$
      for all $\alpha_- \le \alpha < \alpha_+$ do
          for all $i \in T(\alpha)$ do
              $j_N^{\text{dep}}(\xi_i, t) \leftarrow j_N^{\text{dep}}(\xi_i, t) + qv(p_n)\frac{1}{r_n^d} S\left(\frac{\xi_i - x_n}{r_n}\right)$
          end for
      end for
  end for
```

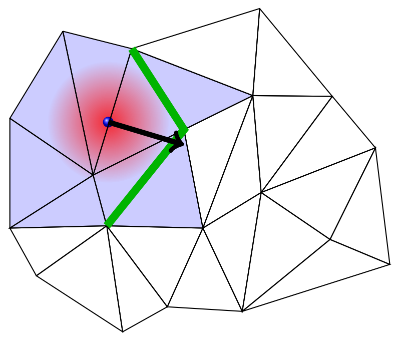
7.7.2 Advective Deposition

Interpolatory deposition, as explored above, is very dependent on the exact distribution of interpolation nodes within each element and across the mesh. This and the following section represent my attempts to alleviate this dependence. The first scheme is based on the recognition that much of the divergence error (see Section 7.5) produced by the schemes of the previous section originates from the repeated transitions from an analytic representation (7.17) to a polynomial representation (7.18) of the current density j and the desire to avoid them.

In principle, this is easy enough–the only transformation that a single particle's shape function undergoes in time is a simple translation. If a DG-discretized representation $\mathcal{S}_N^{(n)}$ of the particle n's shape,

$$\mathcal{S}^{(n)}(x, t) = \frac{1}{r_n^d} S\left(\frac{x - x_n}{r_n}\right),$$

has been fixed in some way (say by interpolation), then the translation implied by (7.15a)


(a) Activation Stage of Advective Deposition.

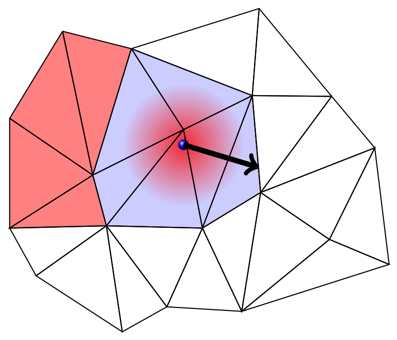

(b) Post-timestep Deactivation Stage of Advective Deposition.

Figure 7.7. The stages of advective deposition from the point of view of one particle.

can be performed on $\mathcal{S}^{(n)}$ by an advection equation with constant speed $v(p_n)$:

$$\partial_t \mathcal{S}^{(n)} + \nabla \cdot (v(p_n)\mathcal{S}^{(n)}) = 0, \tag{7.21}$$

or, respectively on $\mathcal{S}_N^{(n)}$ by a DG discretization of (7.21). Because of its reliance on the advection equation, it seems sensible to call the resulting scheme *advective deposition*.

For reasons of computational expense, and because $\operatorname{supp} \mathcal{S}^{(n)} = B(x_n, r_n)$, it seems appropriate to truncate the domain on which the solution to (7.21) is to be computed as close to $\operatorname{supp} \mathcal{S}_N^{(n)}$ as is feasible, taking into account that for $t > 0$ $\operatorname{supp} \mathcal{S}_N^{(n)} \neq \operatorname{supp} \mathcal{S}^{(n)}$.

The scheme progresses as follows: To bootstrap the computation, advective deposition requires another depositor to establish for each particle (say, number n) a "*mini-domain*" $\bigcup_{j=1}^{K_n} \mathsf{D}_{M_{n,j}}$, where $M_{n,1}, \dots, M_{n,K_n}$ are the elements in particle n's mini-domain, and the approximation $\mathcal{S}_N^{(n)}$.

Once this initial data is established, $\mathcal{S}_N^{(n)}$ becomes part of the time-stepped state of the solver and thereby requires that a right-hand side be calculated for it. For the nth particle, this right-hand side calculation progresses as drafted in Algorithm 7.4.

Algorithm 7.4 Update procedure for advective deposition.

```
Apply DG operator for $\partial_t \mathcal{S}_N^{(n)} + \nabla \cdot (v(p_n)\mathcal{S}_N^{(n)}) = 0$ on $\bigcup \mathsf{D}_{M_{n,j}}$ with zero-inflow or outflow boundary conditions as appropriate.
for all elements $\mathsf{D}_{M_{n,j}}$ do
    for all faces $F \subset \partial\mathsf{D}_{M_{n,j}}$ do
        if $\max_{\text{node } \xi\in F} |\mathcal{S}_N^{(n)}(\xi)| > \alpha_{\text{activate}}||\mathcal{S}^{(n)}||_\infty$ then
            Add cross-$F$ neighbor of $\mathsf{D}_{M_{n,j}}$ to mini-domain, if any.
        end if
    end for
end for
```

Observe that no particle moves faster than the speed of light and hence the discretized advection equation obeys the same CFL condition as the system of Maxwell's equations (7.1). Further note that the adaptive enlargement of the mini-domain may occur in the middle of a time step and requires some bookkeeping effort. After each time step has completed, element $\mathsf{D}_{M_{n,j}}$ from particle n's mini-domain is retired if

$$\int_{\mathsf{D}_{M_{n,j}}} |\mathcal{S}_N^{(n)}(x)|^2 \, \mathrm{d}x < \alpha_{\text{deactivate}}.$$

Figure 7.7 illustrates element activation and deactivation.

Advective deposition achieves the goal that I set out to accomplish: By working directly on a polynomial representation of $\mathcal{S}_N^{(n)}$, it is not as dependent on evaluation node locations, and thereby far more homogeneous in space. If there was no artificial domain truncation, advective deposition would also achieve divergence error (see 7.5) that decays with the same order in the local mesh size h as the error in the numerical solution of the system of Maxwell's equations (7.1). However even if the domain truncation for $\mathcal{S}_N^{(n)}$ could be done perfectly, the entire scheme is still very expensive, both in terms of memory and computation:

- Unlike in element-wise deposition, the discretized $S_N^{(n)}$ becomes part of system state and must be retained, updated and processed by the time stepper.

- The interpolatory deposition of Section 7.7.1 need to touch each nodal coefficient roughly once per particle. The approximation of (7.21) requires many more nodal data movements even just to carry out the element-local derivatives.

- In practice, much depends on the finite threshold values α_{activate} and $\alpha_{\text{deactivate}}$. For values of these that are small enough to ensure $\mathcal{S}_N^{(n)}$ retains charge, I have observed element counts on the order of 100 in three dimensions. For lower threshold values, particle charge is lost surprisingly quickly.

For a minor savings of work, $\mathcal{S}_N^{(n)}$ can be maintained pre-multiplied with q.

Based on the performance remarks in the previous paragraph, it is clear that, unfortunately, advective deposition is not a practical scheme as-is. Luckily though, it is not a complete write-off. First, it is valuable in showing that near-order-of-the-scheme divergence error for particles with extent is feasible (see Section 7.9), even if this feature comes at an uneconomical cost at present. But it may be possible to generate a current density j_N similar to the advective one at a smaller cost. Second, advective deposition has significant untapped potential. For example, since $\mathcal{S}_N^{(n)}$ becomes part of global solver state, the scheme can trivially deal with non-uniform particle shapes. Further, one could add a spatial dependency to the velocity $v(p_n)$ in (7.21). The end result of these two improvements would be a localized fluid model that might be helpful in resolving some phase space features.

7.7.3 Cartesian Deposition

While the idea of advective deposition was to obviate the problem of varying nodal resolution by remaining in a polynomial representation for all time, this section presents a

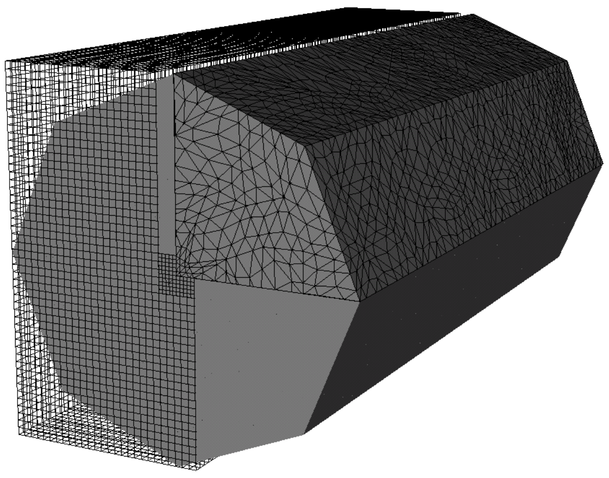
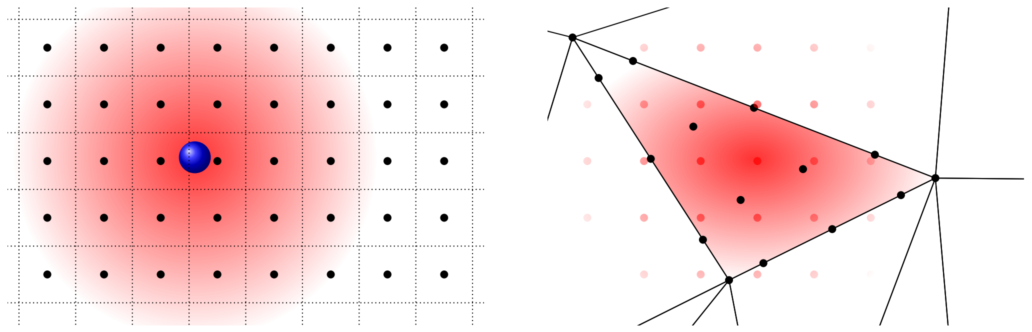


Figure 7.8. Matching Cartesian and unstructured mesh resolution. Observe that both the Cartesian and the unstructured mesh are refined at the center of the beam tube. The picture shows nodal resolution in both the structured and the unstructured meshes.

different approach that addresses that problem more directly, namely by providing mostly uniform resolution across all space.

In *Cartesian deposition*, deposition occurs initially on a structured, Cartesian grid. The method uses pointwise interpolation onto nodes on a structured Cartesian grid and a subsequent remapping onto the discontinuous Galerkin function space. The two stages of the procedure are portrayed in Figure 7.9 on the following page. In addition to providing an attempt at a remedy for the non-uniformity in node density, Cartesian deposition also represents an attempt to benefit from the efficiency of algorithms running on Cartesian grids, as was already done in the node binning method of Section 7.7.1.

The idea behind Cartesian deposition is straightforward, but its implementation faces at least two challenges, both rooted in the fact that the Cartesian mesh may provide “too little” resolution. First, the non-uniformity of nodal resolution is a byproduct of the need to arrange nodes in such a way that Vandermonde matrices remain well-conditioned. Second, if a computational mesh has been refined to provide additional resolution in areas of interest or computational importance, this non-uniformity is entirely intentional, and deposition



(a) Initial (fast, per-particle) deposition onto the Cartesian grid.

(b) Subsequent (slower, per-element) conversion of Cartesian data to a local function in the approximation space.

Figure 7.9. Stages of grid-based deposition. See also Algorithm 7.6.

onto a Cartesian overlay mesh that does not respect this additional resolution will give poor results. One solution would be the use of a generic recursive-bisection-based adaptive refinement for the Cartesian mesh, however this has the potential to negate the performance gains one was hoping to achieve by moving to a structured mesh in the first place. Another possibility is the construction of the Cartesian mesh from "bricks" of varying grid sizes, as illustrated in Figure 7.8.

The practical execution of Cartesian deposition consists of some pre-processing, shown in Algorithm 7.5, and the actual on-line part of the algorithm, shown in Algorithm 7.6. There are two key observations that greatly influence Algorithm 7.6's execution speed. First, the second-to-innermost loop, which requires finding a set of bricks overlapping the support $B(x_n, r_n)$ of a particle's shape function. Since this information changes relatively slowly and (depending on the number of bricks) can be somewhat expensive to recompute, I have found it expedient to retain it from one deposition to the next, checking before use whether it is still current. Second, the innermost loop of the algorithm is of a very structured nature resembling that found in codes implementing finite-difference methods. A multitude of techniques such as vectorization has been developed to implement this type of loop efficiently, and many of them are applicable in this instance, further contributing to the efficiency of Cartesian deposition.

Algorithm 7.5 Cartesian Deposition: Computation of Cartesian-to-nodal map, accomplished during pre-processing.

```
Require: $\xi_{k\nu}$ is the $\nu$th interpolation node in element $\mathsf{D}_k$.
Require: $\Psi_k$ is the local-to-global map for element $\mathsf{D}_k$.
Require: $\phi_m$ is the $m$th element of an element-local orthonormal basis [see e.g. Dubiner,
    1991, Koornwinder, 1975].
Require: $\Omega$ decomposed into bricks $\mathbf{B}_i$, each containing nodes $c_{i\nu}$ arranged in a regular,
    Cartesian grid.
Ensure: For each element $\mathsf{D}_k$, a list $C_k$ of Cartesian Cartesian points $\{\widetilde{c}_{k1}, \dots, \widetilde{c}_{k|C_k|}\}$ and
    a well-conditioned matrix $M_k \in \mathbb{R}^{N_p \times |C_k|}$ mapping Cartesian values on the points $C_k$ to
    DG-nodal values on $\mathsf{D}_k$.
    for all elements $\mathsf{D}_k$ do
        { Find and number cell centers inside element }
        $C_k \leftarrow \{\widetilde{c}_{k1}, \dots, \widetilde{c}_{k|C_k|}\} \leftarrow \{c_{i\nu} : c_{i\nu} \in \mathsf{D}_k\}$
        { Calculate grid Vandermonde matrix $G_k$ }
        $(G_k)_{\nu m} \leftarrow \phi_m(\Psi_k^{-1}(\widetilde{c}_{k\nu}))$ for $\nu \in \{1, \dots, |C_k|\}$ and $m \in \{1, \dots, N_p\}$
        If necessary, improve conditioning of $G_k$ by Algorithm 7.7.
        { Compute Cartesian-to-unstructured map }
        $M_k \leftarrow V G_k^{+}$, where $V_{\nu m} = \phi_m(\Psi_k^{-1}(\xi_{k\nu}))$
    end for
```

Second, the preprocessing stage (and its failure modes) deserve some mention. Realize that the net effect of Algorithm 7.5 is to ensure that the result computed by Algorithm 7.6 locally on each element D_k satisfies an l^2-optimality condition:

$$\sum_{\nu=1}^{|C_k|} \left| j_N(\widetilde{c}_{k\nu}) - j_G(\widetilde{c}_{k\nu}) \right|^2 \to \min! \tag{7.22}$$

in the notation of Algorithm 7.6. Technically, this is always possible, but in the event that not enough data is available on the Cartesian grid to overdetermine (or at least fully determine) j_N, the solution to the least-squares problem (7.22) is non-unique, because coefficients for too many basis functions need to be determined. I have explored three strategies of dealing with this issue:

Basis reduction, which removes functions from the approximation basis until there is

Algorithm 7.6 Cartesian Deposition: Grid-based deposition and nodal unstructured remapping.

```
Require: $\Omega$ decomposed into bricks $\mathbf{B}_i$, each containing nodes $c_{i\nu}$ arranged in a regular,
  Cartesian grid.
Require: Remap matrices $M_k$ as computed by Algorithm 7.5
Ensure: $j_N$ contains an approximation of $j$ from (7.17) on the DG grid.
  { Deposit onto Cartesian mesh }
  for all particles with number $n \in \{1, \dots, P\}$ do
      for all bricks $\mathbf{B}_i$ overlapping $B(x_n, r_p)$ do
          Find bounding box of $B(x_n, r_p) \cap \mathbf{B}_i$
          for all cell centers $c_{i\nu}$ in bounding box do
              $j_G(c_{i\nu}) \leftarrow j_G(c_{i\nu}) + q_n v(p_n) \frac{1}{r_n^d} S\left(\frac{c_{i\nu} - x_n}{r_n}\right)$
          end for
      end for
  end for
  { Remap from Cartesian onto DG mesh }
  for all elements $\mathsf{D}_k$ do
      $j_N|_{\mathsf{D}_k} \leftarrow M_k(j_G(\widetilde{c}_{k\nu}))_{\nu=1}^{|C_k|}$
  end for
```

enough data to fully determine the resulting basis expansion,

Domain-of-dependence enlargement, which considers a (relative to D_k) barycentrically larger area in which to search for Cartesian points to use in the remapping,

Placement of additional, non-grid evaluation nodes, which finds points outside the Cartesian mesh, which, if available, would lead to an optimal reduction of the condition number.

Domain-of-dependence enlargement encountered issues because the polynomials under consideration start oscillating very quickly outside the simplex. The placement of additional evaluation nodes was somewhat effective, but both expensive, and it highlighted the fact that ill-conditioning typically occurs for “odd-shaped” simplicial elements such as slivers, where the accuracy gain through additional resolution would be somewhat artificial. Hence, the in my experience best strategy of coping with ill-conditioning is is basis reduction, which is encoded in Algorithm 7.7.

Algorithm 7.7 Conditioning fix for grid Vandermonde matrix G_k.

```
Require: An element number $k$
Require: $\Psi_k$ is the local-to-global map for element $\mathsf{D}_k$
Require: $\phi_m$ is the $m$th element of an element-local orthonormal basis.
Require: A list $C_k$ of Cartesian points $\{\widetilde{c}_{k1}, \dots, \widetilde{c}_{k|C_k|}\} \subset \mathsf{D}_k$
Ensure: $\kappa(G_B) = O(1)$
  { a set of "participating" basis function indices, initialized to full basis }
  $B \leftarrow \{1, \dots, N_p\}$
  loop
      { Compute "restricted" grid Vandermonde matrix }
      $(G_B)_{\nu j} \leftarrow \phi_{B_j}(\Psi_k^{-1}(\widetilde{c}_{k\nu}))$ for $\nu \in \{1, \dots, |C_k|\}$ and $j \in \{1, \dots, |B|\}$
      { Perform SVD }
      $U\Sigma V^T \leftarrow \operatorname{svd}(G_B)$, where $\Sigma_{ij} = 0$ for $i \neq j$ and $|\Sigma_{ii}| > |\Sigma_{jj}|$ for $i > j$
      $n \leftarrow \min(|B|, |C_k|)$
      if $|\Sigma_{11}/\Sigma_{nn}| > 10$ then
          { $G_B$ is ill-conditioned, remove "most singular" basis function }
          $i \leftarrow \operatorname{argmax}\{|V_{jn}| : j \in B, \deg \phi_j = \max\{\deg \phi_l : l \in B\}\}$
          $B \leftarrow B \setminus \{i\}$
      else
          Done, return.
      end if
  end loop
```

One closing observation is that the current density output of Algorithm 7.6 will, in most cases, be discontinuous, unphysically providing for element interfaces that exhibit different values of j on each side. Given that the modeled j is a sum of smooth particles, it seems sensible to “repair” this deficiency by averaging among coincident nodal values. The efficacy of this remedy will be assessed in Section 7.9.

Cartesian deposition provides a deposition method that is both fast and very predictable in terms of resolution availability. It could further be extended to include specialized current density deposition schemes such as the one by Villasenor and Buneman [1992] (see Figure 7.1(b)).

Lastly, the introduction of an auxiliary Cartesian mesh is somewhat of an embarrassment to an unstructured method, but so far the computational effectiveness and the quality of the results seems to justify at least retaining it within consideration.

7.8 Particle Pushing in DG-PIC

7.8.1 Interpolatory Pushing

In comparison with particle deposition, the reverse direction of the coupling (appropriately called “*particle pushing*”) between (7.1) and (7.2) is rather simple–the main goal is to evaluate, for each particle, the Lorentz force (7.3),

$$L(E, H, x_n, p_n, t) = q(E(x_n, t) + v(p_n) \times \mu H(x_n, t)) \tag{7.23}$$

to use as a right-hand side in the particle equation of motion (7.15b)

$$\partial_t p_n = L(E, H, x_n, p_n, t)$$

As such, computing (7.23) requires a number of point evaluations of the approximate electric field E_N and the approximate magnetic field H_N. Algorithmically, this is accomplished by preparing a Vandermonde matrix

$$(V_{ij})_{i,j=1}^{N_p} := \phi_j(\hat{\xi}_i),$$

where $(\phi_j)_{j=1}^{N_p}$ represent a per-element basis of the DG approximation space and $(\hat{\xi}_i)_{i=1}^{N_p}$ are node locations on the reference element I, and a vector of point evaluations

$$(m_i)_{i=1}^{N_p} := \phi_i(\Psi_k^{-1}(x_n)), \tag{7.24}$$

where Ψ_k is the local-to-global map for element D_k. Then, with $\alpha := V^{-T}m$ one has

$$E(x_n) = \sum_{i=1}^{N_p} \alpha_i E(\xi_{ki}), \tag{7.25}$$

where one supposes that the particle position $x_n \in \mathsf{D}_k$, and, as above $(\xi_{ki})_{i=1}^{N_p} = (\Psi_k(\hat{\xi}_i))_{i=1}^{N_p}$ are the nodal locations on element D_k. Observe that this represents another place where the information on which element contains a given particle is used, as maintained by Algorithm 7.1.

Interpolatory pushing is the method used by Jacobs and Hesthaven [2006], it generally works well and is reasonably efficient. Both of its algorithmic halves allow the use of tricks that increase computational efficiency:

- The method does not make assumptions on the nature of the basis (ϕ_i). Hence, as long as the chosen basis spans the same space as the one used for approximation, the computed force should not vary significantly. As a result, one may evaluate basis sets solely on two criteria: the efficiency with which the point evaluations (7.24) can be computed and the conditioning of the resulting Vandermonde matrix V, and thereby, the accuracy with which $\alpha = V^{-T}m$ can be computed. While the simplicial ONB used throughout the rest of this work [Dubiner, 1991, Koornwinder, 1975] provides very good conditioning for a large range of polynomial orders, its evaluation is somewhat expensive. In accordance with the work of Jacobs and Hesthaven [2006], I have found that for the polynomial orders of interest here, a plain monomial basis

$$\{\tilde{\phi}_{klm}(x, y, z) := x^k y^l z^m : k + l + m \leq N\}$$

provides a better trade-off of these two factors. If higher-order accuracy is desired than is feasible by monomial interpolation, efficient evaluation schemes for the Dubiner basis provide an attractive option [Kirby, 2010].

- Once the point evaluations m are available, the computation of the interpolation coefficients α can be performed in a large batch for all particles at once, turning the matrix-vector operation $\alpha = V^{-T}$ into a matrix-matrix operation. In the context of computations, matrix-matrix operations are generally much more efficient than their matrix-vector counterparts, which is exploited by making this change.

7.8.2 Mean-based pushing

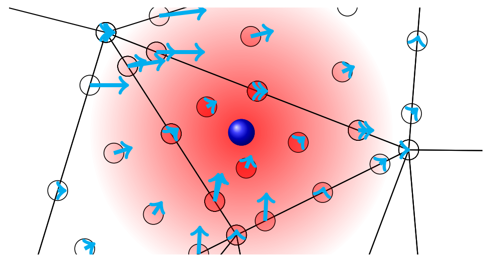
The point-shape, interpolatory pushing of the preceding section, and in particular the derivation of (7.15b), made the assumption that $S \equiv \delta$, i.e. particles occupy a single point in space. But consider that throughout Section 7.7 in the context of deposition, this assumption has been violated by the use of particles with a nonzero extent in space for reasons of noise mitigation. It seems appropriate to examine what might happen if one admits particles with an extent in particle pushing.

