A RULE-BASED SPECIFICATION SYSTEM FOR COMPUTATIONAL FLUID DYNAMICS

By

Edward Allen Luke

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computational Engineering
in the College of Engineering

Mississippi State, Mississippi

December 1999

A RULE-BASED SPECIFICATION SYSTEM FOR COMPUTATIONAL FLUID DYNAMICS

By

Edward Allen Luke

Approved:

Donna Reese Associate Professor of Computer Science (Director of Dissertation)	Pasquale Cinnella Associate Professor of Aerospace Engineering (Committee Member)
Roger Briley Professor of Mechanical Engineering (Committee Member)	Jianping Zhu Professor of Mathematics (Committee Member)
Bradley D. Carter Professor of Computer Science and Graduate Coordinator	A. Wayne Bennett Dean of the College of Engineering

Name: Edward Allen Luke

Date of Degree: December 17, 1999

Institution: Mississippi State University

Major Field: Computational Engineering

Major Professor: Dr. Donna Reese

Title of Study: A RULE-BASED SPECIFICATION SYSTEM FOR COMPUTATIONAL FLUID DYNAMICS

Pages in Study: 126

Candidate for Degree of Doctor of Philosophy

This study seeks to reduce the complexity and associated costs of developing computation fluid dynamics simulation software. A high level rule-based specification language is proposed as an approach to reducing the complexity of simulation software. The proposed specification language, using a mixture of declarative single-assignment semantics and domain specific mapping operators, provides a means of automatically assembling numerical simulation components. As a result of both the high level of specification and the automatic assembly process, much of the more mundane implementation issues involved in traditional Fortran based specifications are eliminated.

In addition to a description of the proposed specification language, this study develops numerical simulation software for compressible flows that include finite-rate chemical kinetics. This application is used as a illustration the proposed rule-based approach in the development of complex numerical software. The numerical software is validated against several test cases using a five species chemically reacting model for air including a high temperature supersonic diffuser nozzle and a Mach 10 blunt body geometry. The performance of this application is measured and found to be competitive with a representative Fortran simulation. The growth of scheduling overhead incurred when using the rule-based approach is also measured. The results of these measurements indicate that the scheduling costs will remain small even for large simulation meshes.

ACKNOWLEDGMENTS

I would like to express my appreciation to my major professor, Dr. Donna Reese, who has provided valuable moral support and friendship through the many years that we have known each other. I also would like to thank Dr. Pasqualle Cinnella. He has been an exceptional editor and technical advisor on the numerical methods for chemically reacting flows – even on my more obstinate days. I would also like to thank the rest of my committee, Dr. Jianping Zhu and Dr. Roger Briley for their contributions. In addition, I would like to thank Dr. Donald Trotter and the Engineering Research Center for providing financial support and facilities to this research effort.

TABLE OF CONTENTS

CHAPTER		Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

NOMENCLATURE

Identifiers:

$\mathbb{F}$	Fact Database
$\mathbb{R}$	Rule Database
α	A generic variable name
β	A generic mapping name
χ	Property Existential Function
τ	Iteration level name

Fluid Dynamic Variables:

$\mathcal{A}_f$	Area of a Face
$c_{v\,s}$	Species Specific Heat at Constant Volume
e_0	Fluid Total Energy Scalar
$h_{f\,s}$	Species Heat of Formation
$K_{c,r}$	Reaction Equilibrium Constant
$k_{f,r}$	Forward Reaction Rate
$k_{b,r}$	Backward Reaction Rate
k	Fluid Thermal Conductivity
I	Identity Matrix
$\tilde{\tilde{I}}$	Identity Tensor
$\mathcal{M}_s$	Species Atomic Mass
p	Fluid Pressure
p_{ref}	Reference Pressure
$\hat{R}$	Universal Gas Constant
R_s	Gas Constant for species s
s_{ref}	Entropy measured at Reference conditions
T	Fluid Temperature
T_{ref}	Reference Temperature
t	Time
$\tilde{u}$	Fluid Velocity Vector
$\tilde{u}_\Omega$	Grid Velocity Vector

$\mathcal{V}_c$	Volume of a Cell
$\dot{w}_s$	Species Production Rate
Y_s	Species Mass Fraction (ρ_s/ρ)
Ω	Volume of Space Domain
Γ	Surface of Space Domain
ρ	Fluid Density
ρ_s	Species Density
$\tilde{\tilde{\sigma}}$	Deviatoric Stress Tensor
θ_v	Vibrational Temperature
Tensor Operators:	
div	Divergence
grad	Gradient
curl	Curl
Logical Operators:	
ran(β)	The range of map β
dom(β)	The domain of map β
$\wedge$	Logical AND
$\vee$	Logical OR
$\neg$	Logical Negation
$\Rightarrow$	Logical Implication
$\rightarrow$	Mapping (composition) operator
$a \leftarrow b$	Rule operator, meaning b generates a
$\oplus$	A generic operator
$\cup$	Set Union
$\cap$	Set Intersection

CHAPTER I

INTRODUCTION

Numerical modeling of physical processes is becoming a significant component of modern engineering design. To a large part, this reality has been accelerated by the wide availability of low cost computing platforms. In fact, for a few thousand dollars it is now possible to purchase a desktop workstation that is in the same class of performance as the Cray I supercomputer of the mid-seventies. This availability of low cost computing power makes numerical modeling a highly cost-effective option for engineering analysis. In addition, the low cost of these platforms makes parallel processing, obtained by linking many of these computing platforms together, much more economically feasible than ever before. Combined with the narrowing gap in the performance of these low cost systems to the high cost supercomputer installations (particularly in terms of per processor performance), the economical modeling of phenomena that were previously considered too complex is now a possibility. To this end, the Beowulf project [1] (using a parallel machine cobbled together from off the shelf components) was able to deliver a performance in excess of one gigaflop at a cost of about fifty thousand dollars. These results only underscore the potential payoff of current technologies.

Unfortunately, although the hardware costs of putting together a system capable of supercomputing performance levels is dropping radically, the software cost required to actually utilize such resources is substantial. There are several factors which contribute to the high software costs required to utilize such systems. The most significant one is that very little progress has been made in the area of automatic parallelization of traditional sequential applications, particularly for distributed memory architectures that offer the best price-performance ratio. Instead of automated methods, tedious message passing based paradigms are currently the principal method of programming these lower cost high performance platforms. Another factor that is exacerbating this problem is that higher performance platforms often make simulating increasingly complex scenarios possible. As a result, the availability of high performance platforms pressures modeling developers to implement increasingly complex simulation software. Consequently, as the technology of numerically modeling advances, the complexity of the numerical algorithms (particularly from a software point of view) is increasing. All of these factors suggest that new technology that can address the issue of software complexity is a key component to accelerating future progress in the field of numerical modeling.

Modeling physical phenomena on a computer requires a careful validation of the reliability of these models. Problems in the validity of numerical software can occur at several levels. At the highest level, a modeler is presented with the problem of obtaining or selecting an appropriate mathematical model for the physical phenomena under investigation. This usually involves identifying the mathematical equations that accurately describe the problem. Typically, these equations are well known in each discipline. For example, the Navier-Stokes equations are known to model a broad range of physical phenomena of interest for both fluids and solids provided certain assumptions hold (for example, the continuum hypothesis). However, these mathematical models may include components that are less well understood such as closure equations for turbulence models in the case of the Navier-Stokes equations already meantoned. Once the appropriate mathematical model is chosen, it is then necessary to obtain an appropriate and accurate numerical representation of these equations in order to obtain a computational solution. Many of the fundamental problems of these steps in the modeling process are well understood, although research is still underway to refine these analyses. Analysis in validation is given the most attention: it is important to have a sound foundation to a numerical model; and to have numerical methods that are accurate and efficient. What is considered less often in validation of numerical software is that inaccurate results are often caused by software complexity itself. Thus "bugs" in numerical modeling software impact the reliability and accuracy of the numerical solution. As a result, it is possible to obtain solutions that are incorrect but reasonable. While theoretical analysis can help to obtain valid solutions when trying to select the model equations and numerical methods, a different approach is required when considering problems caused by incorrect implementations of valid numerical models.

Brooks[2] identified two sources of software complexity: essential and accidental. Under this classification, essential complexity is the complexity required to solve a problem, while accidental complexity results from unexpected requirements of implementation. Thus, while the derivation of a numerical scheme may be rather simple, an implementation in Fortran could be considerably more complex. Much of the complexity associated with the Fortran implementation has little to do with the numerical model, but instead is a result of bookkeeping work (input/output and control). This bookkeeping work is compounded in parallel implementation, where significant complexity is consumed in the management of data transfers and processors synchronization. With this accidental complexity comes opportunity for software "bugs", and increased validation costs. For example, while a simulation may provide correct results on ten processors, there could be little guarantee that it will work on eleven processors, when such a change invokes possibly different aspects of the applications communication and synchronization infrastructure.

The picture is not quite so bleak as the the discussion up to now would indicate. Complex simulation software has been developed and successfully deployed to solve real engineering problems. In fact, there is much that is known about how to solve these problems on modern architectures. For example, although partitioning workload to processors is not a completely solved problem, there has been much success in achieving excellent scaling and machine utilization for complex numerical models, involving simulations ranging from fluid flows to the evolution of black holes. Although management of data transfers and synchronization adds to the complexity of applications, this complexity has been managed through various application design approaches. The statement made here is not that these simulations cannot be accomplished in the current state-of-the-art, but rather that there is much room for improvement.

This study proposes to eliminate much of the accidental complexity encountered in numerical simulation software development by changing the way in which this software is specified. In order to successfully attack this accidental complexity, the specification must mimic the essential properties of the underlying numerical model equations. In addition, it needs to be a fundamentally concurrent specification, since there is a desire to develop software for parallel high performance architectures. This study proposes such a specification approach and argues that specification of a numerical application in this way reduces much of the accidental complexity by simply eliminating it from the implementation specification.

The development of this thesis begins with an overview of the current state of software development technologies for numerical and parallel applications (Chapter II). Chapter III presents the proposed specification language and illustrates its derivation from underlying principles of numerical algorithm development. For the specifications language described in chapter III to be meaningful to simulation software, it must be translated into a machine executable form, as discussed in the chapter IV. A numerical model for compressible flows involving finite-rate chemistry is presented in chapter V, while the implementation of this model using the proposed specification language is discussed in chapter VI. The numerical and performance results for this finite-rate chemistry model implemented within this specification approach are presented in chapter VII. Finally, chapter VIII provides a concluding summary and underscores the key findings of this work.

CHAPTER II
RELATED WORK

The recent success of parallel processing architectures in the field of numerical modeling is apparent. Unfortunately, utilization of these cost-effective parallel computing platforms also contributes additional complexity to simulation software. Much of the allocation of processor resources, management of data distribution amongst processors, and generation of communication scheduling must be provided by the simulation software, significantly increasing its complexity. Reducing the impact of this complexity on high performance applications is a topic of much research. Although there are no clear winners from the diversity of approaches attempted to date, the message passing models embodied by standards such as MPI [3] have gained significant popularity. However, this popularity is more linked to their success at delivering portable high performance than at their success at reducing complexity of high performance applications. This chapter will discuss some of the current technology that is presently being used to address some of these issues.

2.1 Parallel Programming Models

Programming models are always used in program development, whether by accident or intent. They provide the means by which program behavior is analyzed and ideas about algorithms are organized. In parallel computing, several formalized models have been developed which help application developers reason about parallel implementations. Many of these models are coupled to programming support tools and libraries that facilitate development using the constructs implicit to the programming model. Some parallel models have close correspondence to the hardware architectures that execute the parallel applications, while others have closer correspondence to the abstract requirements of concurrent algorithms.

Parallel programming models also play two important roles in the development of parallel algorithms: 1) they facilitate algorithm description by partitioning the algorithm into a set of model primitives, and 2) they facilitate estimating resource requirements such as computation time, communication cost, and memory cost. When a parallel model has associated support in the form of libraries or language constructs, then the algorithm description within that model yields straightforward implementation strategies that involve mapping the abstract primitives of the parallel model to their corresponding library or language primitives. In this sense, parallel models help address the concerns of complexity in high performance applications.

2.1.1 Shared Memory Models

A common model of parallel computations is the shared memory model. This model assumes that there is one globally accessible memory system that a collection of processors share. A common property of shared memory models is an ability to preemptively access memory utilized by another processor. Consequently, computations can proceed in each processor, provided that special precautions are taken when two or more processors must access the same memory location. There are several variants of the shared memory model, but the most commonly used approaches in high performance computations involve variations of thread and vector models.

2.1.1.1 Threads

Thread models describe computations in terms of a collection of concurrent threads of control that share a common address space. Some of the earliest developments of the thread model of computations were developed by Dijkstra[4] in the 1960s. Many of the basic ideas from that work, such as semaphores and critical sections, still apply to thread models today. More recently, standardization efforts such as POSIX threads[5] have increased the popularity of thread based approaches in high performance computing. Although the common address space is an attractive advantage of a threads paradigm (since interprocessor communication is transparent), complications due to memory corruption and synchronization concerns can make building robust thread based applications challenging. In addition, although thread based approaches have demonstrated reasonable parallel gains on small numbers of processors, it is not clear if performance gains using this approach will ever scale to massively parallel systems.

However, some recent results that combine threads with other paradigms (such as distributed memory models) have shown some success on very large scale computations.

2.1.1.2 Loop Scheduling

When the bulk of high performance computations were developed on vector computers, many of the approaches to achieving high performance computation involve identifying loops in applications that could be computed concurrently. These approaches involved the automatic identification of loops that could be safely "vectorized", while compiler directives could be used to inform the compiler of special exceptions that could not be automatically identified. Analogous support to vectorizing compilers has developed for symmetric multiprocessing servers, where compiler directives could be used to parallelize an application. Recently the OpenMP standardization effort [6] has produced a standard set of directives for this form of parallelization improving the portability of directive based approaches. The advantage of directive based approaches to parallelism is that, unlike other techniques, an application can be parallelized by incrementally adding parallelization directives, validating the results at each step. This is particularly useful when parallel implementations of applications originally developed for older vector machines are required. However, like the thread model, this approach does not appear to scale well to massively parallel systems.

2.1.2 Distributed Memory Models

Distributed memory models offer an alternative to shared memory models that is a more accurate representation of scalable computing architectures. In the distributed memory model, a parallel computer is constructed from a collection of complete computers that have their own independent memory systems. Memory on one processor cannot be directly modified by another processor except by means of communication primitives. The most common primitives of such models are send/receive pairs where one processor sends a message to another. The development of a standard applications program interface (API) for the message passing model[3][1] has helped to make message passing a popular paradigm for parallel computations. To date, message

[1] The informal standard of MPI (Message Passing Interface) was originally defined in 1994. In 1998 an updated version of MPI coined "MPI-2" was accepted. Currently, programs that conform to the original MPI specification are highly portable.

passing has provided the most scalable and portable platform for high performance application development.

The success of the message passing paradigm is largely due to the fact that it closely matches the architectures of most high performance computing systems. However, message passing does add significant complexity to high performance applications. Most of this complexity is due to the low level of control offered by message passing systems. To a large degree, message passing can be seen as a fine grained management of both processing and communication resources within a parallel system.

2.1.3 Data Parallel Models

Data parallel models are derived from the application of the same function over a collection of data elements. Since computations are closely bound to data in this model, a distribution of data to processors naturally leads to a distribution of computations to processors. The data parallel models were often put to use on SIMD (single instruction multiple data) architectures of the mid-1980's, but can also be used to take advantage of distributed memory architectures. The most common approach to data parallel models are array based calculations. These are calculations where equations manipulating large arrays of data are transformed into distributed computations by distribution of these arrays across processors. Several modern languages that support data parallel constructs are Fortran 90[7], High Performance Fortran (HPF)[8], and ZPL[9]. ZPL is based on a slightly more general model of computation that identifies Phase Abstractions as a model for parallel computations[10]. These Phase Abstractions are identified by the XYZ levels of programming, where the X level corresponds to an individual sequential processor, the Y level represents a collection of processors working on a common task in a data parallel mode, and the Z level which represents the control structure "in the large" of an application. It is argued that this model maps well to most parallel architectures, and, as a result, applications built around this model will execute with high efficiency on a wide range of architectural variations.

2.1.4 Bulk-Synchronous Protocol

The Bulk-Synchronous Protocol (BSP) has been proposed as a bridging model for parallel computations[11]. The term "bridging model" is used here to describe a model, much like the von Neumann model for sequential architectures, that facilitates development of computer architectures such that certain quantitative guarantees regarding software performance are met. Given such a model, it becomes possible for the development of software systems and hardware architectures to proceed more or less independently. In the BSP model, computations are represented by a collection of virtual processors. It is assumed that the number of virtual processors exceeds the number of physical processors, in order to provide sufficient "parallel slackness" to manage communication latency. In addition, the model assumes that there is a router that is able to perform point-to-point communications between physical processors. Finally the model assumes that there are facilities for synchronizing the processors at regular intervals, identified as super-steps. The performance of a program in this model can be estimated by four bulk parameters: the number of processors p, the speed of the processors s, the cost of a super-step l, and the cost of point-to-point communication g. Programs developed using this model can estimate the quantitative performance of their implementations based on these bulk parameters. There is currently an API for BSP based applications that has been used in the development of several high performance applications[12].

2.1.5 Dataflow Models

In dataflow models[13], a computation is described by its data dependencies. In this model a set of operations are "fired" when data for them arrives. The computational model specifies execution by way of a dependency or dataflow graph that describes the relationship between various operations. Since multiple operators in the graph may execute concurrently, the dataflow model of computation can be used to describe parallel computations. Dataflow models are supported by declarative programming languages such as Id[14] and Haskell[15]. In addition, there are high performance variations of such declarative languages such as SISAL[16], that provide a dataflow model perspective to scientific computing applications. These languages all share the common feature that they are referentially transparent: in these models, state is not explicitly represented, but rather flows through representations of operations. The lack of state is a near universal property of dataflow models. One exception to this rule is the F-Net[17] model, which expands the basic dataflow graph with state models. In this approach, the dataflow graph contains event nodes, that act like traditional dataflow operators, combined with content nodes that model variables with state. One major advantage of dataflow models is that the description of computations is independent from the allocation of work to processors. However, dataflow models have yet to demonstrate scalable performance comparable with that of message passing paradigms.

2.1.6 Linda

Linda[18] is a programming model that provides an abstraction of communication via a tuple space. In the Linda model a process may place tuples in the tuple space, while another process may extract these tuples through an associative mapping. A process may either install, read, or delete tuples from the tuple space, providing the basic communication mechanisms. The advantage of the Linda model is that partitioning of communication happens automatically; however, performance may suffer since hardware implementations do not directly support associative lookup. Although hashing implementations of this associative lookup help to address this issue, Linda still remains a high latency solution to communication in parallel programming models.

2.2 Software Libraries

One approach to reducing the complexity of parallel applications is by developing libraries that implement the most complex components of common application primitives in parallel. For high performance applications, these primitives usually either implement specific algorithms (such as solving an FFT), or provide infrastructure for some domain-specific subset of high performance applications. This section will discuss some of the more common options in this regard.

2.2.1 Linear Algebra Libraries

The most obvious numerical kernels that can be implemented for scientific computations are ones that solve systems of linear equations or provide other linear algebra support such as eigenvalue computations. There are many parallel libraries that provide kernels for linear algebra. For dense matrices that occur in many classes of scientific problems, including some solution methods for partial differential equations (e.g. potential methods), LAPACK[19] provides a library of parallel solver kernels for dense and banded matrix structures. However, the majority of partial differential equations produce highly sparse matrices, often with irregular structures. For these types of problems a library such as PetSc[20] provides a general interface to a family of Krylov subspace solvers for linear and non-linear systems of equations. The PetSc library provides a flexible object-based interface that facilitates the creation of the complex hierarchical solver architectures which are often encountered in domain decomposition methods for partial differential equations. In addition to this object-based structure, PetSc also provides some support for the development of simple mesh structures and their associated matrices. However, both LAPACK and PetSc have a linear system oriented perspective, and as a result they do not sufficiently address the complexities associated with the set up and distribution of linear equations on parallel systems. However, once these equations are described, libraries provide an effective solution methodology, and their efficiency at developing scalable high performance applications has been repeatedly demonstrated.

2.2.2 The CHAOS/PARTI Library

The CHAOS/PARTI[21] library provides a set of routines for generating communication schedules for unstructured grid computations. The basic CHAOS approach is embodied in an inspector/executor model of computations. During the inspector phase, the unstructured connectivity lists are examined to produce communication schedules of inter-processor accesses. Then, during the executor phase, computations are actually performed, calling the necessary CHAOS primitives when communication schedules need to be invoked. The CHAOS library also attempts to optimize schedules, reducing communication demands. This is done by removing redundant off-processor requests and caching multiple accesses, and by coalescing multiple off-processor communication requests.

2.2.3 The POOMA II Library

The POOMA II library[22], under development at Sandia, puts together a package of C++ classes and templates for the development of data parallel computations. In many ways, this library provides many of the features that were previously only available through specialized compilers. Through the use of expression templates developed in Blitz++[23], POOMA II is able to deliver performance comparable to a specialized compiler. However, since POOMA II is a library, it can move quickly to adapt problem-specific optimizations; moreover, POOMA II uses C++ template facilities to provide a generic data-parallel interface that can work with built in types as well as complex user-defined types.

2.2.4 Frameworks for Adaptive Mesh Refinement

Frameworks differ from libraries in the sense that instead of the application invoking a library, a framework invokes application modules. Frameworks can have an advantage for parallel processing since they can perform many optimizations and reordering of computations that are not possible with libraries. On the other hand, libraries are more likely to inter-operate than frameworks. There have been several frameworks developed for high performance computing, usually associated with particular solution methodologies. LPARX[24] and HDDA/DAGH[25] are recent examples of frameworks that have been applied to adaptive mesh refinement (AMR) algorithms. The LPARX framework provides constructions for mesh hierarchies. These constructions allow for a "region calculus" that provides straightforward ways of obtaining sets of cells in overlapping regions of meshes. From this region calculus, much of the communication scheduling is derived. Parallel formulation is not completely transparent, but is straightforward and relates sensibly to the AMR grids. HDDA/DAGH on the other hand centers around a flexible parallel data structure for storing hierarchical data. The abstraction defines a hierarchical index space and assumes a novel hashing algorithm that distributes data among processors, utilizing properties of space filling curves[26].

2.3 Domain-Specific Languages

Another approach to reducing the complexity of building scientific applications has been the development of domain-specific languages. These languages provide special constructs that facilitate more economical expression of numerical algorithms. In some cases these are general purpose languages with augmentations, and in other cases they are specialized to an extent that they can only be applied to a limited domain. In many cases, the additional semantic content in the domain-specific language can be used to provide optimizations that would be difficult to perform in the more general case.

2.3.1 FIDIL

FIDIL[27] is a functional (dataflow) language that was developed to exploit vector processors for scientific computing. The major additions to the language that support scientific computing are domain and map types. The domain represents the indices of an array, while the map represents array values. Domains are flexible, and can hold arbitrary sets of indexes. In addition, operations on domains such as intersection, union, and shifting are supported. Using domains, it is possible to represent fairly complex grid structures economically, while identifying the array structure allowed for the generation of efficient vector code. The features of domain and map types are recurring themes and appear in other specialized languages such as ZPL[9].

2.3.2 Strand

Strand[28] is a single assignment logic programming based language that is designed for the implementation of parallel applications. A strand program consists of a set of procedures defined by rules. A rule has the form of

$$H : -G_1, G_2, \cdots, G_m | B_1, B_2 \cdots, B_n \quad m, n \geq 0.$$

In this specification, the rule head, H consists of a function prototype, the G_i's represent guard tests, while the B_i's are body processes that are executed when the guard tests are satisfied. Parallelism is obtained by concurrent execution of the guards and arguments in the rule head. Communication between rules is provided by single-assignment variables. Recursive rules are used to generate streams. Mapping of processes to processors is accomplished through constructs in the language. Because of careful use of single-assignment semantics, mappings do not affect program correctness, only program performance. Thus, the developer of a Strand program can work on developing a correct implementation, and then tune the parallel mapping of processes to processors. Strand has been used for a variety of applications, including discrete event simulations, computational biology, and automated theorem proving. There are some instances of applications using strand for PDE solutions such as weather modeling, but Strand is weak in these application areas due to lack of support for distribution of array values across processors.

PCN[29] is a more recent implementation that incorporates many of the features of Strand, but with a stronger focus on parallel program composition[30]. Again, PCN uses Strand-like declarative syntax, but has a much stronger focus on providing interfaces to imperative languages like C and Fortran. The focus of PCN is in the coordination of program components to form parallel programs. Thus, much of the code of a program written using PCN may be in a traditional language, while the high-level coordination is facilitated through PCN constructs.

2.3.3 The CHAINS Model

The CHAINS[31] language utilizes an abstraction of algebraic-topological structures to describe finite element problems. Chains are simply structures that represent partial orderings (in this case of grid components). In the chains model, the basic constructs include k-cells, which are abstractions of k dimensional regions of space, and cell complexes, which define collections of various dimensional cells. Physical quantities and mathematical models are then represented on these cell complexes. Cell complexes have properties that are represented by the topological structure of the region they represent. For example, the intersection of any two cells in a cell complex is also a cell contained within the complex. Thus, if two neighboring cells are in a cell complex, then the intersection of them, or the face between them, must also be a cell in the cell complex. In addition, the CHAINS model supports various canonical ordering on cell complexes (for example the nodes of a face are represented canonically in a counter-clockwise direction).

The CHAINS language facilitates the description of a finite element problem at a very high level. The CHAINS language is problem-specific, and cannot be used to solve general problems, even problems of interest to scientific computing, such as linear system solutions or even temporal evolution. However, the CHAINS language does raise the semantic level of problem description, and does open up many possible optimizations that would be impossible otherwise, such as tailoring the linear system solvers to properties of the system (for example, symmetry).

2.3.4 Formal Specification Approach

Another interesting approach to reducing the complexity of parallel computations for numerical PDE problems has been proposed by Birken[32, 33]. In his approach, the graph is represented as a set of facts such as $nb(e1, e2)$, which indicates that element $e1$ and $e2$ are neighboring elements (elements that need to communicate). Then, a distribution of elements to processors is specified via similar facts, that is $orig(e1, p1)$ indicates element $e1$ is associated with processor $p1$. The problem of parallel distribution of work involves providing consistent copies of neighboring elements. For this, Birken identifies three specifications which must be satisfied for consistency. For example, if two given elements are neighbors and one element is on processor p, then the other element must either originate on p or it must be a copy. These three rules provide the specification of a correct communication schedule. Using these specifications and an automatic theorem prover, the space of all programs is heuristically searched to find a program that satisfies the specification. Once this program is found, it will manage communications for any topology satisfying the given specification.

The search process for this approach is a very time consuming step; however, the resulting program is proven correct, and can work for a large class of problems. Unfortunately, this approach has only been applied to simple example problems, and it is yet to be seen if the approach is workable for applications of significant complexity. Unless the heuristic search is very good, it is likely that the cost of automatic program generation could be prohibitive.