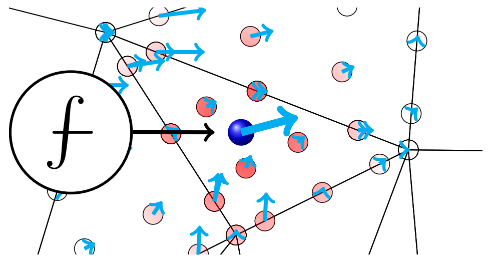
I would like to remark upfront that this change does *not* make the scheme entirely self-consistent. In particular, substituting a shape function like (7.9) into the Vlasov equation (7.2) and following the procedure of Section 7.6 yields the (somewhat obvious) observation that, aside from points, no fixed shape is in general mapped onto itself by the Vlasov equation. There have been efforts to design parametric classes of particles that make steps towards being closed under the action of the Vlasov equation [Hewett, 2003]. The latter work also discusses splitting and merging techniques when the parametric approach is regarded as having broken down.

Even with this shortcoming, something may be gained by investigating the behavior of a particle-in-cell scheme where particles have an extent for both deposition deposition and pushing. My approach to the design of such a scheme is based on an analogy of an extent-having particle with a rigid body. I assume rotational symmetry of the associated shape function and therefore may ignore rotational motion and forcing. By examining the Lorentz force (7.3) exerted on an infinitesimal volume $\mathrm{d}x$ within a rigid particle and integrating, I arrive at the total linear force acting at the center of mass of a particle under the influence of electromagnetic fields:

$$L_n = \int_{B(x_n, r_n)} \rho_n(x)[E(x) + v(p_n) \times \mu H(x)] \,\mathrm{d}x. \tag{7.26}$$

Linearity allows another, equivalent way of viewing (7.26). Instead of using a point evaluation (7.25) for particle pushing as in the previous section, one uses a weighted mean


(a) Setting for mean-based pushing: Cloud-shaped particle with adjacent vector-valued fields, evaluated at element nodes.



(b) Computation of the particle cloud mean, assignment of the point force as the weighted mean force.

Figure 7.10. Stages of mean-based pushing.

of the electric and magnetic fields, i.e. for example

$$\bar{E}_n := \frac{1}{q} \int_{B(x_n, r_n)} E_N \rho_n \, \mathrm{d}x. \tag{7.27}$$

One then uses $\bar{E}_n$ instead of $E_N(x_n)$ in (7.15b). Figure 7.10 illustrates (7.27).

Two ideas are necessary to turn (7.27) into an implementable method. First, the integral in (7.27) may be viewed as an inner product that can be computed exactly by using either a suitable quadrature or the mass matrix available as part of the DG discretization. Second, to obtain an approximation of ρ_n, one reuses the output of the existing deposition algorithm, though at a different level than before. In particular, this, in conjunction, with the element-wise computation of the integral, requires that the deposition output be separable both by particle and by element. This constraint is satisfied by non-grid-binned element-wise deposition (Section 7.7.1), and advective deposition (Section 7.7.2), but not (cheaply) by Cartesian deposition (Section 7.7.3).

This leads to the scheme

$$\bar{E}_{n,N} := \frac{1}{\mathcal{N}} \sum_{\mathsf{D}_k \cap B(x_n, r_p) \neq \emptyset} \int_{\mathsf{D}_k} E_N(x) \rho_n(x) \, \mathrm{d}x, \tag{7.28}$$

where there are two plausible choices for the normalization $\mathcal{N}$:

- the particle charge q, or
- the *actually deposited* particle charge

$$\sum_{T_k \cap B(x_n, r_p) \neq \emptyset} \int_{\mathsf{D}_k} \rho_N \, \mathrm{d}x,$$

noting as above that the two will not necessarily agree. Unlike in the discussion of Section 7.7.1, the second choice may be superior in this instance as, because of averaging, the exact local charge distribution is less important than overall correct weighting, which might lead to oscillatory effects in the magnitude of the averaged field $\bar{E}_{n,N}$.

As will be seen in the subsequent results section, numerical results of mean-based pushing are, surprisingly, somewhat disappointing. In addition to the added expense of evaluating the integrals in (7.28), it tends to be vulnerable to the imperfections of the deposition method in use. As commented initially, the world view presented by mean-based pushing is somewhat more consistent than that of other methods, but it is not actually consistent with the governing equations. Earlier in this chapter, it was commented that sucessful methods for the Vlasov equation should allow some form of adaptivity to account for varying resolution needs. The first step to an adaptive scheme is, of course an indicator of how well-resolved a solution is at present, and mean-based pushing offers one option: If one views (7.28) as the computation of a mean as suggested above, one may also compute an estimate of its associated standard deviation. High standard deviations indicate that particles span areas of fields with large spatial variation in E and H, which is indicative of an area that should likely be refined.

7.9 Numerical Evaluation

In addition to presenting significant obstacles to the construction of a workable numerical scheme, the Vlasov-Maxwell system also presents significant obstacles to the validation of such numerical treatments. I have implemented my approaches, as detailed in the preceding sections, in the context of the discontinuous Galerkin solver hedge (see Chapter 3). Like Jacobs and Hesthaven [2006], I have successfully validated my implementation against simple tests such with spatially and temporally non-varying fields, such as Larmor screwlines and $E \times B$ drifts, essentially confirming that time stepping and field-to-particle interpolation is working as designed in all schemes of Section 7.8.

Jacobs and Hesthaven [2006] have already achieved initial validation of the scheme consisting of per-element deposition (Section 7.7.1) and interpolatory pushing (Section 7.8.1) on nontrivial tests such as plasma waves, Landau damping, and a full magnetron simulation. As discussed in Section 7.3.3, I consider it likely that high-order unstructured DG-PIC methods will see their first application in areas where there are there are well-separated regions of mainly electromagnetic and mainly particle-driven activity. I have therefore decided to bias my tests away from full-plasma tests as performed by Jacobs and Hesthaven [2006] and towards one area that exhibits this property, namely beam simulation in accelerator physics. Even in absence of nontrivial analytic solutions, it fortunately is possible to evaluate the behavior of the method by how well it matches known physical invariants of the system. Examples of such invariants include momentum (consisting of particle and field momenta)

$$\begin{aligned}
p_{\text{tot}} &= p_{\text{part}} + p_{\text{field}}, \\
p_{\text{part}} &= \sum_{n=1}^{P} |p_n|^2, \\
p_{\text{field}} &= \frac{1}{c_0^2} \int_\Omega E(x) \times H(x) \,\mathrm{d}x,
\end{aligned}$$

field and particle kinetic energy

$$\begin{aligned}
E_{\text{tot}} &= E_{\text{part}} + E_{\text{field}}, \\
E_{\text{part}} &= \sum_{n=1}^{P} \left(\frac{|p_n|}{|v(p_n)|} - m \right) c_0^2, \\
E_{\text{field}} &= \int_\Omega \frac{1}{2} (E(x) \cdot D(x) + H(x) \cdot B(x)) \,\mathrm{d}x,
\end{aligned}$$

and (in certain beam physics problems) beam emittance, of which the RMS (root mean square) variety is shown here:

$$\varepsilon_{y,\text{rms}}^2 = \langle (y - \langle y \rangle)^2 \rangle \langle (y' - \langle y' \rangle)^2 \rangle - \langle (y - \langle y \rangle)(y' - \langle y' \rangle) \rangle^2. \tag{7.29}$$

In this expression, $\langle \cdot \rangle = (1/P) \sum_{n=1}^{P} (\cdot)_n$ is a particle average, and $y' = p_y/p_z$ is a phase space coordinate common in accelerator physics [see, e.g., Wiedemann, 1993]. In this case, z is assumed to be the longitudinal direction in which the beam travels, whereas y is one of the transverse directions.

A further figure of merit is provided by the charge conservation identity (7.10), whose L^1 error one may use as a measure of error:

$$\frac{\int_\Omega |\rho_N(x) - \nabla \cdot (\epsilon E_N(x))| \,\mathrm{d}x,}{|Pq|} \tag{7.30}$$

where I have chosen to use the L^1 norm for two reasons: first, because it retains the physical unit of Coulomb and therefore gauges the amount of "misplaced" charge, and second, because it is preferred in the field of density estimation [see, e.g. Devroye and Lugosi, 2001], which is closely related to the problem of deposition in PIC. For comparability, I have scaled the error indicator relative to the total amount of charge present in the domain.

A further, simpler quality measure is that of comparing total intended charge with actually deposited charge.

$$\frac{\left|\int_\Omega \rho \,\mathrm{d}x\right|}{|Pq|} \tag{7.31}$$

This quantity is somewhat less informative than the L^1 divergence error, as the integral in (7.30) can be peeled away and local contributions to the error examined, whereas the same thing is impossible in (7.31). Further, the difference between (7.30) and (7.31) is indicative of how much of the divergence error is due to oscillatory behavior, as this would tend to not affect the quantity in (7.31). Again for comparability, I have scaled this error indicator relative to the total amount of charge present in the domain.

Note that most of the conservation properties mentioned in this section require that the system under consideration is closed, i.e. it has no absorbing or dissipating components.

Finally, I would like to add that the majority of the numerical evaluation to follow should be seen as, at best, qualitative in nature. In my opinion, this is to be expected, as the methods I am evaluating are still mostly experimental.

7.9.1 Gaussian Electron Beams

Based on the large variety of possible methods presented in Sections 7.7 and 7.8, the goal of this subsection is to provide some initial guidance on the strengths and weaknesses of

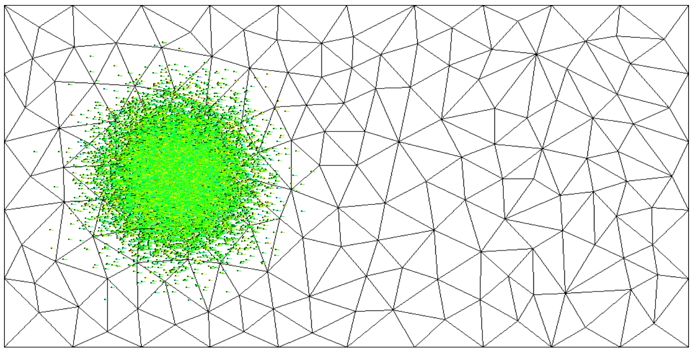


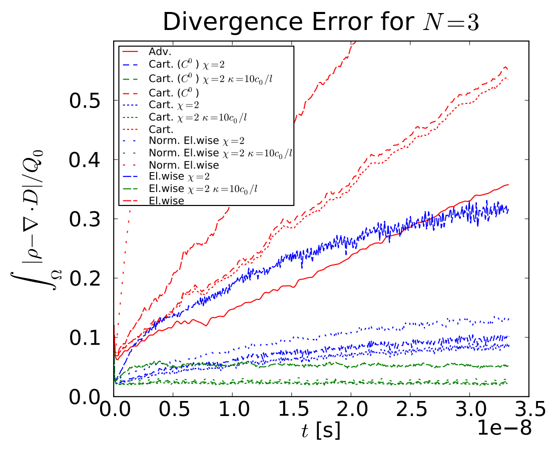
Figure 7.11. Spatial setting of the Gaussian electron beam test case.

each individual method in a beam physics setting. I will base this initial assessment on the simulation of the behavior of a simple Gaussian beam in a two-dimensional beam tube. The test setup is depicted in Figure 7.11. The transverse wall of the (2D) beam tube is a perfect electric conductor, and the longitudinal boundary is periodic.

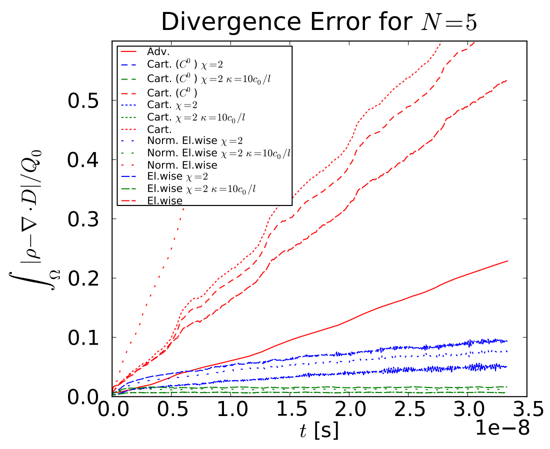
Unlike for the beam distribution in Section 7.9.2, no analytic predictions regarding beam behavior are available, so I will use the aforementioned physical invariants for evaluation.

The present section is based on a comprehensive survey of all methods at a wide variety of parameter settings. In the interest of brevity and readability, I will only present only the most salient results. Unless otherwise noted, all tests were run with on the mesh of Figure 7.11, containing $K = 302$ elements using a field approximation of polynomial degrees N of 3 and 5, with $P = 20000$ particles in the beam. The simulation is run for a time that allows the particle beam to cross the domain once along the beam-longitudinal x axis.

I begin my evaluation with a look at the methods' behavior in terms of the divergence error. This behavior is portrayed for polynomial degrees 3 and 5 in Figures 7.12(a) and 7.12(b) on the following page, respectively. These figures are perhaps the most important ones of the entire evaluation, as they a) depict performance on a quantity that is crucial for successful physics modelling (see Section 7.5) and b) show methods as having



(a) Divergence error for $N = 3$.



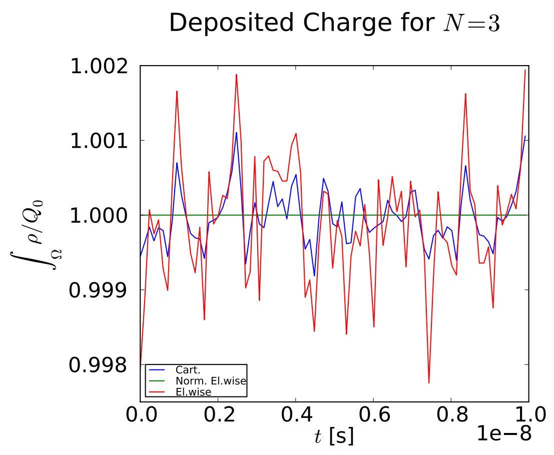
(b) Divergence error for $N = 5$.

Figure 7.12. Divergence error in the 2D Gaussian beam test. Each figure shows a plot of divergence error vs. simulation time.

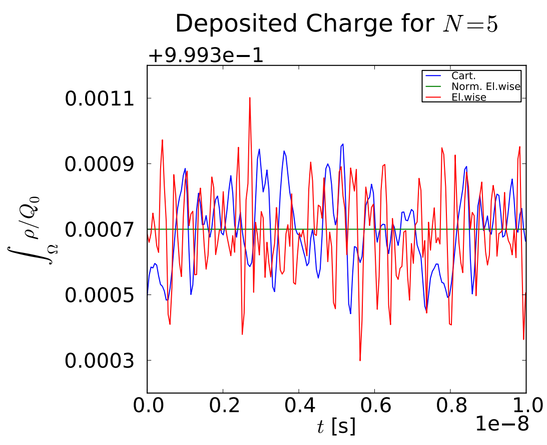
performance differences of more than an order of magnitude on this particular quality criterion. Regardless of the method, it appears that, as long as no corrective scheme is applied, divergence error accumulates nearly linearly, which is consistent with the picture of a beam moving through an area and leaving a trail of stationary noise behind. Because of this linear behavior, one may evaluate deposition methods based on the rate at which this accumulation occurs. I further observe that (as is plausible) particle pushing methods do not have any influence on the behavior of the divergence error, so results are shown for monomial particle pushing as a representative choice.

The first (and simplest) observation settles the question of post-deposition charge scale correction as brought up in Section 7.7.1. The line labeled “Norm. El.wise” and “El.wise” represent the same, element-wise deposition method, once without, and once with normalization. Normalization appears to contribute significantly to the accumulation of divergence error, increasing it by a factor of at least three in this instance. I shall therefore mostly exclude it from further consideration.

Results on whether continuity enforcement for Cartesian methods is beneficial are



(a) Deposited charge for $N = 3$.



(b) Deposited charge for $N = 5$.

Figure 7.13. Deposited charge in the 2D Gaussian beam test. Each figure shows a plot of deposited charge vs. simulation time.

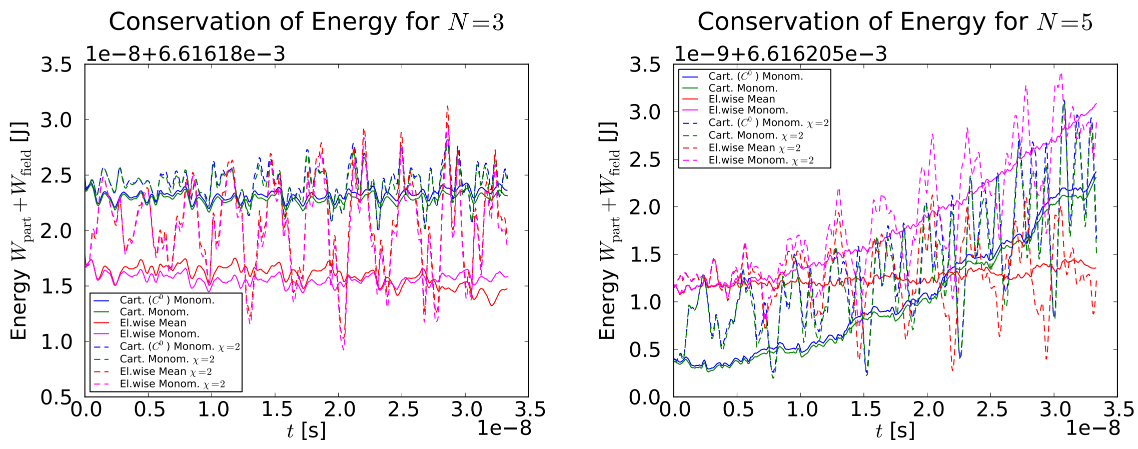
somewhat inconclusive, showing a beneficial effect at $N = 5$ and a smaller detrimental effect at $N = 3$. Similarly inconclusive results are visible for whether uncorrected Cartesian or element-wise deposition is superior. Cartesian deposition achieves lower divergence error by about a third at $N = 3$, whereas at $N = 5$, element-wise deposition has a small advantage. I hypothesize that the reason is this: the case $N = 5$ has considerably more expansion modes to choose from than the $N = 3$ one. In the low-order case, Cartesian deposition can benefit from its nearly uniform availability of resolution, and the troublesome grid-to-grid interpolation procedure cannot do to much damage by exciting very high-order modes. Unfortunately then, the availability of more modes at $N = 5$ seems to spell trouble for Cartesian deposition. This hypothesis about the trouble encountered by Cartesian deposition is backed by Figure 7.13, which depicts that the overall deposited charge for the same two cases, showing *greater* oscillations for Cartesian deposition at $N = 5$ than at $N = 3$. This hypothesis also affects the case for continuity correction of Cartesian deposition: It is based on a premise of smoothness and well-resolvedness. When that premise holds (e.g. at $N = 5$ above), Cartesian deposition becomes less viable. Continuity averaging therefore likely has no chance to be beneficial.

In discussing these results on Cartesian deposition, I would like to add that as nodal resolution increases from $N = 3$ to $N = 5$, Cartesian resolution is increased proportionally and kept at a level where the are $1.5\times$ more Cartesian than simplicial DG nodal points.

Lastly, advective deposition lives up to its design goal at least to some extent. Of the methods unaided by hyperbolic cleaning, it consistently achieves the smallest divergence error. At $N = 3$, advective deposition is competitive even with hyperbolically-cleaned element-wise deposition. As indicated in the section describing its construction, the reason why advective deposition does not provide near-perfect charge conservation is of course rooted in the need to truncate each particle's mini-domain. Unfortunately, even at these low threshold settings (which led to correspondingly poor charge conservation), advective deposition is rather expensive (in these runs, often by more than a factor of three over even the methods employing hyperbolic cleaning)–an expense that is not justified by the results obtained.

Moving on to methods aided by the use hyperbolic cleaning, the observations become more straightforward. A speed multiple of $\chi = 2$ appears able to help control divergence error at $N = 3$ equally well for all methods *except* for element-wise deposition. (Jacobs and Hesthaven [2006] recommend $\chi = 10$.) At $N = 5$, element-wise deposition is competitive with the other methods' post-correction divergence error. The addition of hyperbolic cleaning further slows the accumulation of divergence error to a below-linear rate. Once an additional decay term (denoted κ in the figure legend) is added, divergence error stops increasing in time and plateaus at about the same level for all methods.

As a final observation on divergence error, I would like to add that it is calming that a decrease in divergence error is observed in going from $N = 3$ to $N = 5$, confirming that approximation properties improve as resolution increases, even taking into account the caveat about modal resolution in Cartesian deposition above.



(a) Energy conservation for $N = 3$.

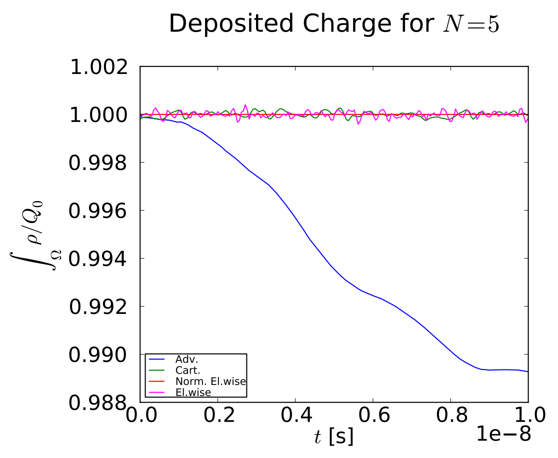
(b) Energy conservation for $N = 5$.

Figure 7.14. Energy conservation in the 2D Gaussian beam test. Each figure shows a plot of energy vs. simulation time.

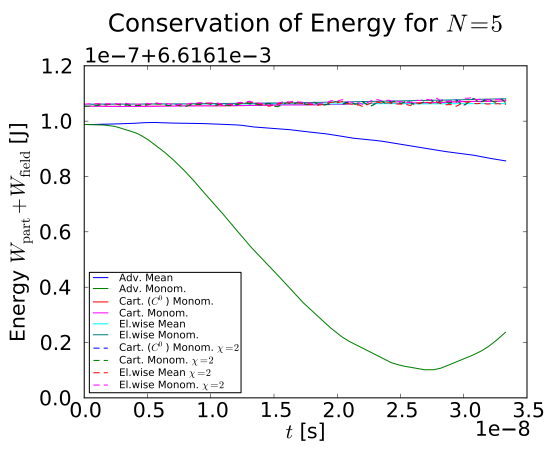
A further remark concerns the valid question of why I show results for a problem that appears underresolved. In my view, this is legitimate, and more interesting than showing only well-resolved results. In the simulation of realistic, challenging problems, one is very rarely afforded the luxury of being able to use as much resolution as one would like. To me, it is therefore most interesting to see what happens right at the boundary of underresolution and to observe how gracefully each method decays under these circumstances.

Results for the remaining two conserved properties, energy and momentum, shown in Figures 7.14 and 7.16 on the next page respectively, are somewhat simpler to analyze than the divergence error above. First, and most importantly, both are already quite good, representing ripples of a relative magnitude 10^{-5} and 10^{-4} for energy and momentum, at $N = 3$, and both again reduce considerably at $N = 5$.