2.4 Summary

The coverage of the material in this chapter began with low level specifications that involved scheduling and coordinating concurrent processes and ended with techniques that utilized high level abstractions that remove much of the need for these lower level specifications. For example, while software libraries provide a narrow interface to specific computational kernels, the Strand language utilizes the guarded horn clause as an abstraction for the coordination of various program components. Of all of the technologies covered in this chapter, only the domain specific languages seriously address the problem of accidental software complexity that plagues current high performance numerical model implementations. The most interesting of these is the CHAINS model: it provides an interface that is very close to the process of deriving finite-element methods, although in many respects its domain of applicability is too restricted.

This study proposes a specification language approach that adopts a declarative style similar to Strand but includes semantic identifiers that specifically occur in the derivation of computational fluid dynamics models. These semantic identifiers are similar to the domain and map operators of FIDIL and ZPL, however they are better suited to an unstructured description of computations involving declarative horn clause based rules. Thus, the proposed specification language represents a domain specific approach similar to the CHAINS model except that it can describe more generic scenarios such as iteration or the components of linear system solvers.

CHAPTER III

THE DEVELOPMENT OF A SPECIFICATION LANGUAGE

This chapter will define a specification language for describing the numerical solution of partial differential and integral equations when the finite-element, finite-volume, or finite-difference discretization methods are employed. The basic ideas introduced here include a novel representation of the discretized space, boundary, and initial conditions as a collection of axiomatic statements, and the representation of the numerical solution method as a set of simple production rules. Given these sets of axiomatic statements and production rules, it is possible to deduce a computer program that will solve the given numerical problem, thus automating much of the implementation process.

3.1 A Demonstration Case

Since specification languages are not typically encountered when implementing discretization based numerical schemes, a simple example problem will be used to illustrate the basic ideas and motivations behind the rule-based approach to application development. For this purpose the one-dimensional linear diffusion problem is selected. A formal description of this problem in the interval $x \in [0,1]$ for a given diffusion constant ν, is described by the equations

$$u_t = \nu u_{xx},\ x \in (0,1), t > 0, \tag{3.1}$$

$$u(x,0) = f(x),\ x \in [0,1], \tag{3.2}$$

$$u_x(0,t) = g(t),\ \text{where } g(0) = f_x(0),\ \text{and} \tag{3.3}$$

$$u_x(1,t) = h(t),\ \text{where } h(0) = f_x(1). \tag{3.4}$$

Equations (3.1) through (3.4) formally define the problem to be solved; however, the methodology of solution is left open. A complete specification for finding an analytical solution might be stated as follows: using the Laplace transforms and associated algebraic identities, find the value of the function $u(x,t)$ such that the definitions given in equations (3.1) through (3.4) are satisfied. Notice that this specification contains three distinct parts: 1) a definition of the problem, 2) a collection of transformations, and 3) a goal that must be satisfied. For this case, an analytic solution to the problem may be found for a few specific functions $f(x)$, $g(t)$, and $h(t)$. In general, however, analytical solutions to PDE problems of interest to engineering are either impractical or impossible, due to the complexity of the geometries involved and the non-linearity of the equations themselves. For this reason, approximate numerical methods are often used to solve PDE based problems. However, the basic approach of problem and solution specification through definitions, transformations, and goals applies equally well to numerical solution methods. The question is, how does one formally specify the problem and solution methodology for these numerical methods in this definition-transformation-goal style?

3.1.1 A Finite Volume Solution

The first step in numerically approximating the function $u(x,t)$ is the discretization of the spatial domain (in this case the interval $[0,1]$). For this example, the finite volume discretization method is chosen.[1] Using this discretization approach, the interval $[0,1]$ is divided into N sub-intervals, as illustrated in figure 3.1. To facilitate describing the discretization process, the N sub-intervals, or cells, are labeled by $c = N+1, \cdots, 2N$, while the interfaces at the boundaries of sub-intervals are labeled $i = 0, \cdots, N$. Note that the typical labeling used for theoretical purposes would include half step labels for the interfaces, while a typical unstructured application code might label both cells and interfaces starting from zero and use context to distinguish between the two cases. However, for the purposes of automating reasoning about these entities of computations it is assumed that these labels are integers and that independent computational sites (in this case, cells and interfaces) are labeled distinctly. The proposed labeling satisfies both of these constraints.

[1] Other discretization schemes follow similar lines. See Appendix A for alternative discretization examples.

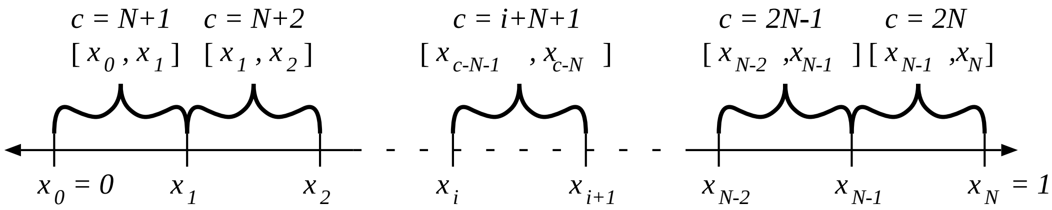


Figure 3.1: A Discretization of the Interval $[0, 1]$

As illustrated in figure 3.1, the discretization yields $N + 1$ interfaces which have the positions given by

$$x = \{(i, x_i)|i \in [0, \cdots, N], x_i = i/N\}. \tag{3.5}$$

Notice that the variable x in this equation is described by a set of ordered pairs where the first entry is the entity identifier, whereas the second entry is the value bound to that entity. This is a more general abstraction of the array. For example `x[i]` is represented abstractly as $\{x_i|(i, x_i) \in x\}$.

In addition, this discretization yields N intervals, or cells, which are represented by the mappings between cells and interfaces by way of the following relationships

$$\begin{aligned} il &= \{(c, l)|c \in [N + 1, \cdots, 2N], l = c - N - 1\}, \\ ir &= \{(c, r)|c \in [N + 1, \cdots, 2N], r = c - N\}. \end{aligned} \tag{3.6}$$

The mappings il and ir provide mappings from every cell to their left and right interfaces. The domain of ir and il is $[N + 1, \cdots, 2N]$, or the cells in the discretization, while the ranges are $\mathrm{ran}(ir) = [0, \cdots, N - 1]$ and $\mathrm{ran}(il) = [1, \cdots, N]$. This mapping is used to conveniently describe subscripts, *i.e.* $x_{c-N} = ir \to x$, where the composition operator, $\to$, defines the application of the mapping, as in

$$il \to x = \{(c, x_l)|(c, l) \in il, (l, x_l) \in x\}. \tag{3.7}$$

Using this notation, it is possible to conveniently describe cell based calculations. For example, a generic description of each cell center is given by

$$x = (ir \to x + il \to x)/2. \tag{3.8}$$

Note that the definition of x provided by equation (3.8) is only applicable to cells since only cells are in the domain of maps ir and il; however, this does not prevent the definition of x for other entities (for example, interfaces) via other rules.

The mappings il and ir are used to describe the first step of the finite volume discretization, where integration of equation (3.1) over each cell produces the equation

$$\int_{t_n}^{t_{n+1}} \int_{il \to x}^{ir \to x} u_t dx dt = \int_{t_n}^{t_{n+1}} \nu(ir \to u_x - il \to u_x) dt. \tag{3.9}$$

Equation (3.9) is an exact equation, which can be integrated numerically to obtain a numerical solution algorithm. For example, a first order end-point rule is applied to the time integrations while a second order mid-point rule is applied to the space integrations to obtain a finite-volume numerical method, expressed as

$$u^{n+1} = u^n + \nu \Delta t \left[\frac{ir \to u_x^n - il \to u_x^n}{ir \to x - il \to x} \right] \tag{3.10}$$

Equation (3.10) describes the numerical method for advancing the time step, but it is not complete. The gradient term, u_x^n, located at the interfaces has not been defined as a numerical approximation. The most straightforward approximation for u_x is a central difference formula using the values at the cell centers at either side of the interface. In order to perform this calculation it will be convenient to have a mapping from interfaces to cells similar to the development of il and ir. These mappings are defined by the relations

$$\begin{aligned} cl &= \{(i,l) | i \in [1, \cdots, N], l = i + N\}, \\ cr &= \{(i,r) | i \in [0, \cdots, N-1], r = i + N + 1\}. \end{aligned} \tag{3.11}$$

Using the definitions of cl and cr of (3.11), a numerical approximation to the gradient can be given as

$$u_x^n = \frac{cr \rightarrow u^n - cl \rightarrow u^n}{cr \rightarrow x - cl \rightarrow x}. \tag{3.12}$$

Notice that this equation uses the x-coordinate at the cell centers that is computed by equation (3.8). In addition, since this rule uses both maps cr and cl, it only defines u_x on the intersection of the domains of cr and cl, given by $[1, \cdots, N-1]$. By this reasoning, equation (3.12) only provides gradients at the internal faces of the domain. The gradient at the boundary faces is provided by the boundary conditions given in equations (3.3) and (3.4). The question is, how do these boundary conditions specify u_x at the boundaries without specifying u_x everywhere in the domain? Obviously additional information must be provided that constrains the application of boundary condition gradients only to the boundary interfaces. A solution to this problem can be found with the observation that the boundary interfaces have the distinction that either cl is defined or cr is defined, but not both. Using this fact, the rules for calculating the boundary gradients can be given by

$$u_x^n = g(n\Delta t), \mathrm{constraint}\{\neg\mathrm{dom}(cl) \wedge \mathrm{dom}(cr)\}, \tag{3.13}$$

$$u_x^n = h(n\Delta t), \mathrm{constraint}\{\mathrm{dom}(cl) \wedge \neg\mathrm{dom}(cr)\}. \tag{3.14}$$

Here the constraint term added to the rule indicates a constraint on the application of the rule. In this case it constrains the application of the boundary conditions to the appropriate boundary faces.

At this point, the computation of u^{n+1} from u^n is completely specified. However, before any such iteration can begin, an initial value, or $u^{n=0}$, must be given. To be consistent with the finite volume formulation, the derivation of the initial conditions begins with the integral form of equation (3.2), given by

$$\int_{il\rightarrow x}^{ir\rightarrow x} u^{n=0} dx = \int_{il\rightarrow x}^{ir\rightarrow x} f(x) dx. \tag{3.15}$$

Using a midpoint rule to numerically integrate this equation one obtains the rule

$$u^{n=0} = f(x), \text{constraint}\{(il, ir) \rightarrow x\}. \tag{3.16}$$

For this rule, the constraint is used to indicate that although the coordinates of the interfaces cancel in the derivation, their existence is predicated by the integration. In other words, the derivation assumed a cell perspective that includes left and right interface positions.

3.1.2 On Problem Specification

For an analytic solution method, equations (3.1) through (3.4) are sufficient to define the problem at hand. For numerical solution methods, additional definitions are required, due to the fact that these are inexact methods. For example, there are often tradeoffs between discretization and accuracy that require additional specification. In addition, since discretization for complex geometries (grid generation) is not a completely automatic process, the discretization becomes part of the problem definition for numerical solution methods. For the example diffusion problem already introduced, the definition of the numerical problem consists of spatially independent information such as the diffusion constant ν, the initial condition function $f(x)$, the numerical time step Δt, and a representation of the discretization of space. The discretization of space is given by a set of positions, (3.5), and the collection of mappings given in (3.6) and (3.11). Table 3.1 summarizes these formal definitions for the example diffusion problem.

Table 3.1: A Summary of Definitions for the Example Diffusion Problem

fact	meaning
ν	given diffusion constant
$f(x)$	given initial condition
$g(t)$	given left bc
$h(t)$	given right bc
Δt	given time-step
x	$\{(i, x_i) \| i \in [0, \cdots, N], x_i = i/N\}$
il	$\{(c, l) \| c \in [N+1, \cdots, 2N], l = c - N - 1\}$
ir	$\{(c, r) \| c \in [N+1, \cdots, 2N], r = c - N\}$
cl	$\{(i, l) \| i \in [1, \cdots, N], l = i + N\}$
cr	$\{(i, r) \| i \in [0, \cdots, N-1], r = i + N + 1\}$

3.1.3 On Specification of Process

Given the definition of the problem, the process of solving the problem is dictated by a prescribed set of transformations. For example, consider equation (3.8) as an example of a transformation that transforms x located at il and ir into a cell x. To simplify discussions of the structure of the calculations, the transformation rules are represented by a rule signature that is denoted by a list of targets of the transformation delineated from the sources of the transformation by the left arrow symbol, '$\leftarrow$'. Thus the cell center position calculation is represented by the rule signature $x \leftarrow (ir, il) \rightarrow x$. This rule signature represents the augmentation of the set of ordered pairs defined in equation (3.5) with the additional set given as

$$x \leftarrow \{(c, x_c)|x_c = (x_l + x_r)/2, (l, x_l) \in x, (r, x_r) \in x, (c, l) \in il, (c, r) \in ir\}. \tag{3.17}$$

For the moment, the augmentation of x with this set can be considered as a set union operation, with the caveat that it will become more complex once issues of specification consistency are considered. Given this notation, the specification of the finite volume scheme derived in this section can be summarized by six rules given in table 3.2.

Table 3.2: A Summary of Rules Describing the Solution of the Example Diffusion Problem.