Because of the already-small magnitude of the variations in energy and momentum, none of the following should be overvalued, but a few observations are definitely in order. First, it is uniformly the case that hyperbolic cleaning adds significant noise to both conserved quantities. This noise appears to be of oscillatory nature and does not

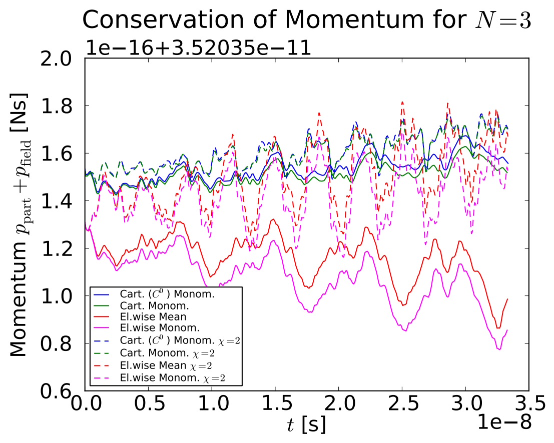


(a) Deposited charge vs. simulation time based on the same data as Figure 7.13(b) on page 216, with data for advective deposition included.

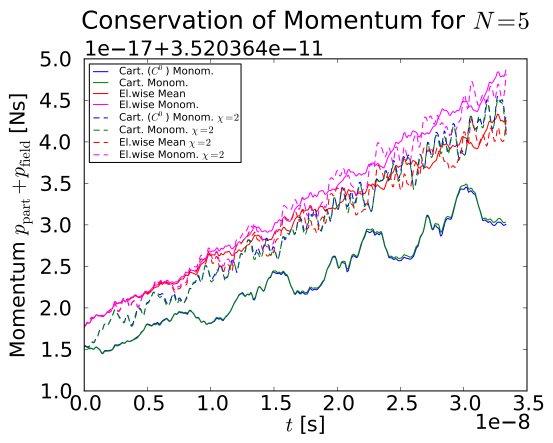


(b) Total Energy vs. simulation time based on the same data as Figure 7.14(b) on the preceding page, with data for advective deposition included.

Figure 7.15. A closer look at the performance of advective deposition.



(a) Conservation of momentum for $N = 3$.

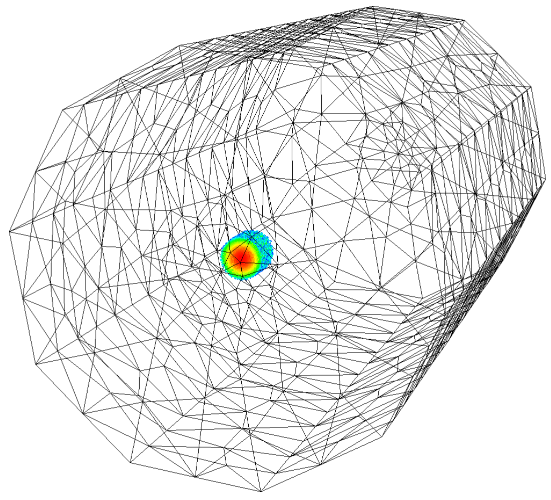


(b) Conservation of momentum for $N = 5$.

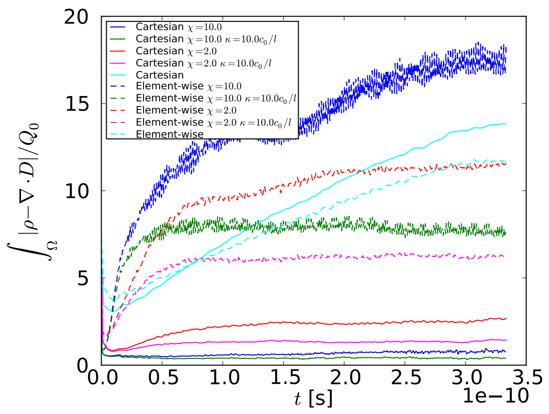
Figure 7.16. Conservation of momentum in the 2D Gaussian beam test. Each figure shows a plot of momentum vs. simulation time.

appear to alter the mean. Second, the overall smaller magnitude of variation at $N = 5$ uncovers drifts in both momentum and energy that may prove troublesome at longer time scales. Remarkably, mean-based pushing remains mostly free of this drift effect in energy at $N = 5$. Lastly, element-wise and Cartesian deposition exhibit very similar behavior in both of these conservation properties–while there is a visible behavior difference between methods, this is not so great that these differences should be called significant.

I would like to conclude this broad overview of performance data for all methods by briefly returning to observations on advective deposition. While it *did* provide the benefit it was designed for, it was already found to be uneconomical for its cost above. Unfortunately, that is not the method's only problem. Figure 7.15 on the preceding page shows a few more observations on advective deposition in the context of the quality measures discussed above. First of all, Figure 7.15(a) reprises Figure 7.13 on page 216. It was stated at the beginning of Section 7.5 that "charge conservation" is a misnomer for the effect for which it is used. Ironically, the figure shows that advective deposition has a *genuine* charge conservation problem, losing about one percent of the starting charge over the course of the simulation run. This, in turn, has grave effects on other conservation properties, as shown for total energy in Figure 7.15(b). Surprisingly, advective deposition coupled with monomial pushing performs far more poorly than advective deposition coupled with mean-based pushing, but, compared to the remaining methods, neither of them provides even reasonable output. While these results certainly seal the fate of advective deposition, that is not a big concern in itself, as the method was not designed to be practical on its own, but rather as a "test balloon" for more complicated methods that endow each particle with significant local structure. Given the results, it seems plausible that in addition to resolving challenges that the local structure might bring with it, one of the foremost questions such a method would have to answer is how to truncate each of its particle "mini-domains".



(a) Spatial setting for the K-V Beam Physics Test.



(b) Plot of L^1 Divergence Error vs. time for K-V Beam Physics with various methods.

Figure 7.17. Kapchinsky-Vladimirsky Beam Physics Test.

7.9.2 Kapchinsky-Vladimirsky Beam Physics

Kapchinsky and Vladimirsky [1959] described one of very few (perhaps the only) particle distributions in an electron beam for which an envelope equation–i.e. an ordinary differential equation governing parameters of the distribution, in this case the radii–can be derived for a fully self-consistent Vlasov-Maxwell system. Unfortunately, the particle distribution involves sharp, noncontiguous localization in phase space, and is therefore difficult to replicate experimentally.

The purpose of this test case is to verify accurate inter-particle forces as mediated by the grid-based field. The electron bunch will expand from its initial size, but maintain its initial distribution. The main observable is the RMS beam radius. An introductory treatment of the physics of this problem may be found in the books by Lee [2004] and Chao [1993, Chapter 1].

Setup for the K-V Test

The test takes place in a cylindrical, infinite tube of radius r_{tube}, aligned with the z axis. This is realized using a periodic mesh that should be of length greater than l_{tube}, as below. These and all further parameters occurring in the test are given in Table 7.1 on the following page. The spatial setting is illustrated in Figure 7.17(a) on the previous page. I would further like to remark that the shape of the outer beam tube cross-section is of not much importance, as long as it is sufficiently far away from the beam to prevent boundary interactions from having a major influence. This is so because the observed behavior is mainly a result of the interaction of the beam with itself, and not so much with fields scattered back from the boundary. The analytic predictions below are derived in infinite space.

Particles are distributed in transverse phase space according to the so-called *Kapchinsky-Vladimirsky* (or "K-V") distribution, which describes a uniform distribution on a spherical surface in the four dimensions x, y, x', y':

$$\frac{x^2}{r_x^2} + \frac{y^2}{r_y^2} + \frac{(x')^2 r_x^2}{\epsilon_{x,\text{tot}}^2} + \frac{(y')^2 r_y^2}{\epsilon_{y,\text{tot}}^2} = 1 \tag{7.32}$$

where $x' = p_x/p_z$ is the scaled speed. All particles have an identical velocity v_z in the z direction. The resulting x-y charge distribution is extended uniformly over a length l_z in z. This final density is then sampled randomly to obtain particle locations and momenta.

Aside from its importance to this particular test, the density resulting from (7.32) is a good example of a phase space distribution that is rather difficult to represent in a purely Eulerian setting and contributes to the justification of a Lagrangian treatment of the Vlasov-Maxwell system.

With the density f fixed, the initial electromagnetic fields E and H are found by solving

Quantity	Description	Value
r_{tube}	Tube radius	25mm
l_{tube}	Length of periodic mesh in z	100 mm
$\epsilon_{x,\text{tot}} = \epsilon_{y,\text{tot}}$	(Total) Emittance	5 mm mrad
Q	Bunch charge	-10 nC
$r_{x,\text{tot}} = r_{y,\text{tot}}$	(Total) Transverse bunch radius	2.5 mm
l_z	Longitudinal bunch length	5 mm
E_{el}	Relativistic energy of the electron, counting only v_z	5.11 MeV

Note: Bunch mass is determined by finding the number of electrons from Q and multiplying by the electron mass.

Table 7.1. Problem parameters for the Kapchinsky-Vladimirsky beam physics test of Section 7.9.2.

for the potential ϕ in the longitudinal rest frame of the particles and Lorentz-transforming the result back to the laboratory frame.

Expected Results

Analytically, the total beam radii

$$r_{x,\text{rms}} = \frac{r_{x,\text{tot}}}{2} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} r_{x,n}^2}$$

and $r_{y,\text{tot}}$ obey the ODEs [Lee, 2004, (2.140)]

$$r''_{x,\text{tot}} + K_x r_{x,\text{tot}} - \frac{2K_{sc,\text{tot}}}{r_{x,\text{tot}} + r_{y,\text{tot}}} - \frac{\epsilon_{x,\text{tot}}^2}{r_{x,\text{tot}}^3} = 0 \tag{7.33}$$

$$r''_{y,\text{tot}} + K_z r_{y,\text{tot}} - \frac{2K_{sc,\text{tot}}}{r_{x,\text{tot}} + r_{y,\text{tot}}} - \frac{\epsilon_{y,\text{tot}}^2}{r_{y,\text{tot}}^3} = 0, \tag{7.34}$$

where the "primed" derivative r'_x is with respect to the distance s travelled in z, K_x and K_y can be used to incorporate an external force and are otherwise 0, and the space charge constant K_{sc} is found by

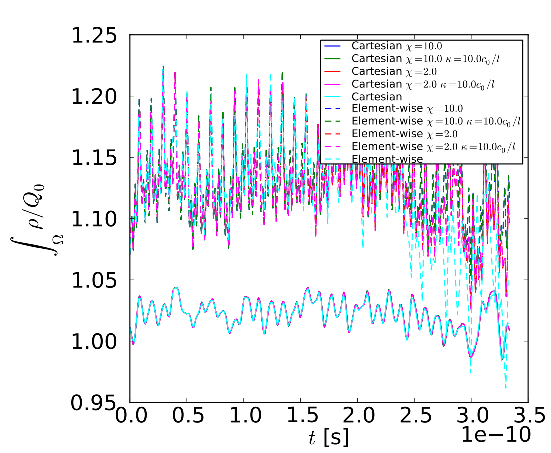
$$K_{sc,tot} = 2\frac{N r_0}{\beta^2\gamma^3},$$

where in turn N is the number of particles per unit length in z, $\beta = v/c_0$, c_0 is the speed of light in vacuum, $\gamma = (1 - \beta^2)^{-1/2}$ is the Lorentz factor, and r_0 is the classical electron radius.

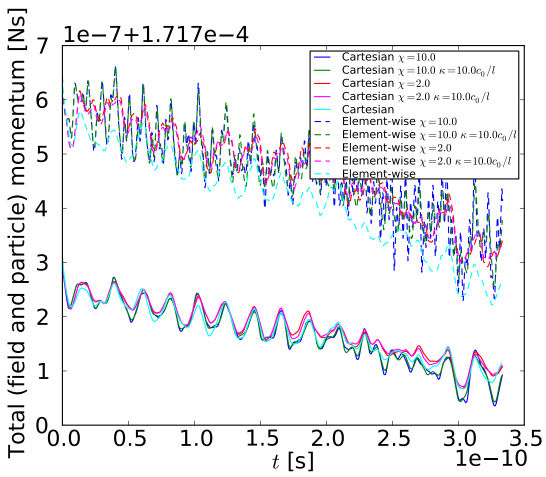
Computation Results and Evaluation

I examine results from a computation conducted in three dimensions on the mesh shown in Figure 7.17(a) on page 221. The transverse walls of the beam tube are perfect electric conductors, and the longitudinal boundary is periodic. While it is not clearly visible from the figure, the mesh is refined in the area where the beam travels. The electromagnetic field is discretized with P^3 polynomials and loaded with 20000 particles randomly sampled from the distribution described above. Based on the poor results of renormalized element-wise deposition and the expense involved in advective deposition, these two have been excluded from this test. I have further only examined the performance of interpolatory particle pushing. Simulation time was fixed to suffice for a full transition of the beam through the beam tube.

As above, I will begin by examining the L^1 divergence error of (7.30), shown in Figure 7.17(b) on page 221. There are a number of things that are remarkable about the plot. Despite being run at a resolution that would be considered moderate for pure electromagnetic simulation (6952 elements, ~ 30 particles per cell along the beam tube), the problem again appears markedly underresolved–a situation that is likely typical of a 3D PIC simulation with particles of non-zero support. Perhaps the first thing to note is the scale of the plot, which indicates that even the best-performing methods have divergence errors on the scale of the amount of charge present in the domain, whereas worse ones have divergence errors an order of magnitude higher. Repeating its success on underresolved problems, Cartesian deposition with any amount of hyperbolic cleaning results in much



(a) Plot of total deposited charge divided by intended charge vs. time for K-V Beam Physics, with various methods.

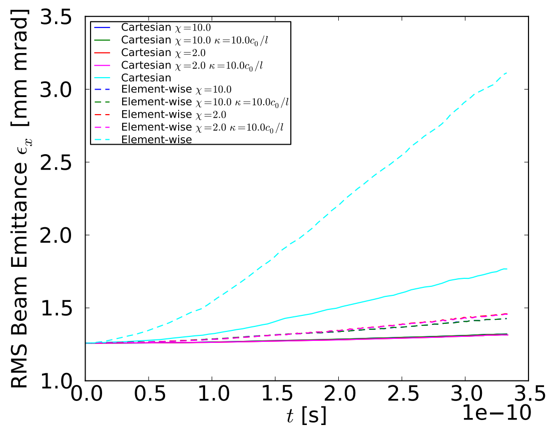


(b) Plot of total momentum vs. time for K-V Beam Physics, with various methods.

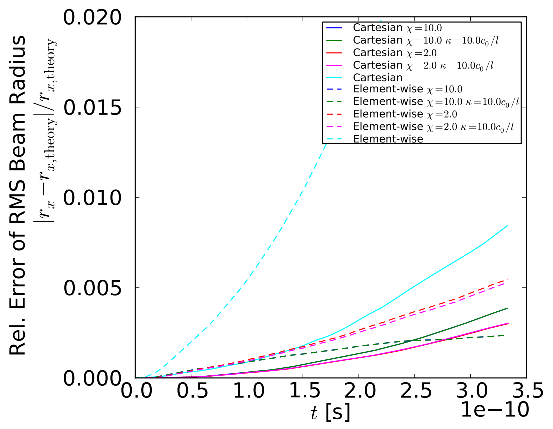
Figure 7.18. Conservation-based measures of solution quality for K-V Beam Physics.

lower divergence error than any other methods, by a factor of around three. Further within that group, we see that a cleaning wave speed factor $\chi = 10$ shows a measurable benefit over $\chi = 2$, and that the addition of a decay term can further help control divergence error, again leading to a reduction by a factor of around two at late times, regardless of original method and χ. It is puzzling (but consistently found across dimensionalities) that hyperbolic cleaning is much more effective on current densities generated by Cartesian than by element-wise deposition. In fact, hyperbolic cleaning in this instance is actively *counterproductive* for element-wise deposition. Only the case of $\chi = 10$ with decay in ϕ ($\kappa = 10c_0/l$) achieves a reduction of divergence error when compared to "plain" element-wise deposition.

A measurement of the ratio of deposited vs. intended charge, i.e. the quantity of (7.31), is shown in Figure 7.18(a). Both Cartesian and element-wise methods show significant oscillations in the amount of charge that arrives on the mesh, however it appears that the Cartesian deposition achieves somewhat better control over the issue than element-wise deposition, which, at times, deposits 20 per cent more or less charge than intended, at this



(a) Plot of the emittance of electron beam vs. time for K-V Beam Physics, with various methods.



(b) Plot of the relative error in the RMS K-V beam Radius vs. time for K-V Beam Physics, with various methods.

Figure 7.19. Physics-based measures of solution quality for K-V Beam Physics.

particular combination of resolutions, whereas deviations for Cartesian deposition stay limited to roughly 5 per cent, again speaking to its superiority on underresolved problems.

Figure 7.19(b) shows the relative error in the main quantity of interest in this test, the root-mean-square beam radius. Uncorrected, Cartesian deposition generates an error that is smaller by a factor about five than element-wise deposition. Surprisingly, after applying the maximum level of correction ($\chi = 10$, $\kappa = 10c_0/l$), the situation is reversed, and element-wise deposition is the deposition method whose relative error displays the least amount of worrisome upward trend. At this point I have no explanation for this behavior. In describing the results, I should note that particle sampling error was corrected out of the theoretical beam radius result. In other words, once the particle locations were sampled from the distribution described by (7.32), the starting radius of the envelope equation (7.33) was calculated from the RMS radius determined from the sampled distribution. Without this correction, the error would have started at a non-zero value and would not have admitted any valid conclusions.

Figure 7.19(a) displays RMS emittance, of (7.29), another measure of simulation quality, that should be identically conserved in time. Unlike RMS beam radius, this criterion again shows Cartesian deposition to yield much smaller increases, and hence better results, in both corrected and uncorrected form.

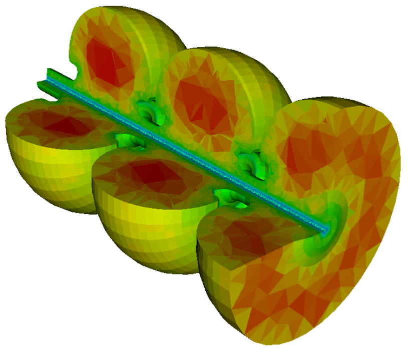
Lastly, Figure 7.18(b) on page 225 shows a plot of beam momentum vs. time, with results that very much mirror that of Section 7.9.1: Conservation is already quite good (if somewhat noisy), but a downward trend (compare the upward one of the previous section) is visible. Fortunately, given the scale of the effect, it would take a long time to actually become troublesome even if it were to continue unabated.

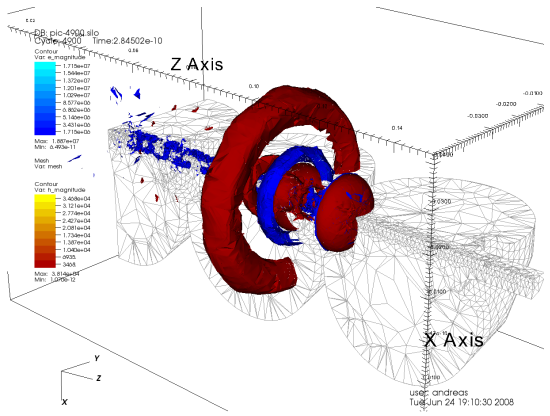
In summary, it appears that particle-in-cell solution quality is quite multi-faceted. While many results agree on which deposition and pushing methods yield better results, some unexplained disagreements do exist. Going forward, along with further improving the method, I am convinced that it is important to understand the factors influencing these error measures in more detail, so as to better understand along what axes the quality of a PIC simulation result needs to be evaluated.

7.10 A Moderate-Scale Application

As a final computational example, I would like to highlight a larger-scale computation, whose focus was not to emphasize the performance of any given method, but to demonstrate that the technology built for this project can already be used for moderately large computations. While the field solver is already parallel-capable, as we have seen, the particle-in-cell component is still very much in its experimental stage and has therefore not been parallelized, limiting the size of the application.

The setting for the computation is an experimental 2.5-cell injector cavity under investi-


(a) An experimental 2.5-cell resonant cavity used in the injector test stand at the Advanced Photon Source (APS) at Argonne National Laboratory. Cells are colored by their size, highlighting mesh refinement.



(b) Contour Plots of the electric field E (blue) and the magnetic field H (red) at one point during the passage of an electron bunch through the experimental ANL 2.5-cell cavity.

Figure 7.20. Application of DGTD-PIC to an experimental injector cavity developed at Argonne National Labs.

gation at the Advanced Photon Source at Argonne National Labs, shown in Figure 7.20(a). The simulation was not physically realistic as it was missing two components: An emission model for the laser-target electron source at the center of the half-cell in the near end of Figure 7.20(a) and a driving electromagnetic cavity mode that would have to be computed separately. Instead, the simulation captures the evolution from situation where a 10nC Kapchinsky-Vladimirsky beam (see Section 7.9.2) is launched in the center of the half-cell towards the emitting end of the cavity at 5.11 MeV and then undergoes interaction with its wakefield. The constrictions in the geometry cause significant beam-field interactions. One snapshot of the simulation is shown in Figure 7.20(b), with contours of the electric field shown in blue, and contours of the magnetic field shown in red. In the trail of the beam, significant pollution from divergence error (see Section 7.5) is visible.

As mentioned, the purpose of this computation was not necessarily to provide physically meaningful results, but rather to confirm that the solver technology created is capable of tackling moderate-scale applications, in this instance 20000 particles on a mesh of 71000 tetrahedral elements at polynomial degree 3. This result gives me confidence that, once the methods presented throughout this chapter have further matured, scaling of the solver will be a solvable problem.

7.11 Conclusions and Future Work

Throughout this chapter, I have investigated methods that I hope will eventually lead to a usable, robust coupling between charge-carrying particles and a high-order unstructured discontinuous Galerkin scheme. The mitigation of divergence error (Section 7.5) remains one of the foremost challenges in this endeavor. My computational experiments have shown that the figures of merit outlined in Section 7.9 cover largely independent points of view under which a PIC scheme should be evaluated, though at present the connection between features of the scheme and observed error profiles is poorly understood. One of my aims for future work is to improve this understanding, to enable users of this technology to make informed solver choices based on which aspects of solution quality are essential to them, and which ones are dispensable.

Another challenge in PIC is that of proper resolution management. Many different types of resolution are available, such as mesh size, particle count, degree of approximation, and particle radius. Some hints on connections between these are available, but a global understanding of them, so far, is largely missing.

It is further likely that the set of criteria in Section 7.9 is not exhaustive, i.e. it does not cover all aspects under which solution should be judged. Like in any attempt of empirical validation, much depends on both the set of test cases and the set of measurements taken. Both of these sets are far from finalized.

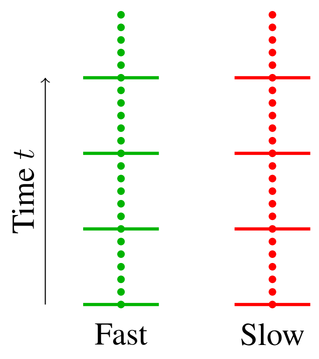
To help enlarge the size of the test set and enable a common ground of comparability for implementers of numerical methods for the Vlasov-Maxwell system, I have initiated and continue to maintain an Internet-based database of test cases. In addition to my own contributions to this database, colleagues from Universität Stuttgart in Germany have made use of and taken part in the creation of the repository, which can be found on the web at `http://webapp.dam.brown.edu/piki`.

It has become clear throughout this chapter that DG-PIC is a challenging problem. I hope to have contributed to the solution of this problem by offering a number of choices for coupling methods with DG field solvers, and by discussing aspects of performance evaluation for them.

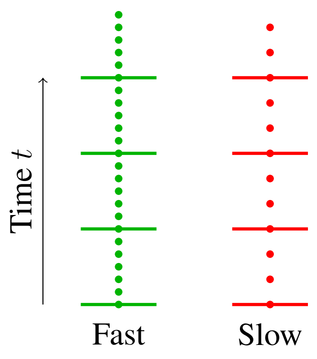
But it is important in my view that PIC is not just a challenge for its own sake–a competitive DG-based scheme for Vlasov-Maxwell would have a potentially broad impact in terms of its direct plasma physics applications. In addition, the problem touches many areas of current numerical research from ODE solvers to particle-field interaction and raises questions whose answers are important in a much larger context.

CHAPTER EIGHT

Multi-rate Time Stepping: Methods and Applications



(a) Single-rate time stepping. State is computed for all components at every time step.



(b) Multi-rate time stepping. State for slow components is only computed at certain intervals.

Figure 8.1. The temporal setting of multi-rate time stepping. Each dot corresponds to a computed fast or slow state.

8.1 Introduction

Most time-domain simulations involve processes on a number of different time scales. In explicit time marching schemes, the fastest processes limit the maximal size of the time step, necessitating that slower processes use an unnecessarily small time step. This is wasteful. Examples of this abound, and I will discuss a few of them in Section 8.6.

It makes sense to try to avoid this waste, and numerous ways are available to do so. First of all, most implicit time steppers can use arbitrarily large time steps, at the risk of not resolving the fine time scales. Operator splitting techniques can be used to the same effect, as can combined Implicit-Explicit methods [see, e.g., Ascher et al., 1995, and references therein]. On the simpler end of the spectrum, a variety of multi-rate explicit time-steppers for both multi-stage and multi-step schemes have been devised [Andrus, 1979, Engstler and Lubich, 1997, Gear, 1974, Sandu and Constantinescu, 2009]. This chapter deals with a number of versions of the latter.

To the best of my knowledge, the first mention of multi-rate multi-step time integrators in the literature was by Gear [1974]. An attempt of an analysis of the method was undertaken by Gomm [1981]. Gear and Wells [1984] then focused on building an algorithm that would automate the application of multi-rate time stepping as much as possible. Similar automation efforts have been a recurrent theme in the literature [Engstler and Lubich, 1997]. Inspired by discussions with Warburton [priv.comm.], and in extension of the work by Gear and Wells [1984], Stock [2009] and I have discovered that more schemes satisfying the order conditions exist than just the two explored in the original reference. All of these schemes are mathematically different, and thus have somewhat different stability properties, as will be seen in Section 8.5.

8.2 The Setting for Multi-Rate Multi-Step Methods

In order for a multi-rate scheme to make sense, at least two different time scales must be present in the ordinary differential equation (ODE) whose initial value problem (IVP) is to be solved. Like Gear and Wells [1984], I will assume that this time-scale separation occurs *by variable*, i.e. that one may speak of "fast" variables f and "slow" variables s. Hence one is dealing with an at least 2-variable system of ODEs. Unlike Gear and Wells [1984], and like Warburton [2008], I will assume an additive separation of right-hand sides into coupling and self-influencing terms, leading to a 2×2 autonomous system of ODEs:

$$\partial_t \underbrace{\begin{pmatrix} f(t) \\ s(t) \end{pmatrix}}_{q:=} = \begin{pmatrix} a_{ff}(f) + a_{fs}(s) \\ a_{sf}(f) + a_{ss}(s) \end{pmatrix}. \tag{8.1}$$

To be able to solve the initial value problem, of course a set of initial values

$$f(0) = f_0, \qquad s(0) = s_0 \tag{8.2}$$

needs to be specified. For simplicity, I will assume here that a_{ff}, a_{fs}, a_{sf}, and a_{ss} are smooth and that the IVP consisting of (8.1) and (8.2) has a unique, smooth solution.

As will be seen in Section 8.6, such splitting is natural in many application situations. Unfortunately, it is not always possible. If what is here labeled a 'coupling term' involves a nonlinear expression of both f and s, or even if it is simply inconvenient from a software perspective, a user of the method may choose to somewhat dilute the distinction between self- and coupling terms by allowing one or both to depend on both f and s, resulting in the system

$$\partial_t \underbrace{\begin{pmatrix} f(t) \\ s(t) \end{pmatrix}}_{q:=} = \begin{pmatrix} a_{ff}(f) + a_{fs}(f,s) \\ a_{sf}(f,s) + a_{ss}(s) \end{pmatrix} \tag{8.3}$$

or even in

$$\partial_t \underbrace{\begin{pmatrix} f(t) \\ s(t) \end{pmatrix}}_{q:=} = \begin{pmatrix} a_{ff}(f,s) + a_{fs}(f,s) \\ a_{sf}(f,s) + a_{ss}(f,s) \end{pmatrix}. \tag{8.4}$$

Observe that by (8.4), the name "coupling term" has become entirely meaningless mathematically and continues to exist only in spirit. I would like to add that, given the knowledge available at this writing, there is no strong reason to prefer the perceived "rigor" of (8.1) over the "sloppiness" of (8.3) and (8.4). Accuracy results continue to hold in each case, and Section 8.5 examines stability for a 2×2 linear test system, with only circumstantial evidence that similar results continue to hold for larger systems. Thus no stronger results are available even in the "rigorous" case of (8.1). For full generality, this chapter presents schemes applicable to all three types of systems. Below, I will refer to the occurrence of such systems as *partial* and *full dependency mixing* ((8.3) and (8.4), respectively).

Without loss of generality, I will assume in this chapter that f and s are scalars, though unless otherwise noted, everything I say (aside from statements about stability) generalizes trivially to vector-valued f and s.

With coupling neglected, the basic assumption for a sensible application of a multi-rate scheme is that a_{ff} requires f to be integrated with a small time step, while a_{ss} has a less rigorous requirement for s. I make the further assumption that, in some sense, a_{sf} "shields" the slow component s from the fast evolution of f, whereas a_{fs} is generally unconstrained.

Before beginning a discussion of multi-rate integrators, I would like to remind the reader of the functioning principle of multi-step explicit time integrators for the approximation of the solution y of the IVP

$$\partial_t y = f(y(t)) \qquad y(t) = y_0.$$

For easier comparison with later modifications, the entire method is shown in Algorithm 8.1.

Algorithm 8.1 Explicit k-step time stepping method.

```
Require: $y_0$: Initial condition
Require: $f$: Right-hand side
  By some start-up method, obtain
  - $y_h^k$
  - start-up history: $H \leftarrow ((f(y_h^0), \dots, f(y_h^{k-1}))$.
  for steps $i = k$ to $N - 1 - k$ do
      Construct an interpolant $\tilde{f}$ such that $\tilde{f}(t_j) = f(y_h^j)$ ($j \in \{i-k, \dots, i-1\}$)
      $y_h^{k+1} \leftarrow y_h^k + \int_0^{\Delta t} \tilde{f}(t)dt$
      $H \leftarrow (H_{2,\dots,k}, f(y_h^{k+1}))$
  end for
```

By far the most common explicit multi-stage time stepping method is the Adams-Bashforth method, which accomplishes the construction of $\tilde{f}$ in Algorithm 8.1 by polynomial interpolation, as illustrated in Figure 8.2.

Algorithm 8.1 offers significant flexibility that can be exploited for the design of a multi-rate scheme:

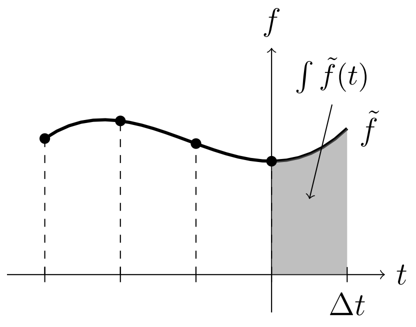


Figure 8.2. Functional principle of extrapolatory multi-step methods.

- It is possible to have separate histories for each variable.
- But not only that: Because of the linearity of the integral, it is possible to have separate histories for different (additive) parts of the right-hand side for a single variable.
- Once two histories have been separated, they can be used in integrations over differing time intervals.

When I say "different histories" above, I am not only referring to the possibility of separate storage of history data. Rather, this separate storage makes it possible to give each history different properties–such as the intervals at which right-hand side values are computed and saved to history, or its length and thereby the accuracy to which interpolation and integration can be carried out.

8.3 Design Choices in Multi-Rate Multi-Step Methods

Section 8.2 indicated the availability of some freedom regarding treatment of right-hand side histories. The next goal in this chapter is to explore the design space that these simple

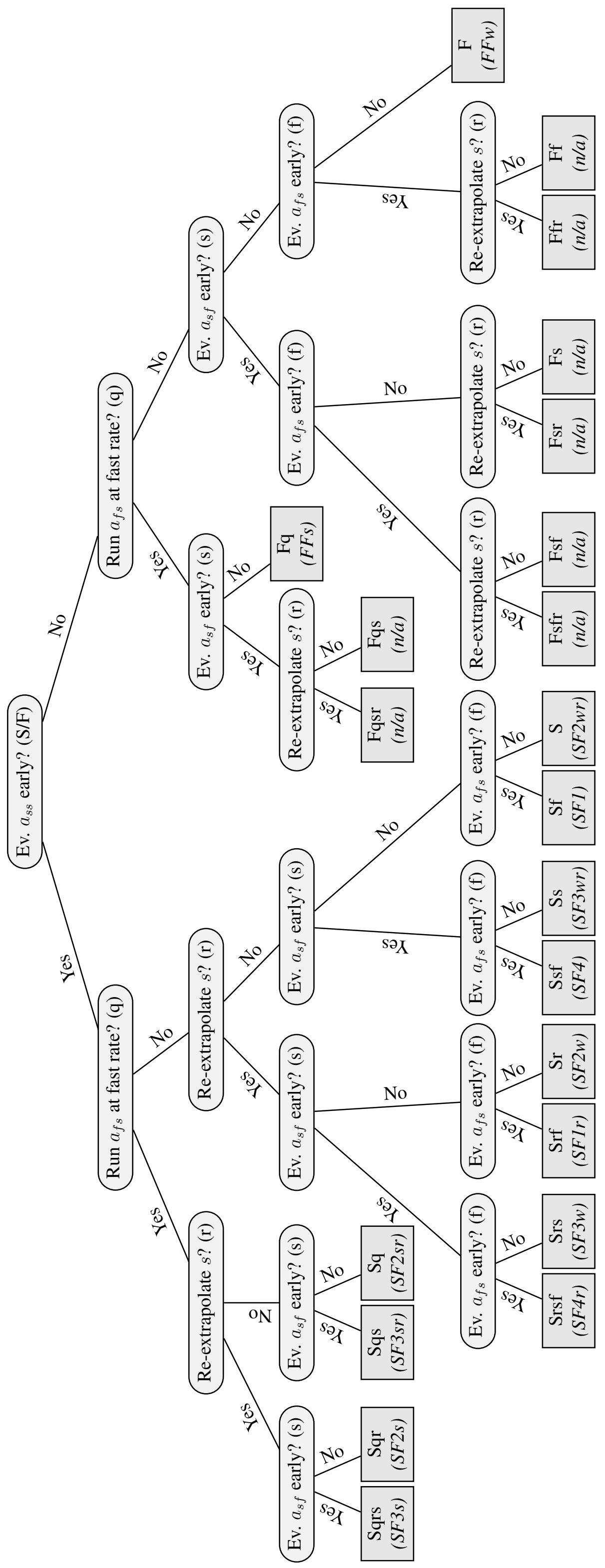


Figure 8.3. Decision tree for the design of multi-rate multi-step schemes. A systematic name for each scheme is given in each of the decision tree's leaves. The nomenclature used by Stock [2009] is given in parentheses in each leaf node.

observations open up for multi-rate time stepping schemes. While Gear and Wells [1984] group by right-hand side and thereby employ only two histories of right-hand side values for the entirety of (8.1), I choose to keep separate histories for each right-hand-side term in (8.1), and thereby end up with four history lists. Section 8.6 will show examples of why this is reasonable: In some applications, much of the expense of integrating the system lies in the coupling terms, so that it makes sense to separate them out and make them individually controllable.

For each such history of right-hand-side values, I will choose one of two of two rates, the fast and the slow rate. The fast rate will be an integer multiple of the slow rate, and their ratio is called the *step ratio* k. Suppose the time step required by a_{ff} is $\tilde{h}$, and the time step required by a_{ss} is $\tilde{H}$, then the hope is that $k = \lceil \tilde{H}/\tilde{h} \rceil$, and that the global "slow" time step H can be chosen as $H \approx \tilde{H}$, giving rise to the "fast" *sub-step* size $h = H/k$. In contrast to the sub-step, a step of size H will be called *full step* or simply *step*. While the step ratio needs to be chosen a priori, it will remain an independent, unconstrained parameter in the remainder of this text.

Once the two rates have been decided upon, the next question is which right-hand side is evaluated at which rate, and which operations to perform in which order. In making these decisions, one tries to adapt the multi-rate scheme to the problem at hand. It would of course be desirable that none of these choices had to be made, but having to make them is still better than not knowing that choices are available. Further, Section 8.5 will provide some (empirical) insight into which decisions are suitable for which problems. A decision tree of all available choices is given in Figure 8.3, and the choices are described in the following narrative, somewhat out of tree order.

I begin by matching rates to right-hand-side terms. Two of the rate choices are obvious, in that a_{ss} will run at the slow rate, and a_{ff} will run at the fast rate. For the off-diagonal coupling terms, the choice of rate is less clear. Since the premise of the method is that the component s will evolve slowly, it appears plausible that the term a_{sf} coupling s to the fast term will also be evaluated at the slow rate. For the opposite coupling term, a legitimate case can be made for both types of evaluation, and I leave this up to the user as a valid design choice (Choice “q” in Figure 8.3).

Next, one encounters a number of ordering questions. Gear and Wells [1984] used a coarser-grained decomposition of (8.1) and thereby have fewer choices, but do address one ordering aspect: They allow a choice between a “fastest-first” and a “slowest-first” scheme. This is reflected in our finer-grained model by the choice of whether to evaluate a_{ss} at the beginning or at the end of the step. (Choice “S/F” in Figure 8.3) But this is far from the only non-uniqueness. One can similarly ask whether a_{sf} and a_{fs} should be evaluated early. (Choices “s” and “f”, respectively, in Figure 8.3.)

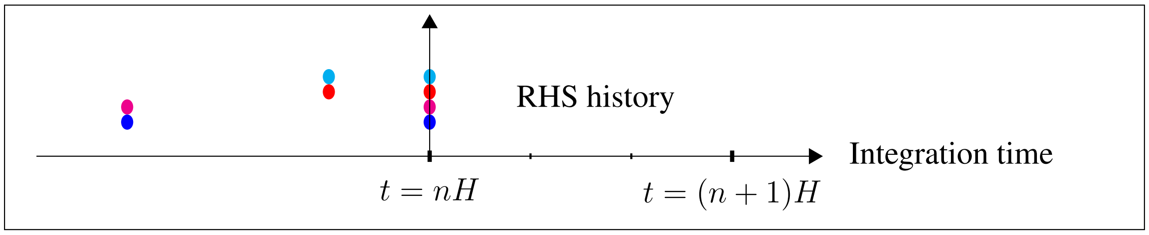
In each instance of the evaluation ordering options, one might expect the late evaluation of slow components to be “superior” because it can make use of additional information gained during the evaluation of the fast part. This is particularly true of the early evaluation of a_{fs}, as it naturally relies on the fast component f, which needs to undergo an extrapolation of size H to the point in time at which a_{fs} is to be evaluated, even though much of its history progresses in steps of h. Such an extrapolation is unusually large in terms of conventional multi-step methods. If however the dependence of a_{fs} on f is sufficiently weak, this may not present much of an issue. Unusually long extrapolations also occur if dependency mixing is encountered, as this again creates an early dependence on values of f. Despite their unusualness, schemes with early evaluation of slow components should not be simply disregarded. The computational experiments of Section 8.5 present evidence that their stability properties are not necessarily inferior, and in some cases even superior, to their fast-early counterparts.

Lastly, one additional option is available if one or more of a_{ss} and a_{fs} are evaluated early (Choices “S” and “f” in Figure 8.3): Since the evaluation of a_{ss} and a_{fs} depends on a value of s, an extrapolation must be performed to make it available. This extrapolation depends on historic values of a_{ss} and a_{sf}. If, in addition, a need for a late value of s exists, either through dependency mixing or late evaluation of one of a_{fs} or a_{ss}, one may wish to let later right-hand-side evaluations benefit from updated data and, in order to do so, perform a “fresh” extrapolation of s. This choice of re-extrapolating is represented as “r” in Figure 8.3 and was, in slightly modified form, already present in the dissertation of Wells [1982].

This section concludes a brief, first exposition of the choices to be made. I am postponing a more comprehensive study of the available methods to the end of the next section, when the tools to accurately describe them will be available.

8.4 Notation and Building Blocks

A close look at Algorithm 8.1 reveals the building blocks from which single-rate multi-step schemes are built. Multi-rate schemes are built from the same basic building blocks, taking into account the greater number of right-hand side functions and histories of right-hand side values. The present section specifies each of those building blocks and introduces a graphical representation that helps uniquely specify a scheme and allows users of the method to quickly gain an overview over the steps performed by the method.



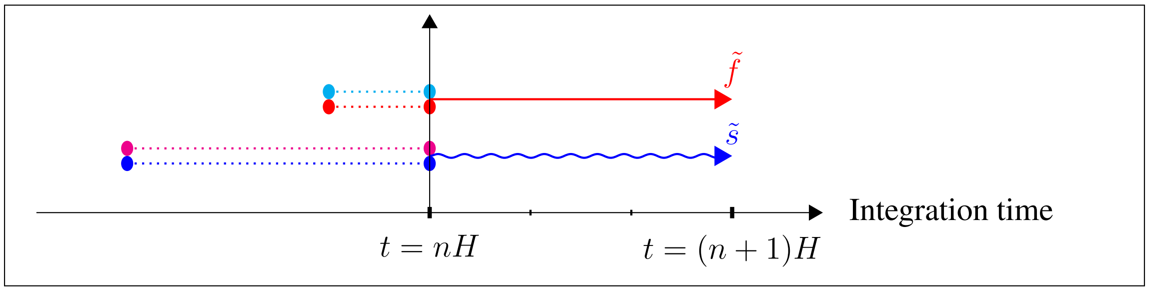
- ● slow-to-fast coupling (a_{fs})
- ● fast self-term (a_{ff})
- ● fast-to-slow coupling (a_{sf})
- ● slow self-term (a_{ss})

Figure 8.4. Color key for the right-hand sides involved in the multi-rate multi-step scheme of the system (8.1).

The starting state of each multi-rate AB is summarized by the points in time for which historic values of the right-hand-side functions a are available. Each value is shown as a colored point in a two-dimensional diagram. The points are color-coded for the right-hand-side components they belong to. A color key can be found in Figure 8.4. The diagram's horizontal axis represents simulation time. The vertical axis uses large increments to indicate computational order and small increments to denote structure within each computational step. Therefore, the depiction of the initial history state yields an easy way of distinguishing schemes that propagate a_{fs} at a fast rate (Choice "q" in Figure 8.3).

Note that at the outset of a time integration, the initial data f_0 and s_0 of (8.2) are available, but no full history as pictured above. To obtain this history, a single-rate, potentially multi-stage method of the same approximation order as the started multi-rate multi-step method may be used.

For the purpose of easier visualization, I will show all schemes (or parts thereof) at a sub-step ratio of $k = 3$, and a uniform history length of two, which would let the illustrated scheme's local truncation error decay as H^2.

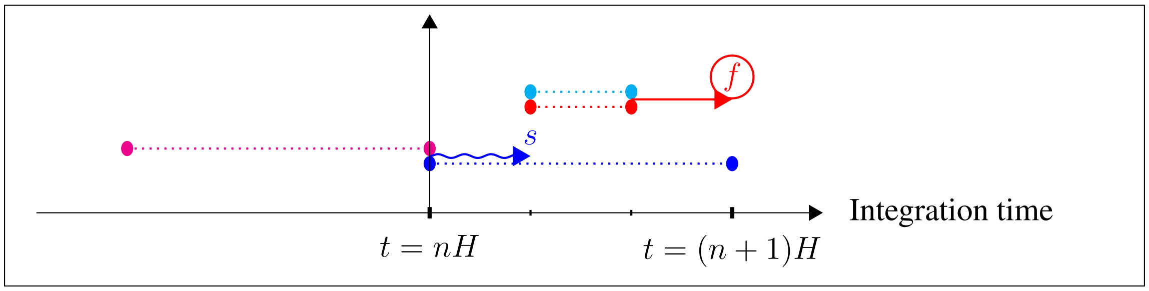


The key ingredient in single-rate multi-step time integration methods is the construction of a function interpolating the right-hand-side values obtained previously and the use of that approximate right-hand side function to compute the integral shown in Figure 8.2, whose value can then be used to advance the state y (in the notation of the Figure). Somewhat sloppily, I will call this entire process (including the evaluation of the integral) the *extrapolation* of a certain time state using specified right hand side data.

To perform an extrapolation, one needs to decide on a basis in which to express the interpolant. A common choice is that of polynomials, which, in the case of single-rate multi-step methods, leads to the well-known Adams-Bashforth methods [Bashforth and Adams, 1883]. Given that the methods I am targeting in this chapter are designed to have an order of accuracy no greater than perhaps six, polynomials are an adequate choice. For this reason, the methods of this chapter may also be called *multi-rate Adams-Bashforth* (or *MRAB*) methods.

I would now like to direct the reader's attention to the specifics of extrapolation in a multi-rate method, for which I would like to refer to the illustration above. Because of the split structure of (8.1), each advancement of the state f or s requires the evaluation of *two* extrapolations of the type of Figure 8.2, one for the self-influencing component, and one for the coupling component. The history data used in a state advancement is (as above) shown as colored dots obeying the color key of Figure 8.4, connected by a dotted line of the same color. Note that there are two stacked history lines, with the coupling term found at the top, and the self-term found at the bottom. This set of history data then determines the function to be integrated (the sum of the two interpolants). Next, one needs to know the interval over which the integration is performed, or, equivalently, the stretch of time over which state is advanced. This information is indicated by a red, straight line for the fast state f, and a blue, curving line for the slow state s. The last piece of information contained in the symbolized extrapolation in the diagram is the equivalent of a variable assignment. This variable will always be f for the fast state and s for the slow state s, but modifiers such as tildes may be present if storing additional state is required by the scheme. The purpose of these “variable assignments” is that their values may be conveniently referred to in upcoming steps of the method. Unfortunately, the present notation somewhat obscures the keeping of this state in favor of other information–but in practice I have not found this to be a major obstacle to the use of the diagrams for understanding MRAB methods.

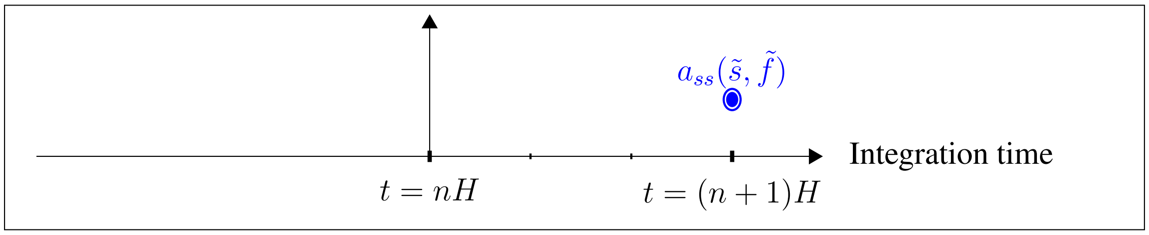
Lastly, I would like to note that MRAB schemes are linear methods, and once the history positions and the integration interval are known, extrapolation is most efficiently carried out as a linear combination of history values and state, with coefficients that are trivially precomputable.