Rule	Rule Signature	Equation
Rule 1	$x \leftarrow (ir, il) \rightarrow x$	(3.8)
Rule 2	$u^{n+1} \leftarrow u^n, (ir, il) \rightarrow (u^n_x, x)$	(3.10)
Rule 3	$u^n_x \leftarrow (cr, cl) \rightarrow (u^n, x)$	(3.12)
Rule 4	$u^n_x \leftarrow g, n, \Delta t, \text{constraint}\{\neg\text{dom}(cl) \wedge \text{dom}(cr)\}$	(3.13)
Rule 5	$u^n_x \leftarrow h, n, \Delta t, \text{constraint}\{\text{dom}(cl) \wedge \neg\text{dom}(cr)\}$	(3.14)
Rule 6	$u^{n=0} \leftarrow f, x, \text{constraint}\{(il, ir) \rightarrow x\}$	(3.16)

3.1.4 Implementing the Problem Specification

How does one translate the definition of the problem given in table 3.1 and the specification of the solution method given in table 3.2 into an implementation that can solve for $u^n, n = 0, 1, \cdots$? One accomplishes this implementation by starting from what is known and using the rules of table 3.2 to incrementally derive the specified goal. As in Prolog[34][35], rule resolution, a generalization of modus ponens, is used to produce these incremental derivations. In other words, rules are applied where their sources are satisfied. The set of entities that satisfy a rule's sources is the context of the rule. For example, Rule 1 from table 3.2 can be resolved with the definitions of il, ir, and x given in table 3.1 for the entities numbered $N+1, \cdots, 2N$. Similarly, once Rule 1 is resolved, Rule 6 can be resolved using the values of x generated by Rule 1. Iteration is recovered by way of induction. For example, if a rule generates $u^{n=0}$ while another rule generates u^{n+1}, then these two rules can be used to iteratively generate u^n for $n = 0, 1, \cdots$. Thus by resolving the rules that provide the interface and boundary gradients u_x^n a complete schedule as shown in table 3.3 can be derived from the given specification.

Table 3.3: A Deduced Execution Schedule for the Example Diffusion Problem

Rule Used	variable computed	context	comment
Rule 1	compute x_c	$c = N+1, \cdots, 2N$	cell centers
Rule 6	compute $u_c^{n=0}$	$c = N+1, \cdots, 2N$	initial conditions
Loop		define $n = 0$	for $n = 0, \cdots$
Rule 4	compute $(u_x)_i^n$	$i = 0$	left boundary condition
Rule 5	compute $(u_x)_i^n$	$i = N$	right boundary condition
Rule 3	compute $(u_x)_i^n$	$i = 1..N-1$	diffusion flux at time n
Rule 2	compute u_i^{n+1}	$i = 0..N$	advance time-step
End Loop		loop to Rule 4	replace $n = n+1$, repeat

3.1.5 Variations: Reductions

The specification of the finite volume solution method for the one-dimensional diffusion equation has been given, but this simple example has left out some important infrastructure that will require special treatment if efficient implementations are desired. The choice of a time integration used in this example is not stable for any time step, Δt, and in a typical application a stable time-step would be computed as part of the iteration calculation. For non-linear equations the stable time-step is a function of mesh spacing and dependent variables and as such can not be computed in advance. The usual approach is to compute the maximum stable time-step for each cell in the domain and then choose the smallest of these time-steps as the global time-step control. It is important to notice that the time-step is a single value that is a function of a set of entities (namely cells). In addition, this is not any ordinary function, it is a function obtained by applying an operation to some set of values. The result of such a process is always that the set of values becomes reduced to a single value by a process called reduction. Reduction processes are common in the implementation of unstructured applications. There are basically two forms of reductions that are found in these applications. One, like the time-step control discussed here, involves applying the reduction over a large set of entities to obtain a global quantity, while the other involves reducing through mappings from one set of entities to another to obtain a set of local quantities (as in summing forces from cells to nodes). The second of these operations is often used to reduce the number of connectivity lists required by the unstructured algorithm, since reductions through maps can represent functions requiring inverses of the same map. Thus using these local reductions it is possible to save computing and storing an inverse of an existing map. Since reductions are quite common in unstructured applications, some support for these cases must be provided by any specification system for unstructured numerical algorithms. The particular approach presented in this thesis will be discussed in the subsequent sections.

3.2 A Formal Specification Language

This section will discuss the specification language for unstructured numerical computations in a more precise form. The presentation will first describe the structure of the specification and what is required for a specification to be well-formed or consistent. Once the basic syntax of the specification is described, the behavior that one expects from a particular specification (its semantics) will be discussed.

3.2.1 The Database

The database is the fundamental starting point for logic programming systems. The definition of the problem to be solved begins as a collection of facts stored in a database, while the result of rule applications is the creation of new database facts. Thus the database becomes a center of communication for programs derived from the specifications. It should be noted that although the term database might be associated with files stored on disk, here the term database refers to a model of data and associated data structures. In all likelihood any efficient implementation of the specification would keep much of the database in resident memory in order to avoid large latencies in data exchange between computational kernels.

3.2.1.1 Naming Variables

Notice that the specification of variables typical in numerical approximations to partial differential equations consists of three distinct components: a name, an iteration identifier, and an entity (or spatial discretization) identifier. For example, the variable u_i^n specifies a variable named u associated with entity i at iteration n. The name of the variable becomes an abstraction for the types of computations that may be bound to it. For example, a variable named "volume" may represent volume calculations for hexahedral or tetrahedral cells. The distinction of the shape of the entity is required to complete a volume calculation. However, a calculation of density as the ratio of mass to volume can simply require a variable named "volume" without reference to any specific shape. In this sense, a density computation becomes generalized to any entity that can provide a volume. The key to utilizing this form of abstraction is the development of consistent notation. Fortunately in most engineering disciplines much of the notation is well established.

The iteration identifier of a variable represents an iteration hierarchy. This hierarchy builds on top of stationary (iteration invariant) variables, where each iteration level is represented by an iterator name (for example, n in u_i^n). Nested levels of iteration are represented by a list of iterator names, as illustrated in figure 3.2. Note that repeated iterator names are not allowed, thus n, it is a valid iteration identifier while n, it, n is not. In addition, the iteration identifier may identify previous iteration values by the use of an offset (as in $n, it-1$). Since superscripting is difficult to express in program input, brace delimited iteration identifiers will be used to signify the iteration identifier of a variable; thus u^{n-1} becomes $u\{n-1\}$.

3.2.1.2 Defining Data

Definition 3.1 An Iteration Identifier *is a possibly empty list of non-repeating iteration variables given by $\tau = I_1, I_2, \cdots, I_n$. If the list of iteration variables is empty, then the iteration identifier is stationary, referred to as $\tau_{stationary}$, or independent of any iteration. Iteration identifiers form a partial order, such that if $\tau^1 = I_1^1, \cdots, I_N^1$ and $\tau^2 = I_1^2, \cdots, I_M^2$ then $\tau^1 \leq \tau^2$ if and only if $N \leq M$ and $(I_i^1 = I_i^2) \forall I_i^1 \in \tau^1$. Similarly, the relation $\tau^1 < \tau^2$ is defined as $\tau^1 \leq \tau^2 \wedge \neg(\tau^2 \leq \tau^1)$. This partial order, and its relationship to iteration, is illustrated in figure 3.2.*

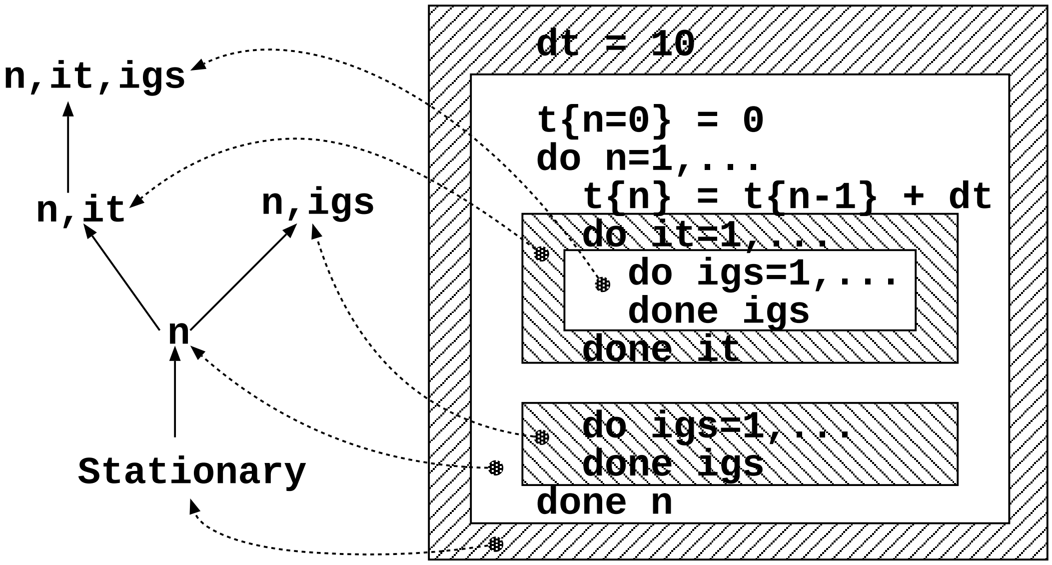


Figure 3.2: Capturing Nested Iterations as a Hierarchy.

Definition 3.2 A set of iteration identifiers, $\mathcal{T}$, is compatible *if for any $\tau^1, \tau^2 \in \mathcal{T}$, $\tau^1 \leq \tau^2 \vee \tau^2 \leq \tau^1$ holds. Note: for any given set of compatible iteration identifiers, $\mathcal{T}$, there exists a greatest lower bound, $\tau_{glb} \in \mathcal{T}$ and a least upper bound $\tau_{lub} \in \mathcal{T}$.*

Example 3.1 Given the iteration hierarchy illustrated in figure 3.2, the set of iteration identifiers $\mathcal{T}_1 = \{(n), (n, it), (n, igs)\}$ is incompatible. However, the set given by $\mathcal{T}_2 = \{(n), (n, it), (n, it, igs)\}$ is compatible with a greatest lower bound of $\tau_{glb} = n$, and a least upper bound of $\tau_{lub} = n, it, igs$.

Definition 3.3 A Variable Identifier *consists of a variable name, α, iteration identifier, τ, and offset, $\theta \leq 1$.*[2] *The variable identifier is represented by the notation $\alpha\{\tau + \theta\}$ for relative offsets and $\alpha\{\tau = \theta\}$ for absolute offsets.*

The fact database provides an association of variable identifiers to facts about entities. In general, these facts consist of associations of entities with either values or other entities. As such, any variable in the fact database has a domain, the set of entities described by the variable, and a range, consisting of a set of either entities or values.

---

[2] *This is consistent with offset of $\theta = 0$ representing the present iteration values and an offset of $\theta = 1$ representing the next iteration values. All iteration offsets for $\theta < 0$ represent the history of computations.*

Definition 3.4 *A* store *defines a bijective mapping from entities to values. Thus for a variable identified as α a store is defined by the set $\alpha = \{(i, v_i)|i \in dom(\alpha)\}$. A store is basically a construct for providing entity labels to values.*

Definition 3.5 *A* map *defines relationships between entities. It is represented by a set of ordered pairs such that the map β is represented by the set $\beta = \{(i,j)|i \in dom(\beta), j \in ran(\beta)\}$. The inverse of map β is given by $\beta^{-1} = \{(j,i)|(i,j) \in \beta\}$.*

3.2.2 Transformation Rules

Definition 3.6 *The* mapping operator, *denoted as $\rightarrow$, is defined by the set $\beta \rightarrow \alpha = \{(i, \alpha_j)|(i,j) \in \beta, (j, \alpha_j) \in \alpha\}$. Mapping operators can be chained such as $\beta_1 \rightarrow \beta_2 \rightarrow \cdots \rightarrow \alpha$.*

Definition 3.7 *A* rule signature *is defined by a head and body written as "Head $\leftarrow$ Body". The head consists of a variable accessor, while the body consists of a list of variable accessors and rule qualifiers (the various rule qualifiers will be defined later). Variable accessors are represented as $\beta_1 \rightarrow \beta_2 \rightarrow \cdots \rightarrow \beta_N \rightarrow \alpha$ where $N \geq 0$, β_i are variable identifiers of maps, and α is a variable identifier for the accessed store.*

Example 3.2 For ideal gases, the relationship between pressure, temperature, and density is given by the equation $p = \rho\tilde{R}T$ where p is pressure, ρ is density, $\tilde{R}$ is the mixture gas constant, and T is the temperature. This equation is represented by the rule signature "$p \leftarrow \rho, \tilde{R}, T$". This rule implicitly applies to any entity for which ρ, $\tilde{R}$, and T is defined.

Definition 3.8 *A rule signature is* compatible *when all of the variable identifiers represented in the head and the body have compatible iteration identifiers. A compatible rule signature has a head time, represented by τ_h, which is the least upper bound of all iteration identifiers in the head, and a body time, represented by τ_b, which is the least upper bound of all iteration identifiers listed in the body.*

3.2.2.1 Iteration Specification Rules

Iteration is defined by way of three types of rule specifications: build rules that construct the iteration, advance rules that advance the iteration, and collapse rules that terminate the iteration. This specification follows an analogy to the inductive proof in that build rules are analogous to an inductive base while advance rules are analogous to an inductive hypothesis.

For example, to describe an iteration where a variable named `q` is iterated to a converged solution may be described by the following three rules. A build rule of the form `q{n=0}<-ic`, while an advance rule might look something like `q{n+1}<-q{n},dq{n}`, and finally the iteration of `q` might be terminated by the collapse rule `solution<-q{n},CONDITION(converged{n})`.

Definition 3.9 A build rule *is a rule where $\tau_b < \tau_h$. A properly specified build rule must have an absolute offset, $\theta \leq 0$,*[3] *to the iteration identifier in the head, for example $\alpha\{\tau = 0\}$.*

Example 3.3 A build rule that initializes an iteration on variable "u" from the stationary initial condition information named "ic" would be of the form "$u\{n = 0\} \leftarrow ic$". Note that the build rule always has an absolute offset in the head variable identifier. For example "$u\{n - 1\} \leftarrow ic$" is an ill-formed rule specification.