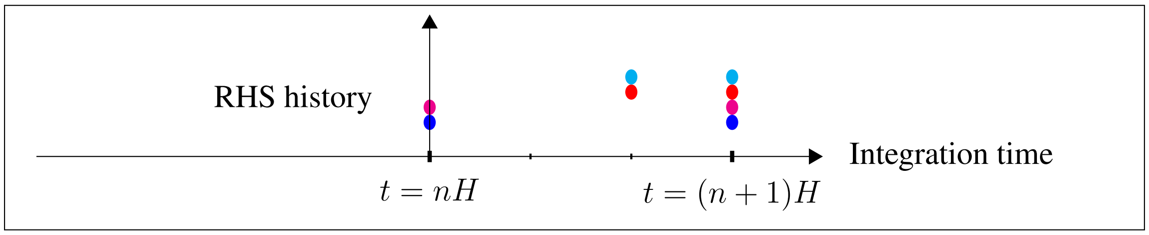
A somewhat more complicated example of a pair of extrapolations is shown in the above diagram. It illustrates two additional details that are captured by the notation.

The first such thing is illustrated in the extrapolation of the slow state s, above, and is genuinely unique to multi-rate methods. It presents the case where one set of history data (out of the self- and coupling terms) has progressed beyond the point in time at which the resultant state is needed. In some sense, the scheme is moving backwards in time. This occurs in all the slowest-first (“S”) schemes, to varying extents. In terms of stability, which will be discussed in more detail in Section 8.5, this might be beneficial, as interpolation is, from a numerical point of view, a more stable operation than extrapolation.

The last notable feature of the extrapolation notation is the highlighting of the final state. Recall that the purpose of the time integrator is to advance the two states f and s in time. As such, at the end of each step, not just history is carried forward to the next step, but also one purposefully chosen set of states f and s. These step-final states are shown with appropriately-colored circles around them, as shown for f above.



If one views the extrapolation discussed so far as the “mortar” that turns existing history values into new state, then the “bricks” are supplied by right-hand-side evaluations, which, compared to extrapolations, are fairly simple objects. They are represented in the notation as an encircled, colored history dot at a certain time that, additionally, indicates in a formula which right-hand side is evaluated with which arguments, drawn from the state space of variable assignments discussed earlier.



At the conclusion of a multi-rate step, after a potentially complicated, looping combination of extrapolations and right-hand-side evaluations, history state reaches the same configuration in which it started, advanced by H in time, and the next time step can begin.

It was already observed in the previous section that a MRAB scheme can have significant internal structure and embodies a number of design choices. Now that the tools to precisely specify an MRAB scheme are available, I would like to revisit the design decision tree in Figure 8.3 and the decisions axes described in the previous section. The first observation that I would like to make that the tree does not include every possible combination of choices, in particular:

No re-extrapolation of s ("r") if no early evaluation occurs. If no early evaluations occur, s is only computed once, late.

No early evaluation of a_{fs} ("f") in fast-a_{fs} ("q") schemes. Since fast-a_{fs} implies that a_{fs} is evaluated at each sub-step, "early" evaluation loses its meaning.

Next, I would like to add that the design choices discussed in this chapter are those that, in my opinion, seemed promising. In other words, more freedom exists than is exploited by the enumeration–for example, instead of always using the newest available history data, which can sometimes lead to "retrograde" state updates (as discussed above), one might choose to use older history data for some of these updates, or one might choose to run one of the coupling histories at yet another different rate. Therefore, while I have tried to cast as wide a net as is practical, I make no claim regarding the exhaustiveness of the enumeration of methods.

Now that an exact visual representation of MRAB schemes is available, it is of course interesting to see what the schemes designed previously look like in terms of that representation. Since there are 22 schemes overall, I shall here present a representative subset that shows the main features, so as to not unnecessarily clutter up the presentation in this chapter. The majority of the schemes is shown (in slightly different notation) by Stock [2009]. Further, a comprehensive survey containing all methods can be generated by a script included with the DG solver `hedge` (see Chapter 3) and is also available from the author upon request.

The brief tour of MRAB schemes begins with the perhaps simplest choice, the scheme "F", shown in Figure 8.5 on page 247. It delays all evaluations to the latest possible point and runs a_{fs} at a slow rate. Next, the scheme "Sq" introduces one early evaluation (that of a_{ss}), runs a_{fs} at a fast rate and therefore features some retrograde extrapolations. (see

Identical schemes
Sf, Ffr, Ff
S, F
Ssf, Fsf
Ss, Fs

Table 8.1. Scheme reduction for unmixed dependencies. The schemes in each row are mathematically identical if no dependency mixing (cf. (8.3) and (8.4)) is occurring.

Figure 8.6 on page 248) Lastly, "Srsf" performs the maximum number of evaluations early and introduces the re-extrapolation for the late evaluation of a_{ff} (instead of re-using the early value $\tilde{s}$). (see Figure 8.7 on page 249)

In concluding my comments on the available design choices, I would like to point out one further fact about the range of schemes presented above. When no dependency mixing (cf. (8.3) and (8.4)) occurs, a few of the schemes in Figure 8.3 become mathematically equivalent. These schemes are listed in Table 8.1. As a result, for the entirely unmixed system (8.1), 17 of the 22 choices remain distinguishable.

8.5 Accuracy and Stability

Two characteristics critically determine the practical applicability of an explicit time stepping method–its order of accuracy and its stability properties. Both will be discussed in this section for the methods above.

Before I begin this discussion, I would like to remark that for all results shown within this section I will let each component operate at the same approximation order for the sake of simplicity. This assumption is not necessary, and if the user deems it advantageous to break it, this is not difficult to achieve, as the only modification applies to the coefficients used in the extrapolation (and, of course, their number). In particular, the software supporting

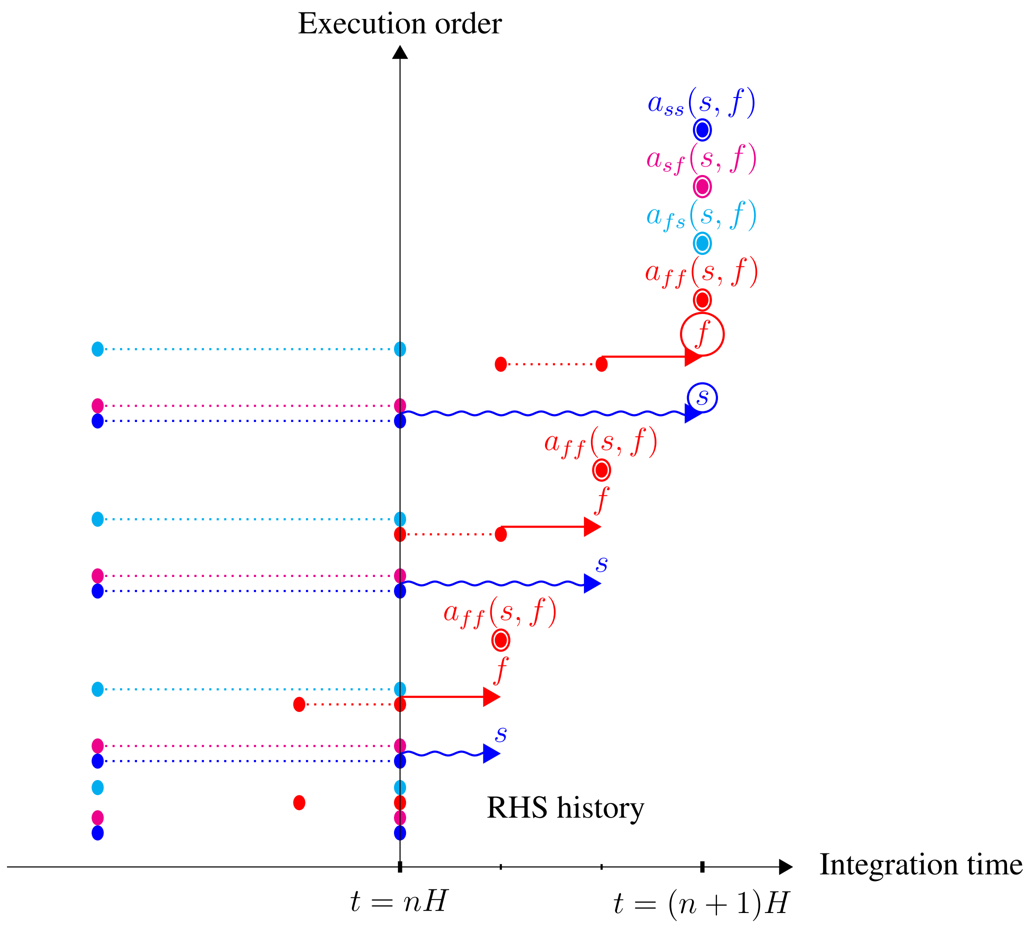


Figure 8.5. Graphical representation of the "F" multi-rate multi-step scheme, exemplifying the scheme notation introduced in Section 8.4.

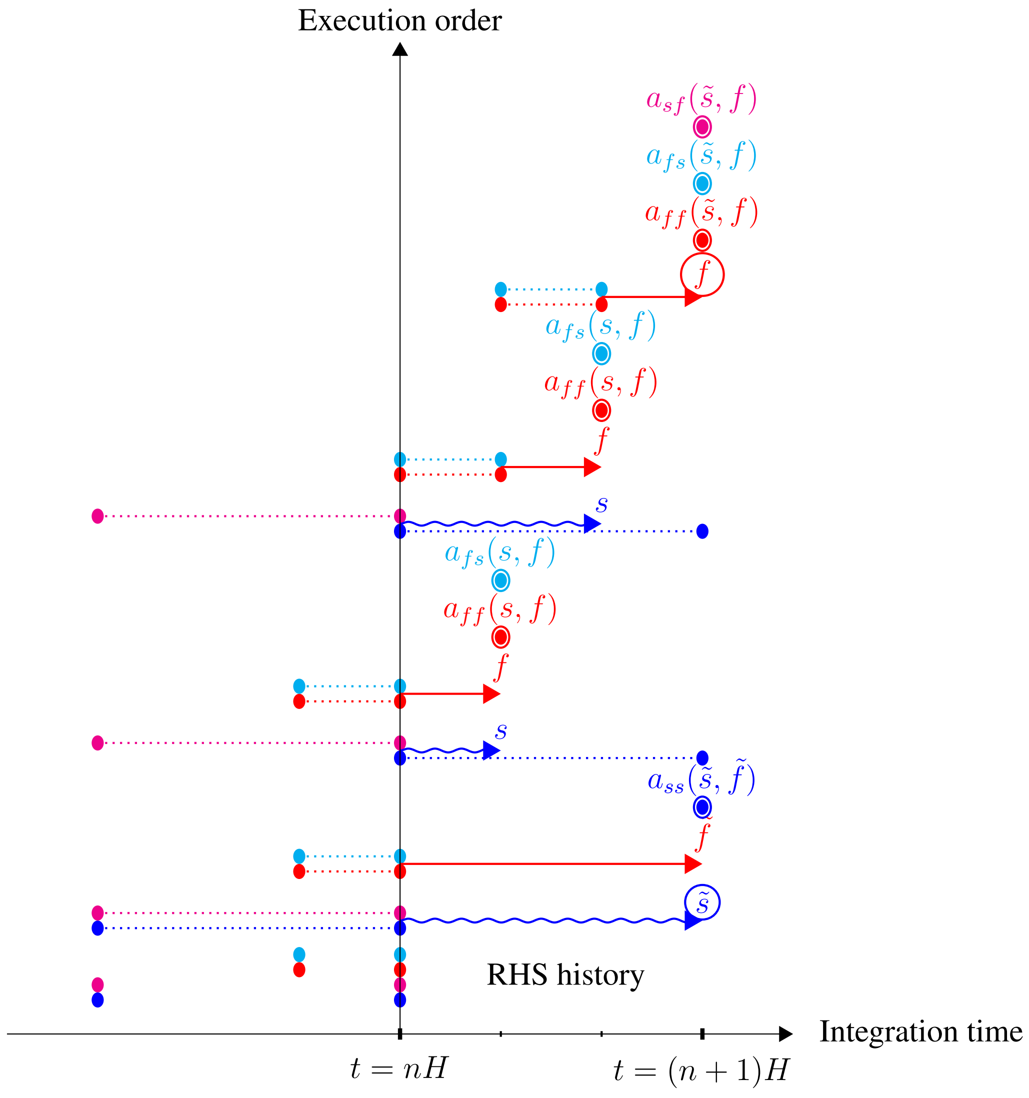


Figure 8.6. Graphical representation of the “Sq” multi-rate multi-step scheme, exemplifying the scheme notation introduced in Section 8.4.

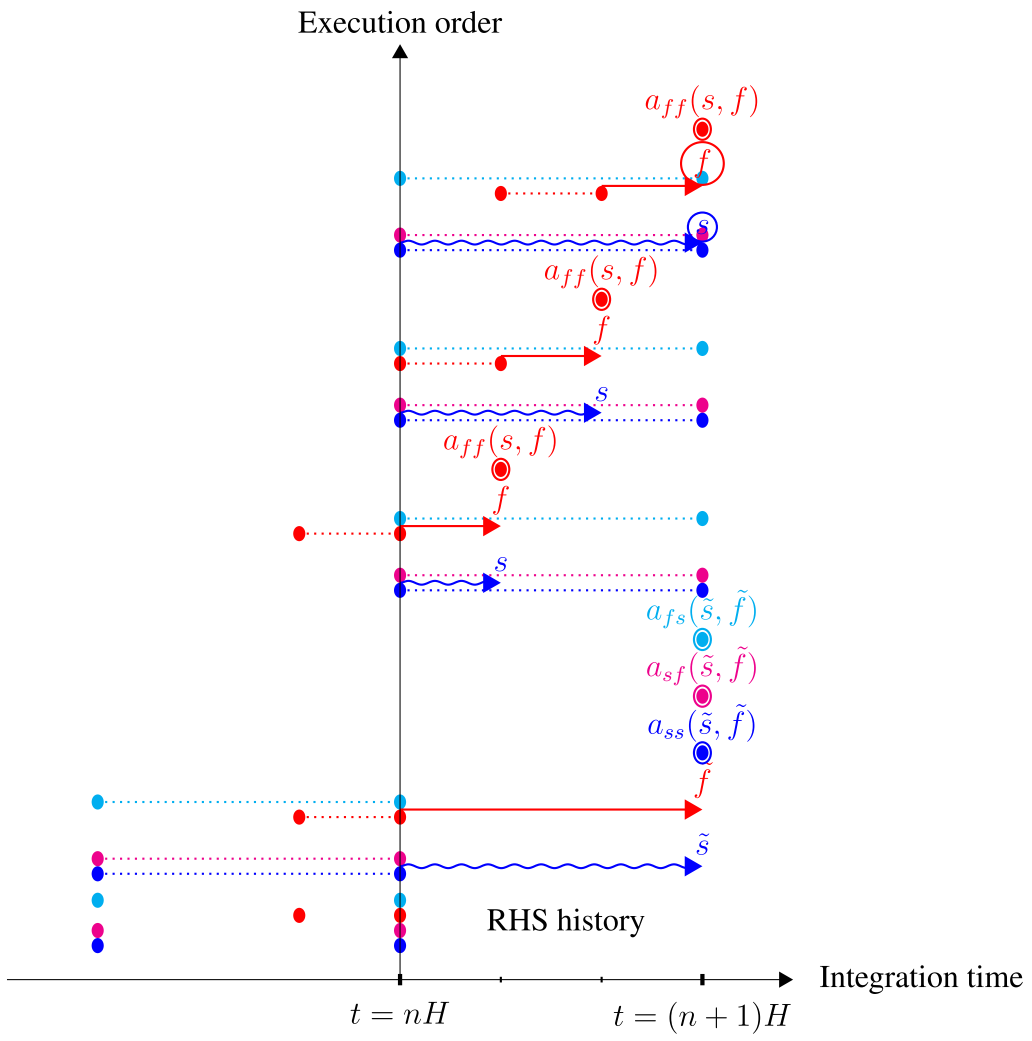


Figure 8.7. Graphical representation of the "Srsf" multi-rate multi-step scheme, exemplifying the scheme notation introduced in Section 8.4.

this chapter (part of `hedge`, see Chapter 3) allows different orders to be specified for each right-hand side.

The local truncation error for all of the methods described above decays as H^{p+1} given sufficient solution smoothness, where p is the order of the polynomial used in the extrapolation. Notably, some multi-rate multi-step methods in the literature [Sandu and Constantinescu, 2009] fail to achieve this due to their design. Gear and Wells [1984] remark that "the analysis of constant step size methods is straightforward but notationally tedious". Since they are further rather unsurprising given the construction of the method, I will move on to the more interesting, but in many senses more difficult, property of the schemes, stability, remarking that Stock [2009] shows actual convergence data for a subset of the methods presented here, and that analogous data was obtained for the remainder of methods.

While many stability results are available for multi-rate Runge-Kutta methods [e.g. Andrus, 1993, Kværnø, 2000], stability results applicable to multi-rate multi-step methods appear much less abundant.

The compelling, simple stability theory of single-rate methods and the main difficulty encountered in transferring it to multi-rate methods is aptly summarized by Gear and Wells [1984]: Consider a (potentially linearized) system $\partial_t y = Ay$. Transform A to Jordan normal form $J = SAS^{-1}$. Then a single time step of a single-rate time integrator can be rewritten as $y_{n+1} = R(HA)y$ with a single-argument (potentially rational) function R. Necessarily, if J is diagonal, then $SR(HA)S^{-1}$ will also be diagonal, and if J is triangular, then so will be $SR(HA)S^{-1}$. Further, diagonal entries of this matrix will coincide with the mapping of the eigenvalues under $R(H\cdot)$, i.e.

$$R((HJ)_{ii}) = (SR(HA)S^{-1})_{ii} \text{ for all } i.$$

Hence, aside from minor deviations in the case of non-diagonalizable A, the mapping R provides a complete description of the stability of a single-rate method, or, equivalently, nearly complete information about the stability of the system can be obtained by studying the behavior of the time integrator on the simple scalar ODE $\partial_t y = \lambda y$. However in a multi-rate method, any rational function describing the action of the method necessarily has multiple arguments, and therefore the system's diagonalizer (triangularizer) does not typically diagonalize (triangularize) the method's matrix representation. What is particularly upsetting about the failure of this line of reasoning is that it limits the hope that a small placeholder system (such as $\partial_t y = \lambda y$ above) can convey complete information on the stability of the method.

Gear and Wells [1984] comment at length on their inability to obtain anything more than a non-quantitative stability result based on a continuity argument. Some work on stability is also available for implicit first-order multi-rate schemes [Skelboe and Andersen, 1989]. Gomm [1981] obtains an explicit expression for the stability polynomials and the transfer matrix of the method applied to a 2×2 test system in terms of the z- and Laplace transforms, but has to rely on numerical evidence to discover actual information about the stable time step.

To shed a more detailed light on the method's stability, I have undertaken a new, comprehensive numerical study on the stability of MRAB methods. Like Gomm [1981], I rely on the study of a 2×2 model system. I would like to remark that I can make no statement regarding whether these results are in any way representative of the behavior that can occur for larger systems. As such, the value of this study lies in exploring the behaviors occurring in a simple case, although it is certainly my hope that, e.g. the system with eigenvalues $(\lambda_1, \lambda_2) = (i, i)$ will be a model of discretized hyperbolic PDE, or the system with eigenvalues $(\lambda_1, \lambda_2) = (i, -1)$ will model such a PDE with some dissipation.

My selection of a 2×2 model system differs from that chosen by Gomm [1981]. I am aiming to explore the limits of stability in a way that is congruent with the successful theory of stability regions for single-rate systems. Linear systems of ODEs whose solutions stay bounded in time are characterized by eigenvalues λ with $\Re\lambda \le 0$. Therefore, the major phenomena modeled by these systems are *exponential decay* and *oscillation*, corresponding to the negative real and the imaginary components of eigenvalues. I will be studying systems of the type

$$\partial_t \underbrace{\begin{pmatrix} f(t) \\ s(t) \end{pmatrix}}_{q:=} = S \underbrace{\begin{pmatrix} \lambda_1 & 0 \\ 0 & \mu\lambda_2 \end{pmatrix}}_{D:=} S^{-1} \begin{pmatrix} f(t) \\ s(t) \end{pmatrix}, \tag{8.5}$$

where I choose the eigenvalues from $(\lambda_1, \lambda_2) \in \{(-1,-1), (-1,i), (i,-1), (i,i)\}$ to capture the behavior of two coupled decay systems, combined decay-oscillation systems, and two coupled oscillating systems. I choose the rate ratio μ from $\mu \in [0.1, 1]$ to ensure that the second component s is indeed the one that evolves at a slow rate. The matrix of eigenvectors S is chosen as

$$S(\alpha, \beta) := \begin{pmatrix} \cos(\alpha) & \cos(\alpha+\beta) \\ \sin(\alpha) & \sin(\alpha+\beta) \end{pmatrix},$$

with $\alpha \in (0, \pi)$ and $\beta \in (0, \pi)$. (The reduction to $(0, \pi)$ in both cases is admissible because

$$S(\alpha,\beta)DS^{-1}(\alpha,\beta) = S(\alpha+\pi,\beta)DS^{-1}(\alpha+\pi,\beta) = S(\alpha,\beta+\pi)DS^{-1}(\alpha,\beta+\pi)$$

for all α, β.) For an enumeration of the parameter space consisting of $(\lambda_1, \lambda_2, \mu, \alpha, \beta)$, for each of the methods of Figure 8.3, and for a variety of step ratios k, a stable time step is found by bisection, where a time step size is deemed "unstable" if it reaches $\|(f,s)^T\|_2 \ge 10$ after 120 time steps from an initial condition of $(f,s) = (1,1)/\sqrt{2}$.

This amounts to determining, by means of the power method, whether the linear operator connecting the state of the multi-rate method at one time step to the state at the next time step possesses an eigenvalue of magnitude greater than one. Convergence in the power method is dependent on the ratio of the two dominant eigenvalues of that operator, and hence may be rather poor. I have experimentally determined that, after 120 time steps, qualitative behavior is captured and about two digits of the stable Δt appear to have converged. Nonetheless, it should be remembered that stability results here are approximate. All experiments are performed with AB3 schemes.