Definition 3.10 An advance rule *is a rule where the offset to the time identifier in the head of the rule has a value of one. An advance rule is properly specified when $\tau_h \leq \tau_b$.*

Definition 3.11 A collapse rule *is a rule where $\tau_h < \tau_b$. A collapse rule is always qualified by a conditional qualifier of the form $CONDITIONAL(\alpha)$ where α is a variable identifier that identifies the loop termination condition. It must have a time identifier that is equal to τ_b. Note, a collapse rule for iteration level τ_b may also be an advance rule for iteration τ_h.*

Example 3.4 A collapse rule that terminates an iteration over the iteration identifier $\tau = n$ can be given by the rule "$solution \leftarrow u\{n\}, CONDITIONAL(converged)$".

3.2.2.2 Variations on a Theme: Constraints

Consider the implementation of an impermeable surface boundary condition. This boundary condition might be specified by marking a set of faces for which the impermeable boundary condition will supply additional information. How do the rules that satisfy the impermeable boundary condition only get applied to the entities that require them? In this case it is necessary

---

[3] *This assumes that all iterations will begin at $\theta = 0$. Thus, all offsets $\theta < 0$ represent the initial history given to the iteration.*

to constrain a rule so that it can only apply to a subset of entities. This is accomplished by defining constraints in the database.

Definition 3.12 A constraint *defines an identity mapping of entities. Thus constraint β is defined by $\beta = \{(i,i)|i \in \mathcal{C}\}$ where $\mathcal{C}$ is the set of entities in the constraint.*

Example 3.5 Using a constraint it is possible to apply a rule to a limited set of entities. The constraint is an identity mapping that is only defined for entities in the constraint, thus it is straightforward to use this information to constrain a calculation to this subset of entities by simply directing the calculations through this map. For example, an application of the Dirichlet boundary condition might be implemented using a constraint named "Dirichlet" by the rule "$dirichlet \rightarrow u_x \leftarrow left \rightarrow u$". Here the constraint is used to limit the application of the rule only to boundary faces.

Definition 3.13 A constraint qualifier *is of the form $CONSTRAINT(clist)$ where $clist$ consists of a comma separated list of variable accessors. The constraint set is formed from the intersection of the domains of these variable accessors.*

3.2.2.3 Variations on a Theme: Parameters

Consider the specification of the diffusion constant in the example at the beginning of this chapter. By definition, this diffusion constant is a single value that is accessible to all entities in the simulation. For this we need a store with a special guarantee, that for all entities it defines the same value, or a singleton guarantee. For this, the parameter is defined that provides these singleton semantics.

Definition 3.14 A parameter *defines a singleton, or mapping from a set of entities to a single value. Thus, for a variable identified as α, a parameter mapping is identified by the set $\alpha = \{(i,v)|i \in dom(\alpha)\}$.*

3.2.2.4 Variations on a Theme: Reductions

In the numerical diffusion equation example given at the beginning of this chapter, a time-step was supplied by the user. As a matter of practice, explicit (and sometimes implicit) formulations of numerical algorithms become unstable if the time-step exceeds a certain stability limit. In production applications, the time-step will be determined by a computation of the maximum time-step that would not exceed a stability limit for any computational entity. Since the stability limit may depend on the solution variables themselves, the time-step may become a function of the entire set of solution variables. In a parallel environment, functions that compute a single value from a distributed variable set can be a source of scalability problems. For example, if all of the variables are sent to a single processor to evaluate this function, a performance bottle-neck would develop. Instead, these functions are evaluated by way of a reduction operation.

A reduction operation can be defined by any function that can be decomposed according the the Bird-Meertens formalism[36]. This formalism identifies a class of functions that can be decomposed into two distinct components: a function that is applied to each element of a container, and an associative binary operator that has an associated identity element. The advantage of identifying this structure is in the parallel execution of these "global" functions.

The Bird-Meertens formalism provides a means of describing what are otherwise known as reduction operations. In this abstraction, a reduction is described by a function composed of three components: a function that is applied to a set of values, an associative, commutative operator $\oplus$ that is defined on the type returned by the above mentioned function, and and identity element of operator $\oplus$, e. Thus a reduction, r, over values, $\{v_i | i \in [1, N]\}$, using function f and operator $\oplus$ is defined as

$$r = f(v_1) \oplus f(v_2) \oplus \cdots \oplus f(v_i) \oplus \cdots \oplus f(v_N). \tag{3.18}$$

When the reduction is evaluated using a left or right precedence rule, then a sequential evaluation is derived; however, the associative property of $\oplus$ allows for different parallel evaluation orders. For example, the set of values can be partitioned into subsets that can be evaluated concurrently as in

$$r = \{e \oplus f(v_1) \oplus f(v_2) \cdots \oplus f(v_p)\} \oplus \{e \oplus f(v_{p+1}) \oplus \cdots \oplus f(v_N)\}, \tag{3.19}$$

where the identity element is used to initialize each subset. Parallel partitioning of reduction operations can be expressed when given three basic computational methods identified as unit, apply, and join, as listed in table 3.4. The unit rule initializes a reduction variable to the identity element, the apply rule "accumulates" results of a function application to a set of values and the join operation "accumulates" two partial results. The algorithm for partitioning this reduction operation among parallel processors is accomplished by creating a copy of the reduction variable on each processor participating in the reduction. Each reduction variable is initialized to the identity, and then followed by the application of all apply rules that are in that processor's partition. Finally the partial results for each processor are reduced to the final result using a join operation.

Table 3.4: Basic Reduction Steps

Rule Type	Function	Rule Signature
Unit Rule	$r_i^0 = e$	$r \leftarrow CONSTRAINT(v), UNIT(e)$
Apply Rule	$r_i^{j+1} = r_i^j \oplus f(v_i)$	$r \leftarrow r, v, APPLY(\oplus)$
Join Op	$r_i^{m+n} = r_i^m \oplus r_i^n$	Derived (No signature)

3.2.3 Rule Semantics

Every numerical simulation can be described by calculations on a finite set of entities. This maximal set of entities becomes the identity of the intersection operation on sets of entities in the simulation.

Definition 3.15 The universe of entities, represented by $\mathcal{Z}$, is defined as the set that contains all entities in the problem description.

Definition 3.16 The context *of a rule is defined as the intersection of the domains of all variable accessors and constraints in the rule specification. A rule actually represents a set of production rules where each entity in the context of a rule represents a production.*

A constraint qualifier, in addition to limiting the context of a rule to some set of values, also provides a guarantee that the rule will provide context for all entities in the constraint. If the intersection of the constraint and the context does not result in the constraint set, then the specification of the rule is inconsistent.

The semantics of a store provides that only one value can be assigned for each entity. Likewise, the semantics of the parameter specifies that only a single value can be assigned to a parameter variable identifier. This is similar to a single assignment semantic, but it differs slightly in that it is rather a specification of value uniqueness. For example, it is possible that the same rule may be applied twice to obtain an identical value. The implication of these semantics are that any specification that involves assigning multiple distinct values to the same entity in a store or to a parameter is an ambiguous specification.

3.2.3.1 Point-wise Rule Semantics

The most common rule that is involved in typical numerical PDE solution algorithms is the point-wise application of a calculation over a set of entities. The body of the rule specification describes how information is brought to the site of computations, while these computations create values that are bound to the variables in the head of the rule specification.

Definition 3.17 A point-wise *rule provides a set of values for every entity in the rule context and binds them to store values (possibly using a mapping operator) specified in the head of the rule. The point-wise rule is a rule of production, and as such the computation that provides values is assumed to be referentially transparent.*

Point-wise rules include recursive specifications where the computed variable identified in the head also exists in the body. Note that point-wise rules must comply with the semantics of the store, which dictates that for each variable bound to a store, there can be at most one value bound to any entity. Thus recursion always involves the mapping operator, and the number of levels of recursion is restricted by the number of entities described in the database. Point-wise rules are the only rules that have semantics that allow recursion.

Definition 3.18 A parameter rule is defined when the rule head consists of a parameter and the rule body does not contain any store variables. In this case the rule computes a single value that is bound to the parameter specified in the head.

3.2.3.2 Reduction Rule Semantics

A variable (store or parameter) may be defined by either point-wise rules or reduction rules, but not both. When a variable is defined by reduction rules, then its value is the result of a set of rule applications. The process of generating the reduction begins with the application of the unit rule. No value for a reduction can be generated without first binding a unit value to the target entities. This is accomplished through the application of a unit rule. The unit rule is identified through the $UNIT(e)$ qualifier. Typically the unit rule has a constraint qualifier that limits which entities are included in the computation. Once a unit rule has been applied to a set of entities, all apply rules, identified through the $APPLY(\oplus)$ signature, are applied to the unit rule initialized entities. It is guaranteed that each apply rule will be applied only once for each entity in its context. Once all apply rules complete, the final value of the entity is uniquely specified based on the assumption of the commutative and associative properties of the operator in the apply rules. Recursion is not allowed in reduction rule specifications for either unit or apply rules.

3.2.3.3 Stationary Rules and Time Promotion

In addition to point-wise, parameter, unit, and apply rules, there are a set of rules that are automatically generated when needed. These rules involve importing information into each iteration level, and are provided to reduce the tedium of making a complete specification. The first of these automatically generated rules is a promote rule: this rule duplicates values from variable identifiers lower in the iteration hierarchy. For example, a promote rule may promote position values from stationary time to the n iteration level as in "$x\{n\} \leftarrow x, PROMOTE$". Promotion rules are a special case of internally generated rules that have the value preserving property. That is, one can safely assume for this example that $x = x\{n\}$. To avoid conflicts with existing rules, promotion rules are only generated for variables that are not already computed by other rules within a given iteration level.

Another form of generalization with respect to iteration occurs with stationary rules, or rules where $\tau_h = \tau_b = \tau_{stationary}$. These rules represent generic transformations that should hold regardless of iteration level. Consider example 3.2 where pressure is obtained from density, a mixture gas constant, and temperature. This gas law relation is independent of iteration, for example "$p\{n\} \leftarrow \rho\{n\}, \tilde{R}\{n\}, T\{n\}$" should be equally valid. In general, rules that apply at constant time should be transportable to different iteration levels. Without this kind of rule promotion, many rules would need to be repeated at several iteration levels to accommodate many different roles of computations in the solver. To support this view of rule specification, any stationary rule may be promoted to any iteration level by renaming all variables in its signature. This sort of promotion leads to alternative derivation paths as illustrated in figure 3.3. Since variable promotion and rule promotion can both be applied, it is possible to first promote the variable and then promote the rule, or instead to apply the rule and then promote the results. However, since the semantics of these rules preserve value, either path will produce the same results. The particular path chosen depends on the priorities placed on memory efficiency versus computational efficiency in scheduling.

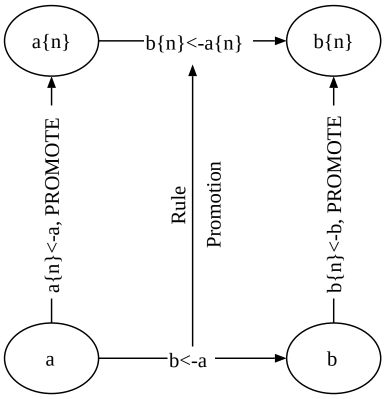


Figure 3.3: Two Alternative Promotion Paths from a to $b\{n\}$

The final automatically generated rule is the generalize rule. This rule takes the absolute offset results computed by build rules and generalizes these results to relative offsets. For example, `qn<-qn=0, GENERALIZE`. This generalization rule is identified by a the $GENERALIZE$ qualifier. It is of a similar form to the promotion rule except that the target variable always has a relative offset while the source variable always has a absolute offset. Like promotion rules, these rules preserve value.

3.3 Summary

This chapter has developed the necessary data structures and production rule specifications required to describe numerical computations on unstructured meshes. The database structure borrows basic ideas from relational structures and includes an approach of binding attributes to entities. These attributes have two basic forms: 1) those that bind values to entities, and 2) those that form relations between entities. These basic structures are summarized in figure 3.4.

Computations are described within this specification system utilizing a variation of the Horn clause. These Horn clauses represent the satisfaction of computations in the numerical model. They are divided into a few major classes including point-wise rules, parameter or singleton

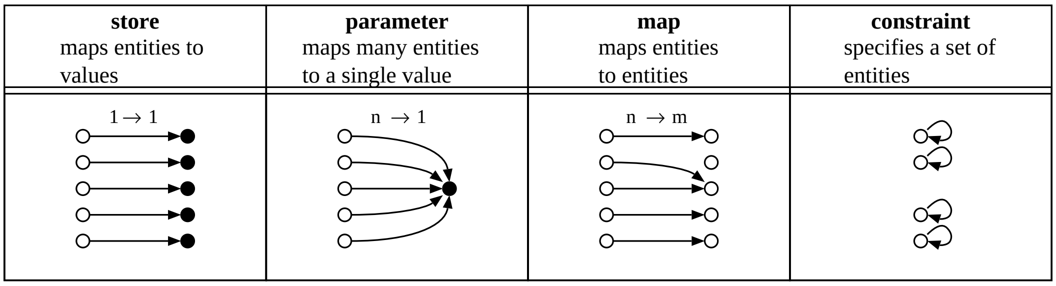


Figure 3.4: Four Basic Database Constructs

rules, and the reduction generating unit and apply rules. Increased flexibility is admitted by allowing recursive point-wise rules, however their recursion is guaranteed to be bounded due to value uniqueness that is ensured by the store and parameter database structures. This value uniqueness also provides a means of checking the consistency of a schedule, since it prohibits two rules from specifying conflicting attributes for any given entity. In addition, convenience rules are discussed which facilitate moving information into iteration levels without explicit specification.