Fully evaluated, an equivalent form of (8.5) reads

$$\partial_t \begin{pmatrix} f(t) \\ s(t) \end{pmatrix} = \underbrace{\begin{pmatrix} -\frac{(\mu\lambda_2-\lambda_1)\,\sin(\beta+2\,\alpha)+(-\mu\lambda_2-\lambda_1)\,\sin\beta}{2\,\sin\beta} & \frac{(\mu\lambda_2-\lambda_1)\,\cos(\beta+2\,\alpha)+(\mu\lambda_2-\lambda_1)\,\cos\beta}{2\,\sin\beta} \\ \frac{(\mu\lambda_2-\lambda_1)\,\cos(\beta+2\,\alpha)+(\lambda_1-\mu\lambda_2)\,\cos\beta}{2\,\sin\beta} & \frac{(\mu\lambda_2-\lambda_1)\,\sin(\beta+2\,\alpha)+(\mu\lambda_2+\lambda_1)\,\sin\beta}{2\,\sin\beta} \end{pmatrix}}_{A:=} \begin{pmatrix} f(t) \\ s(t) \end{pmatrix}. \tag{8.6}$$

An alternative interpretation of (8.5) is shown in Figure 8.8(a) on page 255. It emphasizes the alignment of the eigenvectors relative to the fast/slow split directions enforced by (8.1). Depending on the eigenvalues (λ_1, λ_2), the system models exponential decay or oscillation along each of the directions of its two eigenvectors. Figure 8.8(a) also clarifies that it is sensible to examine relatively small values of α to ensure that the alignment of the fast component with the eigenvector of the large eigenvalue λ_1 is maintained.

From Figure 8.8(a), it is already clear that the angular layout of the system's eigenvectors will play a crucial role in determining the behavior of the system, and hence the stability of the method. Plots like Figure 8.8(b) are used to illustrate this behavior. For a given method,

(λ_1, λ_2)	Stable Δt
$(-1, -1)$	0.587
$(-1, i)$	0.587
$(i, -1)$	0.715
(i, i)	0.713

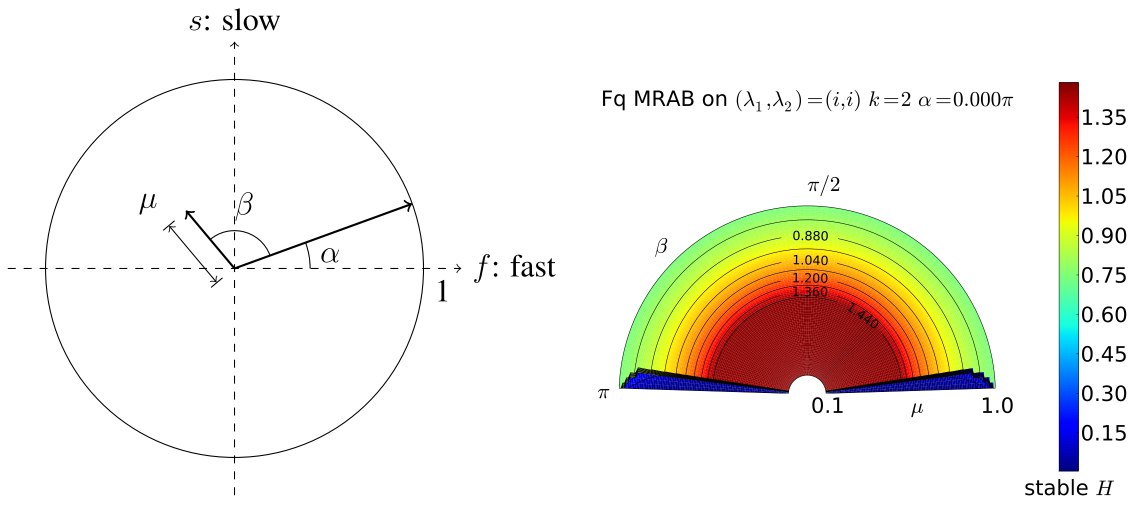
Table 8.2. Maximal stable time steps for single-rate Adams-Bashforth methods on the 2×2 test system.

sub-step ratio k, overall angle α, and eigenvalues λ_1 and λ_2, they show the effect of varying β and μ, where β is shown in the azimuthal direction, and μ is shown radially. Depending on all these parameters, the stable time step is shown, found empirically as described above. For ease of comparison with single-rate methods, Table 8.2 summarizes the stable time step of the single-rate AB3 method depending on (λ_1, λ_2), where I would like to note that this timestep is (largely) independent of α, β, and μ.

The first, simple result of my experiments was already shown by Gear and Wells [1984]: In the case of a triangular system, which is achieved for $\alpha = 0$ (cf. (8.6)), stability of the entire multi-rate method becomes equivalent to the simultaneous stability of each method for each component. Figure 8.8(b) on the following page illustrates this–the dependency on β has vanished, aside from cases where β is close to $\{0, \pi\}$, where the eigenvectors are very nearly linearly dependent and the specification of the system in the form (8.5) becomes meaningless.

Figure 8.8(b) allows one more observation. It shows stability data for a method with a sub-step ratio $k = 2$. Remarkably, about the same stability is observed for all systems whose velocity ratio μ meets or exceeds $\mu \leq 1/k$. This is good news: Even if one underestimates the velocity of the fast system, one still gets the full benefit of the multi-rate method one is applying.

It turns out that many of the system's most salient stability features appear to be found



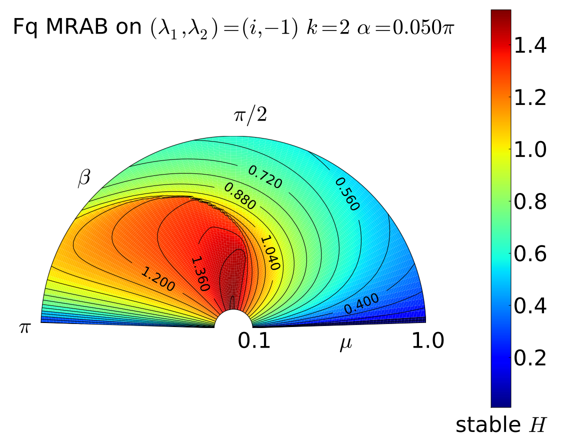
(a) Geometric layout of the eigenvectors of (8.5).

(b) Maximal stable time steps for triangular system with $\alpha = 0$.

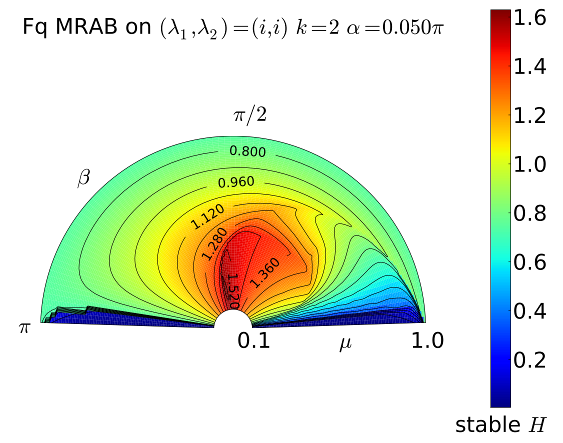
Figure 8.8. Interpretation of and first stability results on the MRAB test system (8.5).

along the direction $\beta \approx \alpha + \pi/2$. When rewritten as $\beta - \alpha = \pi/2$, this condition has a striking interpretation that can be seen from Figure 8.8(a): In this case, the eigenvector for $\mu\lambda_2$ lines up exactly with the slow axis. It is unsurprising that the method maximizes stability in this situation.

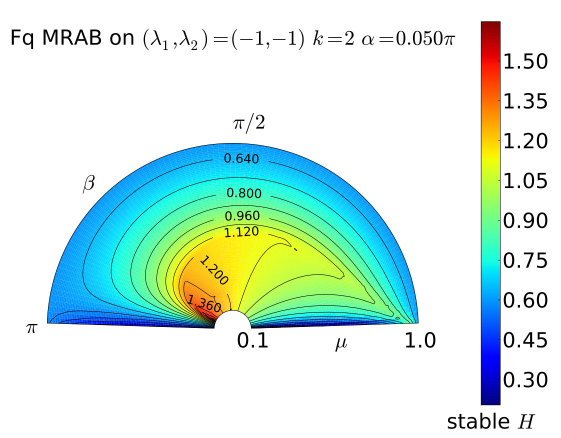
To see the practical effect of this angular relationship, I would like to direct the reader's attention to Figure 8.9(a) on the next page. The lobe of peak stability extending along $\beta = \alpha + \pi/2$ (which, due to the small value of α, is nearly aligned with the vertical axis of the page) extends up to a value of $\mu \lessapprox 1/k = 0.5$, where I note that the figure shows an experiment where $k = 2$. This confirms the earlier finding in the case of $\alpha = 0$: As long as $k \geq 1/\mu$, multi-rate methods achieve their design efficiency gain. It is again somewhat remarkable that, even for "faster" systems (i.e. smaller μ) that could be considered "mismatched" to the scheme, stability remains at least as good as in the case of $\mu = 1/k$. Further, one observes that the most stable lobe does not quite stretch out to $\mu = 1/k$–in fact, there is a rapid drop somewhat short of that point. As a result, it seems advisable to choose k slightly larger than $\lceil 1/\mu \rceil$.



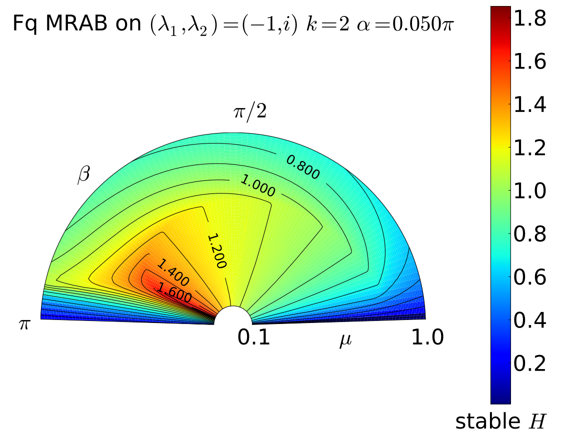
(a) Maximal stable time steps for a oscillation-decay system.



(b) Maximal stable time steps for a oscillation-oscillation system.



(c) Maximal stable time steps for a decay-decay system.



(d) Maximal stable time steps for a decay-oscillation system.

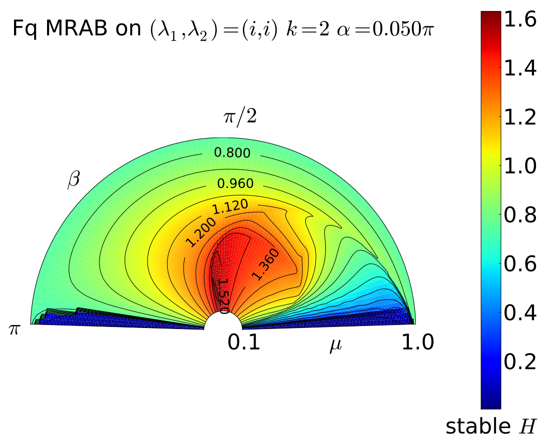
Figure 8.9. Eigenvalue dependency of the stable time step for multi-rate AB methods.

Still viewing Figure 8.9(a), it can be seen that by varying β while keeping $\mu \leq 0.5$, stability decays as the eigenvector belonging to the small eigenvalue $\mu\lambda_2$ loses alignment with the slow component of the method. This is expected. However two further observations are less so: First, even in this more complicated example, the stable time step decays monotonically (in β and μ) from its maximal value. For the most part, the worst-case time step is exhibited by the single-time-scale system represented by $\mu = 0$, at the outer edge of the diagram. In that sense, the method is free of "traps" or "surprises". Second, a lobe of reduced, but still good stability stretches along the areas of $\beta \in (\pi/2, \pi)$ that even captures regions of small μ. The origin of this lobe is unknown, though it is plausible that when μ is small, the method might prefer eigenvectors that are "more linearly dependent".

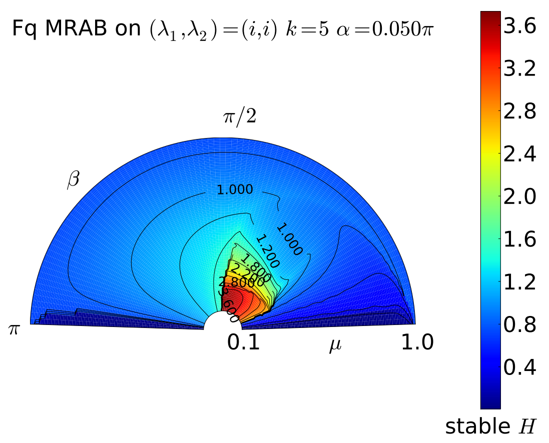
The lobe is however not robust to changes in the eigenvalues (λ_1, λ_2). In fact, while Figure 8.9(a) represented a system with $(\lambda_1, \lambda_2) = (i, -1)$, 8.9(b) on the preceding page depicts the analogous plot for the case of $(\lambda_1, \lambda_2) = (i, i)$. While conceptually (and numerically) similar to 8.9(a), the aforementioned "lobe" here stretches in the opposite direction of where it stretched before. Again, the origin of this is unknown, and until it is found, this feature should not be relied upon in practical uses of the method.

Figures 8.9(d) and 8.9(c) show stability data for the two remaining cases of $(\lambda_1, \lambda_2) = (-1, -1)$ and $(\lambda_1, \lambda_2) = (-1, i)$. It is striking how different the stability behavior of the method is in each of the four cases. Further, since only the major (imaginary and negative real) directions are captured by this experiment, it should be expected that still different phenomena are found in areas not coinciding with these directions, though one would hope (perhaps justified by the shape of single-rate AB3 stability regions) that the behavior along the axes shown here is indicative of the in-between areas.

Two further observations concerning the response of the method to changing eigenvalues is in order: First, by the (relative) similarity between Figures 8.9(b) and 8.9(a) and between



(a) Maximal stable time steps for Fq MRAB with sub-step ratio of $k = 2$. (identical to Figure 8.9(b))



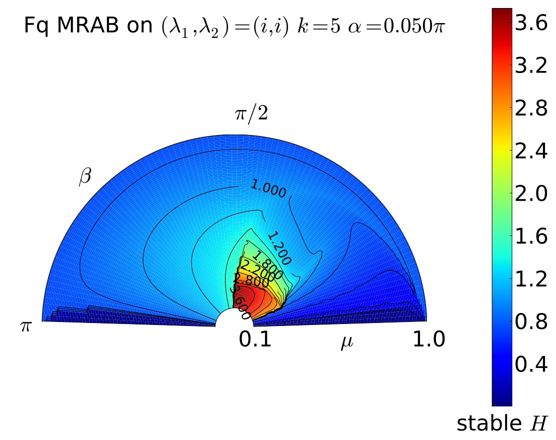
(b) Maximal stable time steps for Fq MRAB with sub-step ratio of $k = 5$.

Figure 8.10. Sub-step ratio dependency of the stable time step for multi-rate AB methods.

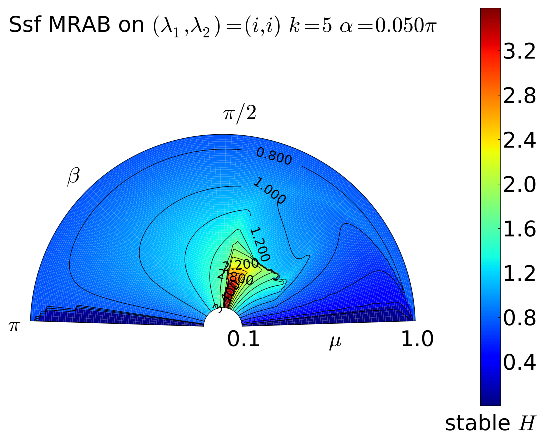
Figures 8.9(c) and 8.9(d), it appears that the dominant eigenvalue λ_1 determines stability, whereas λ_2 only changes a few details. Second, the method appears to be more stable along the imaginary axis than along the negative-real one–the "plateau" of stable H is located at around $H = 1.5$ in the oscillatory (imaginary) cases and around $H = 1.3$ in the decay (neg. real) cases. Interestingly, the same observation can be made about the single-rate results in Table 8.2.

The single-rate data of Table 8.2 allow another interesting comparison, one that determines whether the use of multi-rate methods is justified at all. Fortunately, this appears to be the case, as the stable "plateau" time steps reached by MRAB3, given by $H = 1.5$ and $H = 1.3$ for the oscillation- and decay-dominated cases above are more than 1.9 times the size of their single-rate counterparts, where the expected theoretical optimum would be a factor of two for the methods of sub-step ratio $k = 2$ examined here.

An obvious extension of this line of inquiry is whether results continue to be as good for greater sub-step ratios k. Figure 8.10 aims to shed light on this issue. The diagrams of Figure 8.10(a) and Figure 8.10(b) show stability for the same set of parameters, with one change–the sub-step ratio k has been increased from 2 to 5. Remarkably, the low



(a) Maximal stable time steps for Fq MRAB with a sub-step ratio of $k = 5$ (identical to Figure 8.10(b)).



(b) Maximal stable time steps for Ssf MRAB with a sub-step ratio of $k = 5$.

Figure 8.11. Method dependency of the stable time step for multi-rate AB methods.

contours lines for $\Delta t \in [0.7, 1.2]$ are virtually unchanged between both figures. For small $\mu \lessapprox 1/k$, again a plateau emerges, with a stable Δt of 3.6, which, at a factor of 4.86 over the single-rate Δt, confirms the scalability of MRAB methods to high sub-step ratios k. In addition, I further deem it important and fortunate that the region of high stability for $k = 5$ covers roughly the same range in β as the one for $k = 2$.

The next question concerns the robustness of the scaling and angular stability when changing from the one method (“Fq”) on which all results so far were based, to another. Figure 8.11(a) carries forward the “Fq”, $k = 5$ diagram of Figure 8.10(b) for immediate visual comparison with Figure 8.11(b), which shows stability data for the “Ssf” method on the same parameters. While almost the same stable maximal H as for “Fq” is reached, the range of β values for which that is so is much smaller. In that sense, while the behavior across methods is qualitatively similar, significant quantitative differences do occur.

Everything discussed so far is necessarily a “peephole” perspective of the behavior of MRAB methods, aimed at providing the reader with a coarse overview of the behavior to be expected. Findings in this chapter are based on a full parameter sweep, varying each of the parameters through its entire range, and I have chosen them in such a way that any

Table 8.3. Minimal, average, and maximal stable time steps Δt data across MRAB methods at sub-step ratio $k = 2$, for inner (matched) and outer (mismatched) regions.

(a) Aggregate stability data for a sub-step ratio of $k = 2$. Base data set is restricted by $\alpha = 0.05$, $\beta \in [0.3\pi, 0.7\pi]$ and $\mu \leq 0.5$.

Scheme	$\min \Delta t$	$\overline{\Delta t}$	$\max \Delta t$
Fsr	1.046	1.246	1.532
Fqsr	1.021	1.285	1.588
Ff	1.012	1.287	1.61
Ffr	1.012	1.287	1.61
Sf	1.012	1.287	1.61
Fq	0.942	1.269	1.566
Sq	0.926	1.265	1.637
Fsfr	0.922	1.215	1.58
F	0.872	1.2	1.591
S	0.872	1.2	1.591
Fsf	0.787	1.21	1.55
Ssf	0.787	1.21	1.55
Fqs	0.759	1.178	1.595
Fs	0.754	1.111	1.433
Ss	0.754	1.111	1.433
Sqs	0.737	1.172	1.588
Sqrs	0.615	1.086	1.534
Srs	0.611	1.084	1.525
Sqr	0.61	1.107	1.582
Srf	0.61	1.101	1.552
Sr	0.609	1.079	1.568
Srsf	0.525	1.016	1.509

(b) Aggregate stability data for a sub-step ratio of $k = 2$. Base data set is restricted by $\alpha = 0.05$, $\beta \in [0.3\pi, 0.7\pi]$ and $\mu \geq 0.5$.

Scheme	$\min \Delta t$	$\overline{\Delta t}$	$\max \Delta t$
Fsr	0.562	0.906	1.469
F	0.561	0.892	1.448
S	0.561	0.892	1.448
Fsfr	0.555	0.909	1.464
Ff	0.551	0.922	1.456
Ffr	0.551	0.922	1.456
Sf	0.551	0.922	1.456
Fqsr	0.55	0.92	1.47
Sq	0.545	0.922	1.454
Fq	0.545	0.926	1.453
Fsf	0.524	0.891	1.455
Ssf	0.524	0.891	1.455
Fqs	0.523	0.887	1.494
Sqs	0.514	0.873	1.46
Fs	0.514	0.858	1.42
Ss	0.514	0.858	1.42
Srs	0.319	0.563	1.012
Sqrs	0.316	0.561	0.988
Sr	0.316	0.561	0.967
Sqr	0.316	0.56	0.973
Srf	0.315	0.561	0.98
Srsf	0.313	0.573	1.291

parameter not specifically discussed above had, according to my observation, only limited influence on the result being stated. Next, I would however like to make use of the breadth of the obtained data and move on to a more global analysis of the parameter sweep results.

To do so, I will present statistics on the stable time step that aggregate data from the entire sweep, with an emphasis on providing guidance in the choice of the method. Since the method is usually the only parameter that is not prescribed by the user's problem, I anticipate that guidance in this decision might be helpful. To present a meaningful subset of the data, in all that follows, I let $\alpha = 0.05$ and $\beta \in [0.3\pi, 0.7\pi]$, in accordance with what

Table 8.4. Minimal, average, and maximal stable time steps Δt data across MRAB methods, for sub-step ratios $k = 3$ and $k = 4$.

(a) Aggregate stability data for a sub-step ratio of $k = 3$. Base data set is restricted by $\alpha = 0.05$, $\beta \in [0.3\pi, 0.7\pi]$ and $\mu \leq 0.333$.

Scheme	$\min \Delta t$	$\overline{\Delta t}$	$\max \Delta t$
Fqsr	1.421	1.872	2.406
Ff	1.28	1.797	2.266
Ffr	1.28	1.797	2.266
Sf	1.28	1.797	2.266
Fq	1.274	1.773	2.277
Sq	1.252	1.745	2.282
Fsr	1.244	1.673	2.246
F	1.134	1.557	2.188
S	1.134	1.557	2.188
Fsfr	1.048	1.571	2.227
Fsf	1.005	1.597	2.22
Ssf	1.005	1.597	2.22
Sqrs	0.908	1.562	2.293
Sqr	0.907	1.589	2.286
Srf	0.902	1.574	2.217
Fs	0.901	1.252	2.086
Ss	0.901	1.252	2.086
Sr	0.896	1.476	2.192
Srs	0.864	1.501	2.246
Fqs	0.836	1.375	2.311
Sqs	0.805	1.341	2.271
Srsf	0.695	1.3	2.21

(b) Aggregate stability data for a sub-step ratio of $k = 4$. Base data set is restricted by $\alpha = 0.05$, $\beta \in [0.3\pi, 0.7\pi]$ and $\mu \leq 0.25$.

Scheme	$\min \Delta t$	$\overline{\Delta t}$	$\max \Delta t$
Fqsr	1.553	2.409	2.998
Ff	1.304	2.221	2.95
Ffr	1.304	2.221	2.95
Sf	1.304	2.221	2.95
Fsr	1.297	2.057	2.919
Fq	1.293	2.215	2.95
Sq	1.258	2.177	2.95
Sqrs	1.222	1.978	2.937
Sqr	1.164	1.999	2.95
Srf	1.153	1.997	2.947
Fsf	1.147	1.84	2.88
Ssf	1.147	1.84	2.88
F	1.145	1.819	2.888
S	1.145	1.819	2.888
Sr	1.127	1.793	2.885
Srs	1.088	1.837	2.894
Fsfr	0.989	1.771	2.88
Fs	0.935	1.495	2.29
Ss	0.935	1.495	2.29
Srsf	0.805	1.478	2.875
Fqs	0.789	1.522	2.875
Sqs	0.748	1.436	2.944

was seen about sensible applications for MRAB. Despite the behavioral differences found between different values of (λ_1, λ_2), the statistical data in this section crudely aggregates all of these.