CHAPTER IV

EXECUTION SCHEDULE GENERATION

The purpose of defining a specification language for describing numerical solution methods is to facilitate economical and correct implementations of numerical simulation programs. In this respect, Fortran can be considered a specification language, albeit a low level one. The process of converting the specification described in chapter III into a numerical simulation program requires scheduling of rule subroutines, much like Fortran requires compilation to generate an executable program. Unlike Fortran, this scheduling step must occur at run time. An execution schedule for rule subroutines is developed through a three step process. The first of these steps constructs a variable dependency graph. This dependency graph indicates the connections between variables that is implied by the rule database. The second step, identified as the existential analysis, computes the attributes that are defined for each entity. Finally, a pruning stage computes the entities that are required for the computation. The results of the pruning stage are then used to generate an execution schedule.

4.1 The Dependency Graph

The dependency graph is a directed graph that describes the flow of data through rule specifications. In this graph, both rules and variables (attribute names) are vertices, while data dependencies are indicated by graph edges. There is a symmetry between rules and variables: edges connect variables to rules, and then rules to variables, whereas rules to rules or variable to variable connections do not exist. The dependency graph is generated using rule signatures: for each variable in the body of a rule there is an edge from that variable vertex to the rule vertex and for each variable in the head of a rule there is an edge from that rule to the head variable. Thus a collection of rule signatures forms a dependency graph as illustrated in figure 4.1.

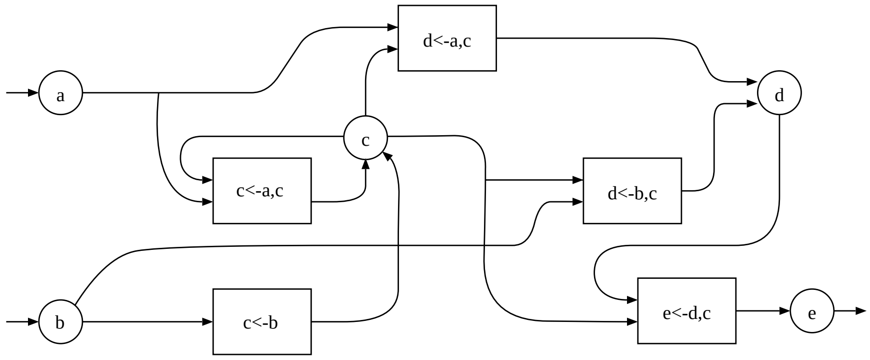


Figure 4.1: An Example Data Dependency Graph

At this stage, a dependency graph has been described for the rules stored in the rule database. Additional "derived" rules must also be placed in this dependency graph before it can be used to generate the execution schedule. These derived rules include the promotion of stationary rules and the generation of $PROMOTE$ and $GENERALIZE$ rules. Once these rules have been added to the graph, the dependency graph represents a data flow graph of all possible programs that may be derived from this rule database. It may not actually be necessary to derive this entire graph since the set of given facts can be used to prune this graph to include only those rules that will be executed.

Definition 4.19 A dependency graph *is defined by the ordered pair* $G = (V, E)$ *where* $V = \mathbb{R} \cup \mathbb{V}$ *is defined by the union of rule signatures,* $\mathbb{R}$*, and variable identifiers,* $\mathbb{V}$*, while the set* E *consists of ordered pairs of the form* (r, v)*, or* (v, r)*, where* $r \in \mathbb{R}$ *and* $v \in \mathbb{V}$*. If* $(r, v) \in E$ *then* v *must be an output of rule specification* r*, likewise if* $(v, r) \in E$ *then* v *must be an input of rule specification* r*.*

4.1.1 Partitioning the Dependency Graph

In the process of analyzing the dependency graph it is sometimes useful to remove a collection of rules from the graph so that different scheduling policies can be applied to them. Removing rules from the graph is accomplished through a graph partitioning procedure. A partition of the dependency graph, $G = (V, E)$, is formed based on a given subset of rules, $\mathbb{R}_p \subset \mathbb{R}$, where the local variables of the partition, denoted as $\mathbb{V}_p$, are given by

$$\mathbb{V}_p = \{v | [(v,r) \in E \vee (r,v) \in E] \Rightarrow (r \in \mathbb{R}_p)\}. \tag{4.1}$$

The subgraph of G, denoted by $G_p = (V_p, E_p)$ where $V_p = \mathbb{R}_p \cup \mathbb{V}_p$ and $E_p = \{(n_1, n_2) | (n_1, n_2) \in E, n_1 \in V_p, n_2 \in V_p\}$, forms the dependency graph of the partition. The partition of the graph is formed by removing the vertices, V_p from G and then inserting a new node that represents the external dependencies of the removed nodes. This new node, represented as r_p, is a pseudo-rule that has input edges specified by $\mathbb{I}_p$ and output edges specified by $\mathbb{O}_p$, where $\mathbb{I}_p = \{(v, r_p) | v \notin \mathbb{V}_p, (v,r) \in E, r \in \mathbb{R}_p\}$, and $\mathbb{O}_p = \{(r_p, v) | v \notin \mathbb{V}_p, (r,v) \in E, r \in \mathbb{R}_p\}$. Thus the new dependency graph, after partitioning is completed, is given as $\bar{G} = (\bar{V}, \bar{E})$ where $\bar{V} = (V - V_p) \cup \{r_p\}$ and $\bar{E} = (E \cup \mathbb{I}_p \cup \mathbb{O}_p) - E_p$.

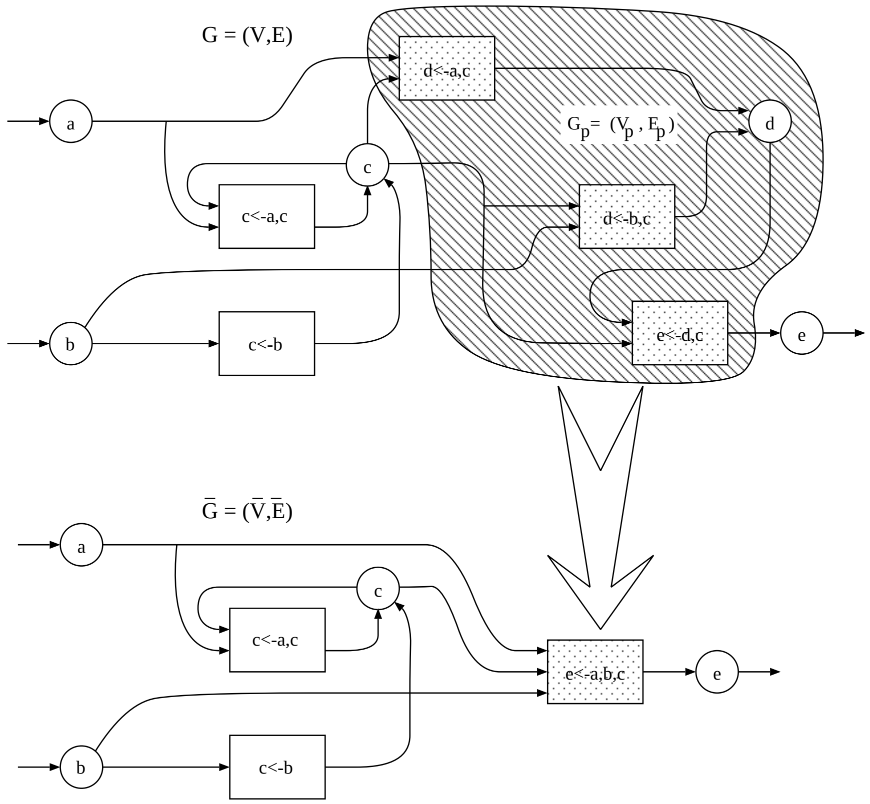


Figure 4.2: Partitioning a Dependency Graph

To illustrate the partitioning of a dependency graph, consider the graph in figure 4.1. If this graph is partitioned based on the rule set $\mathbb{R}_p = \{(d \leftarrow a, c), (d \leftarrow b, c), (e \leftarrow d, c)\}$, then the set variables local to the partition is given by $\mathbb{V}_p = \{d\}$, while the graphs G_p and $\bar{G}$ are illustrated in figure 4.2. The generation of the pseudo-rule $e \leftarrow a, b, c$ is used to represent the partition dependencies of G_p in the partitioned graph $\bar{G}$.

4.1.2 Partitioning the Iteration Hierarchy

A simplification of the dependency graph can be accomplished by partitioning it based on iteration levels. This is accomplished by sorting rules based on the maximum of the head and body times (τ_h and τ_b). If the partitioning proceeds from the leaf nodes of the iteration hierarchy to stationary time, then the partitioning will form a hierarchy of graphs that mimics the iteration hierarchy. Once this partitioning is performed there is a separation of concerns related to iteration: each level of iteration is described by an independent dependency graph. Thus computations that are performed by an iteration will appear as a pseudo-rule in the dependency graphs that use these results, hiding many of the particular details involved with analysis of iterations.

4.1.3 Treatment of Recursive Dependencies

The first step in generating an execution schedule is determining the order in which rules must be satisfied. When the dependency graph is a directed acyclic graph (DAG), this execution order can be determined using a topological sort[37]. Unfortunately, the graph is not guaranteed to be a DAG, since rules may include recursive specifications. Notice, for example, that the recursive rule $c \leftarrow a, c$ in figure 4.1 forms a loop. Actually, a recursion loop in the graph represents iteration over entities, and so a loop of this form must be repeatedly evaluated until all attributes have been generated. In order to transform this dependency graph into a DAG, it is necessary to remove these recursive loops so that they may be treated separately. This is done by noting that recursion produces strongly connected components in the dependency graph. (A strongly connected component is defined as a subset of vertices of a directed graph where each vertex in the subset is reachable from every other vertex.) For example, the strongly connected component formed by the recursive specification in figure 4.1 is the rule $c \leftarrow a, c$ plus the variable c, as illustrated by the shaded section of figure 4.3.

The strongly connected component of figure 4.3 can be removed from the graph in such a way as to recover a DAG. This is accomplished by first noting that a strongly connected component

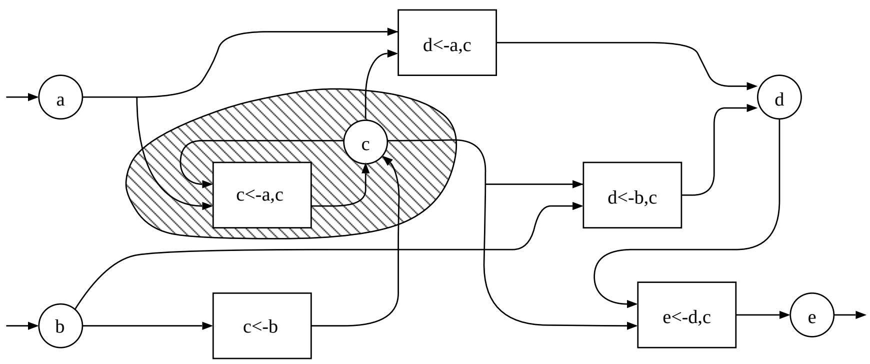


Figure 4.3: Identifying Recursion as a Strongly Connected Component

will consist of a set of rules and a set of variables. (There must be at least one variable and one rule due to the lack of edges directly between rules or variables.) It is not possible for two strongly connected components to share a variable since in this case both strongly connected components would become strongly connected to one another (and thus not individual components). The strongly connected components of a graph can be identified efficiently using common graph algorithms[37]. Strongly connected components can be removed from the dependency graph by partitioning the graph. Unfortunately, a straightforward partitioning based on the rules in the component set does not remove the recursive loop from the graph. In order to remove this loop it is necessary to create input and output versions of the variables included in the component. This is accomplished by aliasing all variables[1] of the strongly connected component to input and output versions and then moving all edges that connect to the component variables to their input aliases and likewise all edges leaving the component variables are reconnected to the output aliases. Now the component variables are local to the partition, thus the pseudo-rule that represents the component will not be recursive, as illustrated in figure 4.4.

4.1.4 Treatment of Iteration

In addition to partitioning the dependency graph based on iteration, there is a need to partition iteration dependency graphs into components that initialize the loop and components that perform the iteration. The approach to identifying the looping component involves temporarily adding edges to the graph that represent the relabeling of variables that occurs

[1]Aliasing temporarily violates the assumption that variables do not connect to variables. This violation is removed once the partitioning is complete.

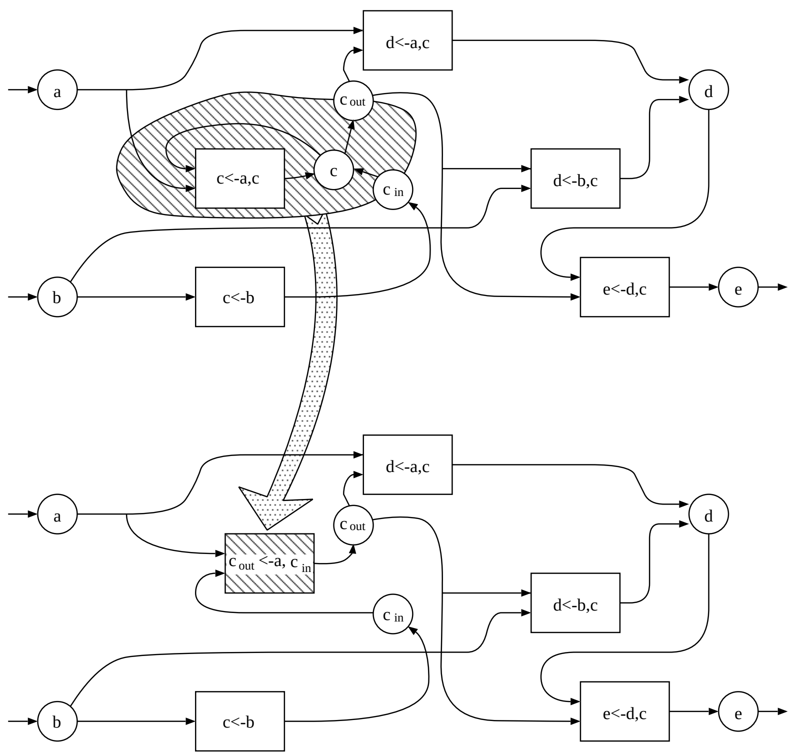


Figure 4.4: Creating a Partition for Recursion

when iterations advance (e.g., u^{n+1} becomes u^n). This is accomplished by creating an edge from every variable in the graph of the form $\alpha\{\beta+\theta\}$ to the variable $\alpha\{\beta+\theta-1\}$. Once these edges are added, a component sort will identify the looping components as recursion through temporal renaming. At this point, the added edges are removed from the graph and then a partition of the graph is formed using the rules identified by the strongly connected component.