The data is shown in Tables 8.3 on page 260 and 8.4 on the previous page. It shows the minimal, average, and maximal stable time steps found empirically across subsections of the parameter domain. Table 8.3(a) presents such an aggreate for a sub-step ratio of $k = 2$, with the base data restricted to an appropriate subset of angles α, β, for $\mu \leq 1/k$ and $\mu \geq 1/k$. Table 8.4 presents the same type of data for sub-step ratios of $k = 3$ and $k = 4$, for $\mu \leq 1/k$.

Of the three values presented per method, the "average" one, $\overline{\Delta t}$ should be viewed with the most suspicion, as it depends on a weighting of a parameter space discretization[1]. The minimum and maximum values for Δt are of course also discretization-dependent, but less so than the average.

While crude, the data appears to admit a few conclusions. First, the data reinforces the mathematical identity of the schemes tagged as equivalent in the no-mixing case by Table 8.1. Second, the rankings are rather similar, with the exception of the slow-mismatch case $k = 2$, $\mu \geq 1/k$ of Table 8.3(b). Here, and to some extent also in the matched case $\mu \leq 1/k$ for $k = 2$ (Table 8.3(a)), methods that include the features "S" and "r" rank poorly, occupying all six bottom ranks. Closer inspection with the help of the azimuthal plots from above reveals that for single-rate ODEs with μ near 1, these methods have a considerably smaller time step than all other methods. In fact, for each of $(\lambda_1, \lambda_2) \in \{(-1, -1), (i, i)\}$ and $\mu = 1$, all methods except for the "Sr" ones exhibited the exact same stable time step. Also, the observed stability plateaus for the "Sr" family are considerably smaller than for

[1]The parameter space discretization used to generate these results is as follows: $\alpha \in [0, \pi]$, 20 equispaced points, $\beta \in (\pi/(N + 1), N\pi/(N + 1))$, $N = 20$ equispaced points, $\mu \in [0.1, 10]$, 10 equispaced points, $(\lambda_1, \lambda_2) \in \{(-1, -1), (-1, i), (i, -1), (i, i)\}$.

other methods. While this group of methods appears moderately competitive on $k \in \{3, 4\}$, these observations cast a shadow of doubt over them, making it hard to recommend them for any practical application.

The method “F”, as shown in Figure 8.5 is perhaps the most ‘natural’ of all, with a simple, short, and straightforward order of operations. It is therefore fairly likely that, given a basic understanding of MRAB, a user would pick this method. Alternatively, guided by the work by Gear and Wells [1984], a user might also try the “Ss” scheme, equivalent to their “slowest-first” method. Unfortunately, especially as k increases, both of these are not among the more stable schemes found in this investigation.

Schemes involving the feature “q” (“Run a_{fs} at fast rate”) are significantly expensive than non-“q” ones. Despite this added expense, only one “q” scheme (“Fqsr”) features prominently at the top of the ranking, especially for high k. It depends on the expense involved in evaluating the coupling term a_{fs} if the extra stability is worth the effort.

This leads to a more general point: The methods presented in this chapter vary somewhat in their expense. When evaluating stable time steps, the expense needed to reach this level of stability should also be taken into account, and factored into the ranking. However, such an expense ranking can only properly be performed in the context of an actual application, where an exact cost for each of the right-hand sides is known. Further, it appears plausible from my experiments that if an appropriate method is chosen, then near-optimal multi-rate efficiency (of nearly 90% on the simple 2×2 test problem) can be achieved. Therefore, I expect that multi-rate methods are beneficial on nearly all ODE systems with time scales separated by a factor of two or more.

In this section, it was my aim to shed light on the stability properties of the multitude of methods introduced in this chapter. To do so in an economical manner, I have restricted myself to a simple 2×2 test problem. An obvious next step would be to confirm the data gained in this section on a larger test problem.

8.6 Applications

The motivation for multi-rate time-stepping almost always comes from an application problem in which multiple time scales occur. It is the goal of this section to explore some examples of such problems, and, if available, provide results on their use with the multi-rate Adams-Bashforth methods of this chapter.

8.6.1 Domain Decomposition in the Treatment of Conservation Laws by Discontinuous Galerkin Methods

Given a hyperbolic conservation law and a mesh of a polyhedral domain Ω, discontinuous Galerkin (DG) methods (see Section 2.1), unlike most other finite element methods, are amenable to discretization in time by explicit time integrators without the use of mass lumping. This is due to the block-diagonal nature of their mass matrix, which may be inverted in an element-by-element fashion. Therefore, any explicit time stepper may be used to advance a DG solution in time, and this section seeks to explain why multi-rate steppers are a particularly promising choice.

The method's Courant-Friedrichs-Lewy condition requires a time step that scales as

$$\Delta t \sim \frac{h}{N^2},$$

where h is the local mesh size and N is the approximation's polynomial degree [Gottlieb and Tadmor, 1991, Hesthaven and Warburton, 2007]. In many applications, it is sensible to vary h and N across Ω to provide geometric or approximative resolution where it is needed, or to accommodate artifacts of poor mesh generation. Hence the time step requirement will also vary across the domain. Once mesh geometry and desired local approximation order are known, Ω may be partitioned according to the time step requirements of its constituent elements and cast in the mixed-dependency form of (8.3), with the numerical fluxes at inter-domain boundaries taking the role of the coupling terms. If the numerical flux expression in use is expressible as a sum of interior and exterior contributions, then the system may also be cast in the unmixed form (8.1). This is the case for all linear problems, in addition to many flux schemes for nonlinear problems, such as the global Lax-Friedrichs or Rusanov fluxes. It is not known which of the two splittings is preferable, if any.

Similar premises were explored by Diaz and Grote [2009], using a Leapfrog-like method, by Liu et al. [2009], using various Runge-Kutta methods, by Lörcher et al. [2008], using space-time expansions, and many others [Cohen et al., 2006, Remacle et al., 2002]. The application of MRAB to locally time-stepped DG originated with Warburton [2008], and quantitative results were published by Gödel et al. [2009b]. They report reductions of computational effort by a factor of six when comparing a multi-rate third-order Adams-Bashforth scheme to its single-rate counterpart, and a still respectable reduction by a factor of three when compared to a Runge-Kutta method achieving fourth (instead of third) order accuracy. It is remarkable that, in further work, they were able to balance processing loads in a parallel computation in such a way that much of the efficiency gain from the sequential case was retained [Gödel et al., 2009a]. The discontinuous Galerkin solver `hedge` discussed earlier in this thesis (see Chapter 3) has facilities for sequential MRAB time integration and should be able to reproduce the results by [Gödel et al., 2009b].

8.6.2 Velocity-Space Decomposition in Eulerian Vlasov-Maxwell Schemes

A slightly different situation in which multi-rate Adams-Bashforth time stepping may be helpful is encountered in the Eulerian discretization of the Vlasov-Maxwell system (see Section 7.3.1).

The Vlasov equation (7.2), as part of the larger Vlasov-Maxwell system, models variable-velocity transport along spatial and momentum axes. If the discretization of the density (which is the quantity of interest in the Vlasov equation) is can be decomposed along the momentum direction, then the resulting semi-discrete system is a profitable target for a treatment with MRAB, because in all explicit discretizations of hyperbolic conservation laws such as (7.2), the maximal possible time step is (as above) governed by the Courant-Friedrichs-Lewy condition, which implies that the time step is inversely proportional to the characteristic velocity. Quite naturally, spatial velocities in (7.2) vary quite drastically as one moves along the momentum axis. If the system can be split along the momentum direction, then high- and low-velocity parts can be separated and the whole system cast into one of the forms (8.1), (8.3), or (8.4), depending on the used discretization.

Note that it is likely that the condition that the system can be split along the momentum direction is likely to be violated. In Section 7.3.1, I have argued that it is unlikely that a non-adaptive Eulerian discretization would enjoy much success on a Vlasov-like equation. Unfortunately, the non-uniform refinement present in such a scheme can easily spoil splittability, unless the splitting is artificially upheld by purposeful placement of refinement and coarsening boundaries. On the other hand, if spectral methods (such as the ones by Narayan and Hesthaven [2009]) are used along the momentum direction and an FFT or an FFT-like transform method is used to compute the right-hand side, then not much work can be saved by the variable splitting imposed by a MRAB discretization–computing a partial right-hand side incurs nearly the same effort as computing it in its entirety.

8.6.3 Multi-Rate Time Stepping for Particle-in-Cell Methods

Yet another situation is encountered in particle-in-cell methods as discussed in Chapter 7, where the state maintained by the method is, by its very nature, partitioned into two components–the particles and the electromagnetic field.

This application of multi-rate multi-step time stepping is discussed in depth in the work by Stock [2009], however a few salient aspects are emphasized in Section 7.6.1.

8.7 Conclusions

In this chapter, I have worked out in detail the design choices and stability properties of two-rate Adams-Bashforth time integrators, as well as highlighted a number of applications where their use is beneficial. In some of these applications, more than two rates are present. For example, Gödel et al. [2009b] have used three- and four-rate schemes with good success. I have shown that a rather large array of design choices exists in two-rate methods, an even larger number of possibilities will exist in methods with more rates. For two-rate methods, if one were to choose a method largely at random, the odds of that method being "good enough" are quite high. It is not known whether the same holds for three and more rates, and thus the issue would merit closer investigation.

Further, I will be working towards finding a concise theoretic result that captures some of the behavior observed empirically in this chapter. While there are naturally many more moving parts in a multi-rate method than in a single-rate one, the ultimate goal would be to have a stability result for multi-rate schemes that is as concise and expressive as that available for single-rate ones.

CHAPTER NINE

Conclusions

Throughout the course of this thesis, I have touched upon a rather varied collection of subjects, all of which are unified by the goal of obtaining tangible improvements for users of discontinuous Galerkin methods or enabling some uses in the first place.

The main contributions made in this thesis are the following:

- **Discontinuous Galerkin schemes on graphics processors.** (joint with T. Warburton, Chapter 5) Two goals are pursued here: First, I have described a concrete set of implementation strategies for DG methods on particular hardware as well as quantified and commented on the obtained results. I have further described a parameter set along which the methods explained are expected to be adaptable to a broad set of future hardware through automated tuning.

  Second, I seek to show through this contribution that, with the emergence of mass-market massively parallel compute hardware, the set of criteria by which numerical methods should be evaluated has shifted–dealing advantages to methods that are computationally adaptable and (typically) of higher order accuracy. I view the chapter as a convincing advertisement for the point of view that practical large-scale implementation effects are neither trivial nor negligible.

- **GPU-capable viscous shock capturing** for nonlinear conservation laws. (joint with T. Warburton, Chapter 6) This contribution seeks to pick up the gauntlet thrown down by the emergence of GPUs in the context of shock-capturing methods for gas dynamics, by adopting the premise that the resulting design should be easily adaptable to such machines. Starting from work by Persson and Peraire [2006], its chief contribution is a more precise understanding of how shock detection might be performed on strictly per-element data, and how effective an artificial-viscosity based stabilization of DG methods based on such knowledge can be.

- **Particle-field couplings for high-order unstructured DG-PIC.** (Chapter 7) In my discussion of the options for the deposition and interpolation halves of a successful PIC scheme, I first and foremost hope to have aptly described the challenges faced by any method that seeks to combine high-order accuracy, geometric flexibility, and freedom from noise. To varying degrees, the methods I have proposed outperform the state of the art in a reasonably well-understood subset of cases, based on a number of quality measures I have proposed. Certainly, the results presented here are those of a work in progress more than a practical method. Nonetheless I hope that my results might inform future approaches to DG-PIC.

- **Refined classification of multi-rate Adams-Bashforth ODE solvers,** based on work by Gear and Wells [1984]. (joint with A. Stock and T. Warburton, Chapter 8) The contribution of this chapter is threefold: First, it sheds light on the ease with which multiple time scales may be captured in explicit time integrators. Second, it gives the user of such methods a certain amount of additional control through increased choice of methods. Third, and last, it strives to give an overview of observed stability behavior and help with choosing the most suitable one among the discussed multi-rate Adams-Bashforth methods. The value of the survey–in my opinion–is that it provides a stepping stone on the way to a refined study and, ultimately, better understanding of the stability of multi-rate methods.

In addition, the following contributions I have made are not quantitative but rather of a methodical or software nature:

- **Tools for metaprogramming on GPU architectures.** (Chapter 4) The key realization here is that run-time code generation, while already a known concept for decades, enjoys drastically increased benefits on massively parallel machines such as GPUs. By providing suitable tools and demonstrating their use and usefulness, this work describes a new, seamless, and simple way to combine the hybrid model proposed in

Chapter 1 with truly high-performance, massively parallel execution.

- **High-performance translators for a discontinuous Galerkin scheme language.** (Chapter 3) This project highlights a way in which computational codes can be built that preserves both abstraction and high performance. In its feature set, flexibility, and achieved performance, it is, to the best of my knowledge, novel and unique.
- **Software modules for scientific computation in Python.** (Chapter 1) One component of enabling the widest reproducibility of computational research is the free availability of the tools on which it is built. By highlighting one such tool set, proving its capabilities through my work, and releasing additional components under liberal licenses, I hope to contribute to this effort.

Needless to say, all the work I have described was heavily influenced by the insights, constant guidance, and advice of my advisor Jan Hesthaven, who deserves much of the credit for these results.

Throughout the chapters of this book, I have taken care to outline unresolved questions and directions for future research. LeVeque [2009] remarks that each piece of computational infrastructure tends to generate interesting research questions in a much larger quantity than it generates answers. In making my results, tools, and future research directions available to all, the best possible outcome and my highest hope for this work is that I might be able to inspire others to join me in pursuing answers to the set of questions laid out.

Bibliography

D. Abrahams, R. Grosse-Kunstleve, B. Goals, E. Classes, and O. Overloading. Building hybrid systems with Boost.Python. *C++ Users Journal*, 21(7):29–36, 2003.

J. F. Andrus. Numerical Solution of Systems of Ordinary Differential Equations Separated into Subsystems. *SIAM Journal on Numerical Analysis*, 16(4):605–611, August 1979. doi: 10.1137/0716045.

J. F. Andrus. Stability of a multi-rate method for numerical integration of ODE's. *Computers & Mathematics with Applications*, 25(2):3–14, 1993. doi: 10.1016/0898-1221(93)90218-K.

D. N. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini. Unified Analysis of Discontinuous Galerkin Methods for Elliptic Problems. *SIAM Journal on Numerical Analysis*, 39(5): 1749–1779, 2002. doi: 10.1137/S0036142901384162.

ASC Flash Center. Flash user's guide, version 3.2. Technical report, University of Chicago, 2009. URL `http://flash.uchicago.edu/`. Retrieved Apr 11, 2010.

U. M. Ascher, S. J. Ruuth, and B. T. R. Wetton. Implicit-explicit methods for time-dependent partial differential equations. *SIAM Journal on Numerical Analysis*, 32(3): 797–823, 1995. doi: 10.1137/0732037.

B. Bagheri and L. R. Scott. About Analysa. Technical Report 2004-09, University of Chicago Computer Science, 2004.

G. E. Barter and D. L. Darmofal. Shock capturing with PDE-based artificial viscosity for DGFEM: Part I. Formulation. *Journal of Computational Physics*, 229(5):1810 – 1827, 2010. doi: 10.1016/j.jcp.2009.11.010.

T. Barth and T. Knight. A Streaming Language Implementation of the Discontinuous Galerkin Method. Technical Report 20050184165, NASA Ames Research Center, 2005.

F. Bashforth and J. C. Adams. An attempt to test the theories of capillary action: by comparing the theoretical and measured forms of drops of fluid. With an explanation of the method of integration employed in constucting the tables which give the theoretical forms of such drops. *University Press*, 1883.

F. Bassi and S. Rebay. Accurate 2D Euler computations by means of a high order discontinuous finite element method. In *XIVth ICN MFD*, Bangalore, India, July 1994. Springer.

F. Bassi, S. Rebay, G. Mariotti, S. Pedinotti, and M. Savini. A high-order accurate discontinuous finite element method for inviscid and viscous turbomachinery flows. In R. Decuypere and G. Dibelius, editors, *2nd European Conference on Turbomachinery Fluid Dynamics and Thermodynamics*, page 99–108, Antwerpen, Belgium, March 1997. Technologisch Instituut.

N. Bell and M. Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2009. ACM.

C. K. Birdsall and A. B. Langdon. *Plasma Physics Via Computer Simulation*. McGraw-Hill, 1984. ISBN 0750310251.

P. Bogacki and L. F. Shampine. A 3(2) pair of Runge-Kutta formulas. *Applied Mathematics Letters*, 2(4):321 – 325, 1989. doi: 10.1016/0893-9659(89)90079-7.

P. Borwein and T. Erdélyi. *Polynomials and Polynomial Inequalities*. Springer, first edition, September 1995. ISBN 0387945091.

I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *Int. Conf. on Computer Graphics and Interactive Techniques*, pages 777–786. ACM New York, NY, USA, 2004.

A. Burbeau, P. Sagaut, and C. H. Bruneau. A Problem-Independent Limiter for High-Order Runge-Kutta Discontinuous Galerkin Methods. *Journal of Computational Physics*, 169 (1):111 – 150, 2001. doi: 10.1006/jcph.2001.6718.

E. Burman. On nonlinear artificial viscosity, discrete maximum principle and hyperbolic conservation laws. *BIT Numerical Mathematics*, 47(4):715–733, 2007. doi: 10.1007/s10543-007-0147-7.

M. Campos Pinto, S. Jund, S. Salmon, and E. Sonnendrücker. Charge conserving FE-PIC codes on general grids. Technical Report HAL-00311429, IRMA, University of Strasbourg, 2008.

A. Candel, A. Kabel, L. Lee, Z. Li, C. Ng, G. Schussman, K. Ko, I. Ben-Zvi, and J. Kewisch. Parallel 3D Finite Element Particle-in-Cell Simulations with PIC3P. Technical Report SLAC-PUB-13671, SLAC Linear Accelerator Center, June 2009.

M. H. Carpenter and C. A. Kennedy. Fourth-order 2N-storage Runge-Kutta schemes. Technical report, NASA Langley Research Center, 1994.

B. Catanzaro, S. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. In *PMEA '09: Programming Models for Emerging Architectures*, 2009.

A. W. Chao. *Physics of Collective Beam Instabilities in High Energy Accelerators*. Wiley-Interscience, 1993. ISBN 0471551848.

H. R. Childs, E. Brugger, K. S. Bonnell, J. S. Meredith, M. Miller, B. Whitlock, and N. Max. A Contract Based System For Large Data Visualization. In *IEEE Visualization*, page 25. IEEE Computer Society, 2005. ISBN 0-7803-9462-3. URL `http://doi.ieeecomputersociety.org/10.1109/VIS.2005.3`.

B. Cockburn and J. Guzmán. Error estimates for the Runge–Kutta discontinuous Galerkin method for the transport equation with discontinuous initial data. *SIAM Journal on Numerical Analysis*, 46(3):1364–1398, 2008. doi: 10.1137/060668936.

B. Cockburn and C. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws II: general framework. *Mathematics of Computation*, 52(186):411–435, 1989. doi: 10.2307/2008474.

B. Cockburn and C.-W. Shu. The Runge-Kutta Discontinuous Galerkin Method for Conservation Laws V: Multidimensional Systems. *Journal of Computational Physics*, 141(2): 199 – 224, 1998. doi: 10.1006/jcph.1998.5892.

B. Cockburn, S.-Y. Lin, and C.-W. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws III: One-dimensional systems. *Journal of Computational Physics*, 84(1):90 – 113, 1989. doi: 10.1016/0021-9991(89)90183-6.

B. Cockburn, S. Hou, and C.-W. Shu. The Runge-Kutta Local Projection Discontinuous Galerkin Finite Element Method for Conservation Laws. IV: The Multidimensional Case. *Mathematics of Computation*, 54(190):545–581, 1990. doi: 10.2307/2008501.

G. Cohen, X. Ferrieres, and S. Pernet. A spatial high-order hexahedral discontinuous Galerkin method to solve Maxwell's equations in time domain. *Journal of Computational Physics*, 217(2):340 – 363, 2006. doi: 10.1016/j.jcp.2006.01.004.

L. Dalcín, R. Paz, and M. Storti. MPI for Python. *J. Par. Dist. Comp.*, 65(9):1108–1115, September 2005. doi: 10.1016/j.jpdc.2005.03.010.

L. Dalcín, R. Paz, M. Storti, and J. D'Elía. MPI for Python: Performance improvements and MPI-2 extensions. *Journal of Parallel and Distributed Computing*, 68(5):655 – 662, 2008. doi: 10.1016/j.jpdc.2007.09.005.

W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonté, J. H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, and J. Gummaraju. Merrimac: Supercomputing with streams. In *Proceedings of the ACM/IEEE SC2003 Conference (SC'03)*, volume 1, 2003.

J. de Guzman. The Boost Spirit Parser Generator Framework, 2008. URL `http://spirit.sourceforge.net/`.

L. Devroye and G. Lugosi. *Combinatorial Methods in Density Estimation*. Springer, first edition, 2001. ISBN 0387951172.

J. Diaz and M. J. Grote. Energy conserving explicit local time stepping for second-order wave equations. *SIAM Journal on Scientific Computing*, 31(3):1985–2014, 2009. doi: 10.1137/070709414.

V. Dolejsí, M. Feistauer, and C. Schwab. On some aspects of the discontinuous Galerkin finite element method for conservation laws. *Mathematics and Computers in Simulation*, 61(3-6):333 – 346, 2003. doi: 10.1016/S0378-4754(02)00087-3.

J. R. Dormand and P. J. Prince. A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19 – 26, 1980. doi: 10.1016/0771-050X(80)90013-3.

M. Drouin, L. Gremillet, J.-C. Adam, and A. Héron. Particle-in-cell modeling of relativistic laser-plasma interaction with the adjustable-damping, direct implicit method. *Journal of Computational Physics*, 229(12):4781 – 4812, 2010. doi: 10.1016/j.jcp.2010.03.015.

M. Dubiner. Spectral methods on triangles and other domains. *Journal of Scientific Computing*, 6:345–390, December 1991. doi: 10.1007/BF01060030.

C. Engstler and C. Lubich. Multirate extrapolation methods for differential equations with different time scales. *Computing*, 58(2):173–185, June 1997. doi: 10.1007/BF02684438.

A. Ern, A. Stephansen, and P. Zunino. A discontinuous Galerkin method with weighted averages for advection-diffusion equations with locally small and anisotropic diffusivity. *IMA Journal of Numerical Analysis*, 29(2):235, 2009. doi: 10.1093/imanum/drm050.

T. Z. Esirkepov. Exact charge conservation scheme for particle-in-cell simulation with an arbitrary form-factor. *Computer Physics Communications*, 135(2):144 – 153, 2001. doi: 10.1016/S0010-4655(00)00228-9.

M. Feistauer and V. Kučera. On a robust discontinuous Galerkin technique for the solution of compressible flow. *Journal of Computational Physics*, 224(1):208 – 221, 2007. doi: 10.1016/j.jcp.2007.01.035.

P. Fischer, J. Lottes, D. Pointer, and A. Siegel. Petascale algorithms for reactor hydrodynamics. *Journal of Physics: Conference Series*, 125(1):012076, 2008. doi: 10.1088/1742-6596/125/1/012076.

J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws. *Journal of Parallel and Distributed Computing*, 47 (2):139 – 152, 1997. doi: 10.1006/jpdc.1997.1412.