4.1.5 Summary of Dependency Graph Analysis

At this stage the dependency graph has been generated. It has been sorted such that it forms a hierarchy of DAGs. This hierarchy of DAGs is composed of three principal types of pseudo-rules: 1) pseudo-rules representing recursion over entities, 2) pseudo-rules representing the creation of iteration, and 3) pseudo-rules that define the iteration loop itself. Also, since all of the analysis made thus far only depends on the rule database, these computations can be performed just once, when the rule database is assembled, rather than at every simulation run. In addition, since these computations are independent of the size of the mesh, their expense does not grow with the simulation size.

4.2 Existential Deduction

There are fundamentally two types of queries that can be made regarding attributes: what entities have attribute α, and what are the values of attribute α for some set of entities. Note that knowing the value of an attribute implies existence; however, while knowledge of values requires actual execution of rules, the existence of attributes can be derived before an execution schedule is generated. In fact, this existential information is required in order to generate an execution schedule. The derivation of this information is called the existential deduction phase of schedule generation. The existential deduction is determined by the function $\chi(\alpha,\mathbb{F},\mathbb{R})$ which returns the set of entities that have attribute described by the variable named α given the database of facts, $\mathbb{F}$, and the database of rules given in $\mathbb{R}$. This function is expressed as the union of more primitive existential functions as in

$$\chi(\alpha,\mathbb{F},\mathbb{R}) = \chi(\alpha,\mathbb{F}) \cup \left\{ \bigcup_{r_j\in\mathbb{R}} \chi(\alpha,\mathbb{F},\mathbb{R},r_j) \right\}, \tag{4.2}$$

where $\chi(\alpha, \mathbb{F})$ represents the existence of entities due to definition in the fact database $\mathbb{F}$, while $\chi(\alpha, \mathbb{F}, \mathbb{R}, r_j)$ represents the attributes that are provided by rule r_j.

The solution of equation (4.2) represents the existential deduction. This function has two important properties: it is monotonic and bounded with respect to $\mathbb{F}$ and $\mathbb{R}$. The function is monotonic since rules may only add attributes to entities, thus any addition of rules or facts can only create additional attribute bindings. In addition since rules create values (stores and parameters), not entities (maps and constraints), the set of entities given by $\chi(\alpha, \mathbb{F}, \mathbb{R})$ cannot be any larger than the set of all entities described in $\mathbb{F}$. These points will become evident as the derivation of the specific rule deductions, $\chi(\alpha, \mathbb{F}, \mathbb{R}, r_i)$, are discussed.

4.2.1 Existential Deductions for Point-wise Rules

The existential deduction for point-wise rules involves the entity by entity application of modus ponens. For example, given the rule $c \leftarrow a, b$, an attribute c is implied by the satisfaction of attributes a and b. Thus the existential function for this rule is written as the intersection of the existential functions of a and b:

$$\chi(c, \mathbb{F}, \mathbb{R}, c \leftarrow a, b) = \chi(a, \mathbb{F}, \mathbb{R}) \cap \chi(b, \mathbb{F}, \mathbb{R}). \tag{4.3}$$

Thus if a is defined on the interval $[0 - 10]$ and b is defined on the interval $[5 - 15]$, then c is defined by this rule over the interval $[5 - 10]$.

If a mapping is specified in the body of a rule (as in $c \leftarrow m \rightarrow b$) then this mapping must be included in the existential deduction. Following a similar procedure to the one described above, the existential results of applying a mapping can be given as

$$\beta \overset{\chi}{\rightarrow} \alpha = \{i | i \in \chi(\beta, \mathbb{F}, \mathbb{R}) \wedge [(i, j) \in \beta \Rightarrow j \in \chi(\alpha, \mathbb{F}, \mathbb{R})]\} \tag{4.4}$$

As an example, consider a map given as $m = \{(1, 0), (1, 10), (2, 0), (3, 10), (4, 11)\}$ and an attribute b that is defined on the interval $[5, 15]$, then following equation (4.4) the existential mapping, $m \overset{\chi}{\rightarrow} b$, produces the interval $[3, 4]$.

This gives the basic approach for determining the existential deduction for point-wise rules: the point-wise rule specifies attributes at the intersection of the arguments in its body. For the general case of a rule given by the signature

$$r_p = \alpha \leftarrow \beta_1 \rightarrow \alpha_1, \beta_2 \rightarrow \alpha_2, \cdots, \beta_i \rightarrow \alpha_i, \cdots \beta_n \rightarrow \alpha_n, \alpha_{n+1}, \cdots, \alpha_j, \cdots \alpha_m, constraint, \quad (4.5)$$

where *constraint* consists of a set of entities that satisfy a given constraint equation, then the existential deduction for rule r_p can be given by:

$$\chi(\alpha, \mathbb{F}, \mathbb{R}, r_p) = \bigcap_{i=1\cdots n} \left[\beta_i \xrightarrow{\chi} \alpha_i\right] \cap \bigcap_{j=n+1\cdots m} [\chi(\alpha_j, \mathbb{F}, \mathbb{R})] \cap constraint. \quad (4.6)$$

Equation (4.6) is a formalization of the property that a rule cannot have a well formed result unless the information specified in the body can be provided. Notice that this rule, in combination with equation (4.2), forms a recursive specification of existential deduction. The evaluation of the existential function (4.2) when recursive rule specifications are present requires an iterative evaluation procedure.

4.2.2 Existential Deductions for Singleton Rules

Singleton rules represent a single value computation that is associated to a collection of entities. This interpretation of value is given by the parameter database construct. The referential transparency of rules means that this calculation can be viewed as a single computation or as a collection of computations carried out for each entity represented by the parameter without loss of correctness. Since existential deduction does not require values to proceed, either of these models can be evaluated with equal efficiency. By assuming that the singleton rule evaluation represents a computation for each entity that is satisfied, the existential deduction for the singleton rule can be computed by way of the point-wise rule equation (4.6).

4.2.3 Existential Deductions for Reduction Rules

Utilizing the join operation, reductions are accomplished by way of two rule specifications: a unit rule and a collection of apply rules. Since the apply rules are only invoked for entities that have been initialized to the unit value, apply rules cannot create attributes. Thus apply rules can be neglected for the purposes of existential deduction. Only unit rules need to be considered during the existential deduction phase. The existential deduction for unit rules can proceed using equation (4.6).

4.2.4 Evaluating χ

Equations (4.2) and (4.6) form the description of the existential deduction. These equations combine to form a recursive specification, and so an implementation of the existential deduction is not straightforward. However, it is possible to develop an iterative algorithm to solve these equations. The iterative algorithm begins with the existential properties of the facts in $\mathbb{F}$ as an initial guess to the solution of χ. Thus the first step of the iterative algorithm begins with

$$\chi^0(\alpha, \mathbb{F}, \mathbb{R}) = \chi(\alpha, \mathbb{F}). \tag{4.7}$$

Once this initial guess for χ is established, a refined guess can be constructed by determining what new attributes are generated by rules in $\mathbb{R}$. This iterative procedure is accomplished by solving

$$\chi^{n+1}(\alpha, \mathbb{F}, \mathbb{R}) = \chi^n(\alpha, \mathbb{F}, \mathbb{R}) \cup \left\{ \bigcup_{r_j \in \mathbb{R}} \chi^n(\alpha, \mathbb{F}, \mathbb{R}, r_j) \right\}. \tag{4.8}$$

This iterative method will produce a monotonically increasing series of estimations for χ. Given that the number of entities is bounded by the finite set $\mathcal{Z}$ and that the number of attributes is similarly bounded for any finite specification, then this iterative algorithm will always converge[2] in a finite number of steps.

4.3 Pruning and Schedule Generation

The existential deduction and the execution schedule are closely related: the existence of an attribute is implied either because it was given or because it can be computed. Thus it is not surprising that the iterative algorithm for performing the existential deduction can in turn be

[2]Convergence is obtained when $\chi^{n+1}(\alpha, \mathbb{F}, \mathbb{R}) = \chi^n(\alpha, \mathbb{F}, \mathbb{R})$ for all α. At this point, further iterations will not produce any additional information.

used to produce an execution schedule. Notice that each step of the iterative algorithm given by equation (4.8) produces a set of computed attributes given by

$$C^n(\alpha, \mathbb{F}, \mathbb{R}) = \chi^{n+1}(\alpha, \mathbb{F}, \mathbb{R}) - \chi^n(\alpha, \mathbb{F}, \mathbb{R}). \tag{4.9}$$

Moreover, since each of the computed values in C^n only depends on values computed in previous iterations of the algorithm, the computations described by C^n are free of serializing dependencies (*i.e.* concurrent). Unfortunately, the schedule generated in this fashion will compute all possible attributes that can be computed from the attributes specified in the fact database. In general we would like to prune this execution schedule to only those calculations that contribute to the requested goal.

The execution schedule denoted by equation (4.9) can be pruned so that it only includes those calculations required to provide the requested variables. The pruned schedule is obtained by transposing the head and bodies of rule specifications. For example, rule $a \leftarrow b, c$ can be represented as the "need" rule, $b, c \leftarrow a$ (can be read as 'a' generates need for 'b' and 'c'). Once the set of transposed rules is generated, the pruning is performed by calculating an existential deduction for the transposed rule database given the existential information derived for the requested variables. For this transposed rule set, the existential deduction derives variable requests. Thus the pruning step can be defined as

$$P_\chi(\alpha, \mathbb{G}, \mathbb{R}) = \chi(\alpha, \mathbb{G}, \mathbb{R}_P), \tag{4.10}$$

where the goal fact database, $\mathbb{G}$, is defined such that

$$\chi(\alpha, \mathbb{G}) = \begin{cases} \chi(\alpha, \mathbb{F}, \mathbb{R}) \text{ if } \alpha \in \text{goal}, \\ \{\} \text{ otherwise}, \end{cases} \tag{4.11}$$

while $\mathbb{R}_P$ contains the transposed rules of $\mathbb{R}$. Finally the pruned execution schedule is given by the sequence of pruned computations that are derived by equation (4.9), as given in the algorithm shown in figure 4.5.

do $n = 0, \cdots, \text{num\_steps}$
  pardo $r_i \in \mathbb{R}$
    pardo $e \in \chi(\alpha, \mathbb{F}, \mathbb{R}, r_i) \cap C^n(\alpha, \mathbb{F}, \mathbb{R}) \cap P_\chi(\alpha, \mathbb{G}, \mathbb{R})$
      evaluate r_i for entity e
    endpardo
  endpardo
enddo

Figure 4.5: Derived Parallel Execution Schedule

4.4 Schemes for Efficient Schedule Generation

The previous section described the generation of an execution schedule using the results of the existential deduction. At first glance it may appear that the existential function χ should be expensive to compute. However, the identification of a few common properties of simulations of this class can be used to yield an efficient schedule generation algorithm. First some discussion needs to be made with regard to basic utilities required to evaluate equation (4.8). For example, the existential deduction algorithm involves finding the unions and intersections of sets of entities. In general, entities are identified by sets of integer identifiers. Moreover, these integer identifiers tend to be labeled in contiguous segments: entities labeled $[1, \cdots, 10000]$ may represent the faces in the problem, for example. Thus entities of like kind are typically grouped together. As a result, it is advantageous to represent sets of entities by a list of intervals. If an ordering is maintained on these intervals, then intersection and union operations can be obtained with work on the order of the number of intervals[37]. Since the number of intervals is typically small and independent of problem size, most of these intersections and unions can be accomplished in effectively constant time. In addition, it is assumed that before the existential deduction begins, transposes of all maps can be pre-computed. This transpose can be performed with work proportional to the number of elements in the map. Thus as a preamble to the discussion of the evaluation of χ, it is assumed that unions and intersections can be performed very quickly on very large sets of entities, and that the transposes of maps have been pre-computed at little cost.[3]

[3]These transposes can be disposed of before execution of the schedule, so their memory cost is hidden by the memory requirements of the simulation.

4.4.1 Optimizations for Evaluation of χ

The iterative algorithm for existential deduction described by equations (4.7) and (4.8) implicitly contain the dependency graph described earlier. This is due to the recursive dependence of the χ function, as described in equation (4.6). Thus the information that was developed in the formulation of the dependency graph can be used to optimize the evaluation of χ. For example, if the goal variables are linked back to the given facts in the dependency graph, a component sort can be used to identify those rules that will participate in the computations. Thus it is possible to reduce the set of rules that need to be considered in the evaluation of χ.

The most important use of the dependency graph is the separation of cyclic structures. Once the dependency graph is represented as a DAG, then χ is no longer a recursive function. For this case, a topological sort of the dependency DAG produces a sequence of attributes for which χ can be evaluated without iteration. Since the recursive elements of the specification can be partitioned out of the dependency graph, it is straightforward to reserve the more general iterative algorithm for these recursive specifications.