D. Friedman and D. Wise. The Impact of Applicative Programming on Multiprocessing. In *International Conference on Parallel Processing*, pages 263–272, 1976.

M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93(2):216–231, 2005. doi: 10.1109/JPROC.2004.840301. Special issue on "Program Generation, Optimization, and Platform Adaptation".

C. W. Gear. Multirate methods for ordinary differential equations. Technical Report UIUCDCS-F–74-880, University of Illinois at Urbana-Champaign, 1974.

C. W. Gear and D. R. Wells. Multirate linear multistep methods. *BIT Numerical Mathematics*, 24(4):484–502, December 1984. doi: 10.1007/BF01934907.

C. Geuzaine and J. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre-and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009. doi: 10.1002/nme.2579.

E. Gjonaj, T. Lau, S. Schnepp, F. Wolfheimer, and T. Weiland. Accurate modelling of charged particle beams in linear accelerators. *New Journal of Physics*, 8(11):285, 2006. doi: 10.1088/1367-2630/8/11/285.

D. Göddeke, R. Strzodka, and S. Turek. Accelerating double precision FEM simulations with GPUs. In *Proceedings of ASIM*, 2005.

W. Gomm. Stability analysis of explicit multirate methods. *Mathematics and Computers in Simulation*, 23:34–50, March 1981. doi: 10.1016/0378-4754(81)90005-7.

D. F. M. Goodman and R. Brette. Brian: a simulator for spiking neural networks in Python. *Frontiers in Neuroinformatics*, 2, 2008. doi: 10.3389/neuro.11/005.2008.

D. Gottlieb and E. Tadmor. The CFL condition for spectral approximations to hyperbolic initial- boundary value problems. *Mathematics of Computation*, 56(194):565–588, 1991. doi: 10.2307/2008395.

A. Grundmann and H. M. Möller. Invariant Integration Formulas for the n-Simplex by Combinatorial Methods. *SIAM Journal on Numerical Analysis*, 15(2):282–290, 1978. doi: 10.1137/0715019.

J.-L. Guermond and R. Pasquetti. Entropy-based nonlinear viscosity for Fourier approximations of conservation laws. *Comptes Rendus Mathematique*, 346(13-14):801 – 806, 2008. doi: 10.1016/j.crma.2008.05.013.

N. A. Gumerov and R. Duraiswami. Fast multipole methods on graphics processors. *J. Comp. Phys.*, 227:8290–8313, September 2008. doi: 10.1016/j.jcp.2008.05.023.

N. Gödel, N. Nunn, T. Warburton, and M. Clemens. Accelerating Multi-GPU Based Discontinuous Galerkin FEM Computations for Electromagnetic Radio Fre-quency Problems. *ACES Journal: Special Issue on GPU Accelerated CEM Computations*, 2009a. submitted.

N. Gödel, S. Schomann, T. Warburton, and M. Clemens. Local timestepping discontinuous Galerkin methods for electromagnetic RF field problems. In *Proceedings of the Third European Conference on Antennas and Propagation (EUCAP 2009)*, pages 2149–2153, Berlin, Germany, 2009b.

M. Harris. Optimizing parallel reduction in CUDA. Technical report, Nvidia Corporation, Santa Clara, CA, 2007. URL `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`. Retrieved Apr 14, 2010.

R. Hartmann. Adaptive discontinuous Galerkin methods with shock-capturing for the compressible Navier-Stokes equations. *International Journal for Numerical Methods in Fluids*, 51(9):1131–1156, 2006. doi: 10.1002/fld.1134.

A. Haselbacher, F. Najjar, and J. Ferry. An efficient and robust particle-localization algorithm for unstructured grids. *Journal of Computational Physics*, 225(2):2198 – 2213, 2007. doi: 10.1016/j.jcp.2007.03.018.

M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952.

J. S. Hesthaven and T. Warburton. Nodal High-Order Methods on Unstructured Grids: I. Time-Domain Solution of Maxwell's Equations. *J. Comp. Phys.*, 181:186–221, September 2002. doi: 10.1006/jcph.2002.7118.

J. S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Springer, first edition, November 2007. ISBN 0387720650.

J. S. Hesthaven, S. Gottlieb, and D. Gottlieb. *Spectral Methods for Time-Dependent Problems*. Cambridge University Press, 2007. ISBN 0521792118.

D. W. Hewett. Fragmentation, merging, and internal dynamics for PIC simulation with finite size particles. *Journal of Computational Physics*, 189(2):390 – 426, 2003. doi: 10.1016/S0021-9991(03)00225-0.

R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. CRC Press, 1988. ISBN 0852743920.

J. D. Jackson. *Classical Electrodynamics*. Wiley, third edition, July 1998. ISBN 047130932X.

G. Jacobs and J. Hesthaven. Implicit-explicit time integration of a high-order particle-in-cell method with hyperbolic divergence cleaning. *Computer Physics Communications*, 180 (10):1760–1767, October 2009. doi: 10.1016/j.cpc.2009.05.020.

G. B. Jacobs and J. S. Hesthaven. High-order nodal discontinuous Galerkin particle-in-cell method on unstructured grids. *Journal of Computational Physics*, 214(1):96–121, 2006. doi: 10.1016/j.jcp.2005.09.008.

G. B. Jacobs, J. S. Hesthaven, and G. Lapenta. Simulations of the weibel instability with a High-Order discontinuous galerkin Particle-In-Cell solver. *44 th AIAA Aerospace Sciences Meeting and Exhibit*, pages 1–11, 2006.

J. Jaffre, C. Johnson, and A. Szepessy. Convergence of the discontinuous Galerkin finite element method for hyperbolic conservation laws. *Math. Models Methods Appl. Sci.*, 5 (3):367–386, 1995.

V. John and E. Schmeyer. Finite element methods for time-dependent convection–diffusion–reaction equations with small diffusion. *Computer Methods in Applied Mechanics and Engineering*, 198(3-4):475–494, 2008. doi: 10.1016/j.cma.2008.08.016.

E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL `http://www.scipy.org/`.

I. Kapchinsky and V. Vladimirsky. In *Proceedings of the Conference on High Energy Accelerators and Instrumentation*, page 274, Geneva, 1959. CERN.

G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comp.*, 20:359–392, 1999. doi: 10.1137/S1064827595287997.

D. Keppel, S. J. Eggers, and R. R. Henry. A case for runtime code generation. Technical Report UWCSE 91-11-04, University of Washington Department of Computer Science and Engineering, November 1991.

R. C. Kirby. Singularity-free evaluation of collapsed-coordinate orthogonal polynomials. *ACM Trans. Math. Softw.*, 37(1):1–16, 2010. ISSN 0098-3500. doi: 10.1145/1644001.1644006.

R. M. Kirby and S. J. Sherwin. Stabilisation of spectral/hp element methods through spectral vanishing viscosity: Application to fluid mechanics modelling. *Computer Methods in Applied Mechanics and Engineering*, 195(23-24):3128 – 3144, 2006. doi: 10.1016/j.cma.2004.09.019.

R. M. Kirby, T. C. Warburton, I. Lomtev, and G. E. Karniadakis. A discontinuous Galerkin spectral/hp method on hybrid grids. *Applied Numerical Mathematics*, 33(1-4):393 – 405, 2000. doi: 10.1016/S0168-9274(99)00106-3.

A. Klöckner. The CodePy C Code Generation Library, 2009. URL `http://mathematician.de/software/codepy`.

A. Klöckner, N. Pinto, Y. Lee, B. C. Catanzaro, P. Ivanov, and A. Fasih. Pycuda: Gpu run-time code generation for high-performance computing. Technical Report 2009-40, Scientific Computing Group, Brown University, Providence, RI, USA, November 2009a. URL `http://www.dam.brown.edu/scicomp/reports/2009-40/`. submitted.

A. Klöckner, T. Warburton, J. Bridge, and J. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *J. Comp. Phys.*, 228:7863–7882, 2009b. doi: 10.1016/j.jcp.2009.06.041.

T. Koornwinder. Two-variable analogues of the classical orthogonal polynomials. *Theory and Applications of Special Functions*, pages 435–495, 1975.

S. Krakiwsky, L. Turner, and M. Okoniewski. Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPU). In *Microwave Symposium Digest, 2004 IEEE MTT-S International*, volume 2, pages 1033–1036 Vol.2, 2004. ISBN 0149-645X. doi: 10.1109/MWSYM.2004.1339160.

L. Krivodonova. Limiters for high-order discontinuous galerkin methods. *Journal of Computational Physics*, 226(1):879–896, 2007. doi: 10.1016/j.jcp.2007.05.011.

D. Kuzmin, R. Löhner, and S. Turek. *Flux-corrected transport*. Springer, 2005.

A. Kværnø. Stability of multirate Runge-Kutta schemes. *Int. J. Differ. Equ. Appl.*, 1A(1): 97–105, 2000.

H. P. Langtangen. *Python Scripting for Computational Science*. Springer, 3rd edition, February 2009. ISBN 3540739157.

A. Lapidus. A detached shock calculation by second-order finite differences. *Journal of Computational Physics*, 2(2):154 – 177, 1967. doi: 10.1016/0021-9991(67)90032-0.

P. D. Lax. Weak solutions of nonlinear hyperbolic equations and their numerical computation. *Communications on Pure and Applied Mathematics*, 7(1):159–193, 1954. doi: 10.1002/cpa.3160070112.

S. Y. Lee. *Accelerator Physics*. World Scientific, 2004. ISBN 9812562001.

C. Lejdfors and L. Ohlsson. Implementing an embedded GPU language by combining translation and generation. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1610–1614, 2006.

C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors. *Domain-Specific Program Generation*. Number 3016 in Lecture Notes in Computer Science. Springer-Verlag, 2004.

P. Lesaint and P. Raviart. On a finite element method for solving the neutron transport equation. *Mathematical aspects of finite elements in partial differential equations*, pages 89–123, 1974.

R. J. LeVeque. Python tools for reproducible research on hyperbolic problems. *Computing in Science and Engineering*, 11:19–27, 2009. doi: 10.1109/MCSE.2009.13.

W. Li, X. Wei, and A. Kaufman. Implementing Lattice Boltzmann computation on graphics hardware. *The Visual Computer*, 19:444–456, 2003.

E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28:39–55, 2008. doi: 10.1109/MM.2008.31.

L. Liu, X. Li, and F. Hu. Nonuniform Time-step Runge-Kutta Discontinuous Galerkin Method for Computational Aeroacoustics. In *Proc. of the 15th AIAA/CEAS Aeroacoustics Conference*, Miami, FL, USA, 2009.

A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. *ACM Transactions on Mathematical Software*, 37(2), 2010. URL `http://www.dspace.cam.ac.uk/handle/1810/221918/`. To appear.

F. Lörcher, G. Gassner, and C.-D. Munz. An explicit discontinuous Galerkin scheme with local time-stepping for general unsteady diffusion equations. *J. Comp. Phys.*, 227: 5649–5670, 2008. doi: 10.1016/j.jcp.2008.02.015.

C. Mavriplis. Adaptive mesh strategies for the spectral element method. *Computer Methods in Applied Mechanics and Engineering*, 116(1-4):77 – 86, 1994. doi: 10.1016/S0045-7825(94)80010-3.

J. McCarthy. *LISP 1.5 Programmer's Manual*. MIT Press, August 1962.

M. McCool and S. Du Toit. *Metaprogramming GPUs with Sh*. A K Peters, Wellesley MA, 2004.

M. McCool and RapidMind Inc. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *Proc. GSPx Multicore Applications Conference*, 2006.

A. Medovikov. High order explicit methods for parabolic equations. *BIT Numerical Mathematics*, 38:372–390, 1998. doi: 10.1007/BF02512373.

A. H. Mohammadian, V. Shankar, and W. F. Hall. Computation of electromagnetic scattering and radiation using a time-domain finite-volume discretization procedure. *Computer Physics Communications*, 68(1-3):175 – 196, 1991. doi: 10.1016/0010-4655(91)90199-U.

C. Munz, R. Schneider, E. Sonnendrücker, and U. Voss. Maxwell's equations when the charge conservation is not satisfied. *Comptes Rendus de l'Academie des Sciences - Series I - Mathematics*, 328(5):431–436, March 1999. doi: 10.1016/S0764-4442(99)80185-2.

C. D. Munz, P. Omnes, R. Schneider, E. Sonnendrücker, and U. Voß. Divergence Correction Techniques for Maxwell Solvers Based on a Hyperbolic Model. *Journal of Computational Physics*, 161(2):484–511, July 2000. doi: 10.1006/jcph.2000.6507.

A. C. Narayan and J. S. Hesthaven. A generalization of the Wiener rational basis functions on infinite intervals. Part I - Derivation and properties. Technical Report 2009-22, Scientific Computing Group, Brown University, Providence, RI, USA, May 2009. (submitted).

A. C. Narayan and A. Klöckner. Deterministic numerical schemes for the Boltzmann equation. Technical Report 2009-39, Scientific Computing Group, Brown University, Providence, RI, USA, November 2009.

Nvidia Corporation. *NVIDIA CUDA 2.2 Compute Unified Device Architecture Programming Guide*. Nvidia Corporation, Santa Clara, USA, April 2009.

T. Oliphant. *Guide to NumPy*. Trelgol Publishing, Spanish Fork, UT, July 2006.

J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of General-Purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007. doi: 10.1111/j.1467-8659.2007.01012.x.

P. Persson and J. Peraire. Sub-Cell Shock Capturing for Discontinuous Galerkin Methods. In *Proc. of the 44th AIAA Aerospace Sciences Meeting and Exhibit*, volume 112, 2006.

N. Pinto, D. Doukhan, J. J. DiCarlo, and D. D. Cox. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PLoS Comput Biol*, 5(11):e1000579, November 2009. doi: 10.1371/journal.pcbi.1000579.

O. Pironneau, F. Hecht, A. Le Hyaric, and K. Ohtsuka. *FreeFEM++*. Laboratoire Jacques-Louis Lions, Université Pierre et Marie Curie, Paris, third edition, 2010.

J. Proft and B. Rivière. Discontinuous Galerkin Methods For Convection-Diffusion Equations For Varying And Vanishing Diffusivity. *Int. J. Num. Anal. Mod.*, 6(4):533–561, 2009.

C. Prud'homme. A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations. *Sci. Prog.*, 14(2):81–110, 2006.

P. W. Rambo. Numerical heating in hybrid plasma simulations. *Journal of Computational Physics*, 133(1):173 – 180, 1997. doi: 10.1006/jcph.1997.5678.

W. H. Reed and T. R. Hill. Triangular mesh methods for the neutron transport equation. Technical report, Los Alamos Scientific Laboratory, Los Alamos, 1973.

J. Remacle, K. Pinchedez, J. Flaherty, and M. Shephard. An efficient local time stepping-discontinuous Galerkin scheme for adaptive transient computations. *Computer Methods in Applied Mechanics and Engineering*, 2002. to appear.

J. Reynders, P. Hinker, J. Cummings, S. Atlas, S. Banerjee, W. Humphrey, K. Keahey, M. Srikant, and M. Tholburn. POOMA: A Framework for Scientific Simulation on Parallel Architectures. In G. Wilson and P. Lu, editors, *Parallel Programming using C++*. MIT Press, 1996. doi: 10.1.1.43.7428.

F. Rieper. On the dissipation mechanism of upwind-schemes in the low Mach number regime: A comparison between Roe and HLL. *Journal of Computational Physics*, 229 (2):221–232, 2010. doi: 10.1016/j.jcp.2009.09.043.

A. Ronacher. The Jinja 2 Templating Engine, 2009. URL `http://jinja.pocoo.org/2/`.

A. Sandu and E. Constantinescu. Multirate explicit Adams methods for time integration of conservation laws. *Journal of Scientific Computing*, 38(2):229–249, February 2009. doi: 10.1007/s10915-008-9235-3.

L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008. ISSN 0730-0301. doi: 10.1145/1360612.1360617.

J. Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. *Applied Computational Geometry Towards Geometric Engineering*, pages 203–222, 1996. doi: 10.1007/BFb0014474.

C. Shu. Total-variation-diminishing time discretizations. *SIAM Journal on Scientific and Statistical Computing*, 9:1073, 1988. doi: 10.1137/0909073.

C.-W. Shu and S. Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes, ii. *Journal of Computational Physics*, 83(1):32 – 78, 1989. doi: 10.1016/0021-9991(89)90222-2.

H. Si and K. Gaertner. Meshing Piecewise Linear Complexes by Constrained Delaunay Tetrahedralizations. In *Proc. of the 14th International Meshing Roundtable*, pages 147–163. Springer, 2005.

S. Skelboe and P. U. Andersen. Stability Properties of Backward Euler Multirate Formulas. *SIAM Journal on Scientific and Statistical Computing*, 10(5):1000–1009, 1989. doi: 10.1137/0910059.

J. Slater, J. Dudek, K. Tatum, et al. The NPARC Alliance Verification and Validation Archive., 2009. URL `http://www.grc.nasa.gov/WWW/wind/valid/archive.html`. Retrieved Apr 11, 2010.

G. A. Sod. A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws. *Journal of Computational Physics*, 27(1):1 – 31, 1978. doi: 10.1016/0021-9991(78)90023-2.

W. Stein and D. Joyner. Sage: System for algebra and geometry experimentation. *ACM SIGSAM Bulletin*, 39(2):61–64, 2005. doi: 10.1145/1101884.1101889.

A. Stock. Development and application of a multirate multistep AB method to a discontinuous Galerkin method based particle-in-cell scheme. Technical Report 2009-34, Scientific Computing Group, Brown University, Providence, RI, USA, October 2009.

J. M. Stone. Athena test archive, 2009. URL `http://www.astro.princeton.edu/~jstone/tests/`. Retrieved Apr 11, 2010.

J. Stratton, S. Stone, and W. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels on Multi-cores. Technical report, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, March 2008.

E. Tadmor. Convergence of spectral methods for nonlinear conservation laws. *SIAM Journal on Numerical Analysis*, 26(1):30–44, 1989. doi: 10.1137/0726003.

D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *Proceedings of the 2006 ASPLOS Conference*, volume 40, page 325–335, 2006.

The International Electrotechnical Commission. Letter symbols to be used in electrical technology - Part 2: Telecommunications and electronics. Technical report, The International Electrotechnical Commission, Geneva, Switzerland, November 2000.

The Khronos OpenCL Working Group. *The OpenCL 1.0 Specification*. Khronos Group, Beaverton, OR, December 2008.

E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*. Springer, 3rd edition, April 2009. ISBN 3540252029.

S. Tu and S. Aliabadi. A slope limiting procedure in discontinuous Galerkin finite element method for gasdynamics applications. *International Journal of Numerical Analysis and Modeling*, 2(2):163–178, 2005.

G. van Rossum et al. The Python programming language, 1994. URL `http://python.org`.

Various authors. Comparison of Nvidia graphics processing units — Wikipedia, The Free Encyclopedia, 2008. URL `http://en.wikipedia.org/w/index.php?title=Comparison_of_Nvidia_graphics_processing_units&oldid=248858931`. [Online; accessed 9-November-2008].

T. L. Veldhuizen. C++ Templates are Turing Complete. Technical report, Indiana University Computer Science, 2003.

T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

S. Venkatasubramanian. The graphics card as a stream computer. In *SIGMODDIMACS Workshop on Management and Processing of Data Streams*, 2003.

H. D. Victory and E. J. Allen. The convergence theory of Particle-in-Cell methods for multidimensional Vlasov-Poisson systems. *SIAM Journal on Numerical Analysis*, 28(5): 1207–1241, October 1991. ISSN 00361429. doi: 10.1137/0728065.

J. Villasenor and O. Buneman. Rigorous charge conservation for local electromagnetic field solvers. *Computer Physics Communications*, 69(2-3):306–316, 1992. doi: 10.1016/0010-4655(92)90169-Y.

J. von Neumann and R. Richtmyer. A method for the numerical calculation of hydrodynamic shocks. *Journal of Applied Physics*, 21:232–237, 1950. doi: 10.1063/1.1699639.

T. Warburton. An explicit construction of interpolation nodes on the simplex. *J. Eng. Math.*, 56:247–262, 2006. doi: 10.1007/s10665-006-9086-6.

T. Warburton. Accelerating the Discontinuous Galerkin Time-Domain Method. In *Proceedings of the Workshop "Non-standard Finite Element Methods"*, number 36/2008 in Oberwolfach Reports. Mathematisches Forschungsinstitut Oberwolfach, 2008.

T. Warburton and T. Hagstrom. Taming the CFL Number for Discontinuous Galerkin Methods on Structured Meshes. *SIAM J. Num. Anal.*, 46:3151–3180, 2008. doi: 10.1137/060672601.

T. Warburton, I. Lomtev, Y. Du, S. Sherwin, and G. Karniadakis. Galerkin and discontinuous Galerkin spectral/hp methods. *Computer Methods in Applied Mechanics and Engineering*, 175(3-4):343 – 359, 1999. doi: 10.1016/S0045-7825(98)00360-0.

J. Wedekind, B. Amavasai, K. Dutton, and M. Boissenin. A machine vision extension for the Ruby programming language. In *Int. Conf. on Information and Automation*, pages 991–996, 2008. doi: 10.1109/ICINFA.2008.4608143.

D. Wells. *Multirate Linear Multistep Methods for the Solution of Systems of Ordinary Differential Equations*. PhD thesis, University of Illinois at Urbana-Champaign, 1982.

R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Par. Comp.*, 27:3–35, 2001. doi: 10.1016/S0167-8191(00)00087-9.

H. Wiedemann. *Particle Accelerator Physics: Basic Principles and Linear Beam Dynamics*. Springer-Verlag, Berlin, 1993. ISBN 3540565507.

P. Woodward and P. Colella. The numerical simulation of two-dimensional fluid flow with strong shocks. *Journal of Computational Physics*, 54(1):115–173, 1984. doi: 10.1016/0021-9991(84)90142-6.

Z. Xu, Y. Liu, and C.-W. Shu. Hierarchical reconstruction for discontinuous Galerkin methods on unstructured grids with a WENO-type linear reconstruction and partial neighboring cells. *Journal of Computational Physics*, 228(6):2194 – 2212, 2009. doi: 10.1016/j.jcp.2008.11.025.

Z. Xu, J. Xu, and C.-W. Shu. A high order adaptive finite element method for solving nonlinear hyperbolic conservation laws. Technical Report 2010-14, Scientific Computing Group, Brown University, Providence, RI, USA, Apr. 2010.

K. Yee. Numerical solution of inital boundary value problems involving Maxwell's equations in isotropic media. *Antennas and Propagation, IEEE Transactions on*, 14(3): 302–307, 1966. doi: 10.1109/TAP.1966.1138693.

I. Zagorodnov and T. Weiland. TE/TM field solver for particle beam simulations without numerical Cherenkov radiation. *Phys. Rev. ST Accel. Beams*, 8(4):042001, April 2005. doi: 10.1103/PhysRevSTAB.8.042001.

Y. C. Zhou and G. W. Wei. High resolution conjugate filters for the simulation of flows. *Journal of Computational Physics*, 189(1):159 – 179, 2003. doi: 10.1016/S0021-9991(03)00206-7.

K. B. Ølgaard, A. Logg, and G. N. Wells. Automated Code Generation for Discontinuous Galerkin Methods. *SIAM Journal on Scientific Computing*, 31(2):849–864, 2008. doi: 10.1137/070710032.