It is not possible to completely avoid the recursive nature of the existential deduction if the rule database includes recursive specifications. In this case it is necessary to evaluate the more general χ function. However, in this case it is possible to limit the analysis to only those rules and variables that are contained within the recursion component. In addition, it can be noted that the new entities that were identified in the previous iteration of the evaluation of equation (4.8) will be the source of the critical information that is required to satisfy rules in this iteration. Thus, the deduction can be limited to a search from those entities described by $C_n(\alpha, \mathbb{F}, \mathbb{R})$. The search proceeds by first following the transpose of any maps in the rule specification and producing a list of candidate entities that may be satisfied by the new information computed in the previous iteration. Once this list is formed, a straightforward evaluation of equations (4.4) and (4.6) for this list will produce the values for χ^{n+1}. Thus the evaluation of the existential deduction for recursive blocks is of the order of the product of the number of recursive variables and the number of entities deduced.[4]

[4]This is assuming that the maps under consideration have a bounded valence.

CHAPTER V

A NUMERICAL MODEL FOR INVISCID FLOWS WITH FINITE-RATE CHEMISTRY

In this chapter a numerical model for flows of chemically reacting gases is discussed in some detail. The particular model employed in the following sections will neglect transport terms such as viscosity, heat conduction and mass diffusion. Although many important flow regimes involve these terms, there are several classes of problems which can be solved under these assumptions: for example, high speed flows over blunt bodies, rocket nozzle combustion, and pre-mixed detonation. In addition, the development of an inviscid numerical model is an important first step in the development of a more general modeling capability.

This chapter is divided into two sections: the first one describes the mathematical model of the physical phenomena of interest (these are the model equations that are approximated numerically); the second one describes the numerical model that is generated by developing discrete numerical approximations to the mathematical equations.

5.1 Model Equations

The development of a numerical model for mixtures of chemically reacting gases begins with a model of fluid motion. Gases can be modeled by the Navier-Stokes equations in cases where they can be treated as a continuum. This continuum hypothesis holds in most circumstances of engineering interest, with an exception for extremely rarefied gases (which are found at the edge of the atmosphere, for example). The Navier-Stokes equations are an open model: they are incomplete by themselves, and require additional equations to produce a complete (closed) set of equations. The mathematical model for chemically reacting gases is developed by providing the Navier-Stokes equations with the proper closure equations.

5.1.1 The Navier-Stokes Equations

The Navier-Stokes equations are derived from continuum mechanical interpretations of the physical conservation laws. The derivations will not be presented here, but can be found in a variety of texts. The form of the equations presented here were derived in Warsi's text[38].

The Navier-Stokes equations are composed of three separate equations: a mass conservation equation, a momentum conservation equation, and an energy conservation equation. The equations are defined in terms of the continuum field variables density, ρ, velocity, $\tilde{u}$, total energy, e_0, pressure, p, and temperature T. The first of these equations, the mass conservation equation, is given in tensor form by the equation

$$\frac{\partial \rho}{\partial t} + \mathrm{div}(\rho\tilde{u}) = 0. \tag{5.1}$$

Similarly the momentum conservation equation is given by

$$\frac{\partial}{\partial t}(\rho\tilde{u}) + \mathrm{div}(\rho\tilde{u}\tilde{u} + p\tilde{\tilde{I}}) = \rho\tilde{f} + \mathrm{div}\ \tilde{\tilde{\sigma}}, \tag{5.2}$$

where the deviatoric stress tensor, $\tilde{\tilde{\sigma}}$, is given in Cartesian form as

$$\tilde{\tilde{\sigma}} = (\lambda \mathrm{div}\ \tilde{u})\,\tilde{\tilde{I}} + \mu\left(\mathrm{grad}\ \tilde{u} + (\mathrm{grad}\ \tilde{u})^T\right). \tag{5.3}$$

In this equation, μ represents the fluid viscosity, while λ is normally derived from Stokes' law which states that $3\lambda + 2\mu = 0$. The vector function $\tilde{f}$ represents a body force such as a force exerted by gravitational or electro-magnetic means.

The final equation of the Navier-Stokes equations is the energy conservation equation. This equation, expressed in terms of the total energy, $e_0 = \frac{1}{2}\tilde{u}\cdot\tilde{u} + e_{internal}$, is given by the equation

$$\frac{\partial}{\partial t}(\rho e_0) + \mathrm{div}\left((\rho e_0 + p)\,\tilde{u}\right) = \mathrm{div}\left(\tilde{\tilde{\sigma}}\cdot\tilde{u}\right) + \mathrm{div}\left(k\ \mathrm{grad}\ T\right) + \rho\left(\tilde{f}\cdot\tilde{u}\right). \tag{5.4}$$

In this equation, k is the thermal conductivity of the fluid. The inviscid formulation of the Navier-Stokes equation is found by taking the limit of these equations as μ approaches zero. Typically an inviscid formulation also implies the additional assumption of $k = 0$.

5.1.2 Closure Using Finite-Rate Chemistry

The Navier-Stokes equations leave pressure undefined. Thus pressure must be defined before they represent a complete model. For the example described here, pressure will need to be defined in terms of a fluid that is in chemical non-equilibrium. The complete specification of pressure in terms of finite-rate chemistry equations will complete the model.

For the model equations presented here, pressure is determined from Dalton's Law, which states that the pressure of a mixture of gases is the sum of the partial pressures. Each species partial pressure is obtained from the ideal gas law. Thus the model equations assume a mixture of thermally perfect gases that have a pressure defined by the equation

$$p = \sum_{s=1}^{NS} p_s = \sum_{s=1}^{NS} \rho_s R_s T, \tag{5.5}$$

where NS is the number of gas species in the mixture, R_s is the species gas constant, and ρ_s is the species density. The species gas constant is computed by the equation

$$R_s = \frac{\hat{R}}{\mathcal{M}_s}, \tag{5.6}$$

where $\mathcal{M}_s$ is the species molecular mass, and $\hat{R}$ is the universal gas constant.

For a chemically reacting flow, the fluid will contain a mixture of reactants (chemical species). Each fluid element will have a fraction of its mass allocated to each species, as represented by the variable $Y_s = \rho_s/\rho$. For each chemical species, a conservation equation that is similar to the global mass conservation equation can be written as follows

$$\frac{\partial \rho_s}{\partial t} + \operatorname{div}(\rho_s \tilde{u}) = \dot{w}_s, \tag{5.7}$$

where $\dot{w}_s$ is the rate of production of species s due to chemical reactions. Assuming there are NS species, then the conservation of mass of equation (5.1) requires that

$$\sum_{s=1}^{NS} \dot{w}_s = 0. \tag{5.8}$$

When this holds, the species equations given in (5.7) are consistent with equation (5.1).

Equation (5.5) relates pressure to temperature, T. The Navier-Stokes equations are expressed in terms of conserved variables such as momentum and energy and do not directly describe temperature, which is determined from the internal energy of the gas. Recall that $e_0 = \frac{1}{2}\tilde{u} \cdot \tilde{u} + e_{internal}$ where $e_{internal}$ is the internal thermodynamic energy of the gas. The relationship between $e_{internal}$ and temperature is given by the caloric equation of state. For a gas consisting of a mixture of species, the caloric equation of state is computed as the sum of species energies, as in

$$e_{internal} = \sum_{s=1}^{NS} Y_s e_s. \tag{5.9}$$

For ideal gases, the relationship between internal energy and temperature is linear. For most reacting gases, however, the billiard-ball model of a gas composed of indivisible structureless particles breaks down. Molecules, in addition to having translational movement like a hard sphere, can also have rotational and vibrational modes. At sufficient temperatures these additional effects lead to a non-linear relationship between internal energy and temperature. These gases are termed calorically imperfect. The general form of the equation for species energy in a calorically imperfect gas is given by the equation

$$e_s = \int_{T_{ref}}^{T} c_{v\,s}(\tau)d\tau + h_{f\,s}, \tag{5.10}$$

where $h_{f\,s}$ is the species heat of formation, or the energy required to create that species at T_{ref}, and $c_{v\,s}$ is the species specific heat. If the components of the species internal energy (translational, rotational, and vibrational) are assumed to be in thermodynamic equilibrium, and the vibrational mode is modeled using a simple harmonic oscillator[39], then the species energy is given as

$$e_s = n_s R_s T + \sum_{v=1}^{NVT_s} \frac{R_s \theta_{v,s}}{e^{\theta_{v,s}/T} - 1} + h_{f\,s}, \tag{5.11}$$

where NVT_s is the number of vibrational modes, $\theta_{v,s}$ is the vibrational temperature(s), and n_s specifies the translational and rotational contributions to internal energy.

At this stage, the equations will be closed once the species production rates, $\dot{w}_s$, are specified for a general chemistry model. One can represent chemical reactions as a set of equations of the form

$$\nu'_{1,r}X_1 + \cdots + \nu'_{s,r}X_s + \cdots + \nu_{NS,r}X_{NS} \leftrightarrows \nu''_{1,r}X_1 + \cdots + \nu''_{s,r}X_s + \cdots + \nu''_{NS,r}X_{NS}, \quad (5.12)$$

where X_s are the species names, whereas $\nu'_{s,r}$ and $\nu''_{s,r}$ are the stoichiometric coefficients of the reactants and the products, respectively. Given these reaction equations, the species production terms can be expressed as by the equation

$$\dot{w}_s = \left(\frac{d\rho_s}{dt}\right)_{chemistry} = \mathcal{M}_s \sum_{r=1}^{NR} (\nu''_{s,r} - \nu'_{s,r}) \left[k_{f,r} \prod_{l=1}^{NS} \left(\frac{\rho_l}{\mathcal{M}_l}\right)^{\nu'_{l,r}} - k_{b,r} \prod_{l=1}^{NS} \left(\frac{\rho_l}{\mathcal{M}_l}\right)^{\nu''_{l,r}} \right], \quad (5.13)$$

where NR is the number of reactions, and $k_{f,r}$, $k_{b,r}$ are the forward and backward reaction rates for reaction r. For a detailed derivation of equation (5.13) see Vincenti and Kruger[39] chapter VII.

Generally the forward and backward reaction rates are determined through empirical means and are tabulated through the use of curve fits. The most common curve fit used for chemistry reaction rates is the Arrhenius equation, given as

$$K(T) = CT^{\eta}e^{-\theta/T}. \quad (5.14)$$

At this point the mathematical model is complete – there is sufficient information to close the Navier-Stokes equations. There are some additional concerns, however, which relate to the equilibrium conditions, or those conditions that exist when sufficient time has passed for the composition of the gas to remain constant due to the internal balance of production and consumption of species. When the equilibrium condition is met, $\dot{w}_s$ should be zero for all species. Thus, equation (5.13) implies that the equilibrium mixture satisfies the equation

$$K_{c,r} = \frac{k_{f,r}}{k_{b,r}} = \left[\frac{\prod_{l=1}^{NS} \left(\frac{\rho_l}{\mathcal{M}_l}\right)^{\nu'_{l,r}}}{\prod_{l=1}^{NS} \left(\frac{\rho_l}{\mathcal{M}_l}\right)^{\nu''_{l,r}}} \right]_{equilibrium} . \quad (5.15)$$

In equation (5.15), $K_{c,r}$ is the so called equilibrium constant (which is actually a function of temperature). The value for $K_{c,r}$ can be determined by two different means. As can clearly be seen, $K_{c,r}$ can be determined from the forward and backward reaction rates. In addition, like most equilibrium states, $K_{c,r}$ can be determined from thermodynamic analysis. Typically reaction rates are difficult to measure accurately, and as a result the equilibrium constant dictated by the forward and backward reaction rates may not be consistent with the thermodynamic properties of the species participating in the reactions. A common practice used to resolve this potential for inconsistency is to provide a forward(backward) reaction rate and then use the thermodynamic equilibrium $K_{c,r}$ to compute the complementary backward(forward) reaction rate. Thus the equilibrium conditions are consistent with the thermodynamic properties of the species, while the rates of reactions match empirical values.

The thermodynamic $K_{c,r}$ is determined by the minimization of the mixture's Gibb's free energy. The expressions for the thermodynamic $K_{c,r}$ presented here are borrowed from the work of Carey Cox [40]. From that work, the equation for $K_{c,r}$ is given as

$$K_{c,r} = \left(\frac{p_{ref}}{\tilde{R}T}\right)^{\sum_{s=1}^{NS}\left(\nu''_{s,r}-\nu'_{s,r}\right)} \exp\left[-\sum_{s=1}^{NS}\Omega_s(T)\left(\nu''_{s,r}-\nu'_{s,r}\right)\right]. \tag{5.16}$$

In the previous equation the function $\Omega_s(T)$ is determined by satisfying the equation

$$\Omega_s(T) = \frac{1}{R_s T}\left[\int_{T_{ref}}^{T} c_{p_s}(\tau)d\tau - T\int_{T_{ref}}^{T}\frac{c_{p_s}(\tau)}{\tau}d\tau + h_{f_s} - T s_{f_s}\right]. \tag{5.17}$$

In general, the solution of this equation depends on the particular thermodynamic model used. For the model stated in equation (5.11), $\Omega_s(T)$ is given as

$$\Omega_s(T) = 1 + n_s + \sum_{v=1}^{NVT_s}\ln\left(1 - e^{\frac{-\theta_{v,s}}{T}}\right) - (1+n_s)\ln T + \frac{h_{f_s}}{R_s T} - \frac{s_s^0}{R_s}, \tag{5.18}$$

where s_s^0 is a constant of integration that is derived from measured thermodynamic properties. The value of s_s^0 for this thermodynamic model is given as

$$\begin{aligned} s_s^0 =& (s_{ref})_s - R_s(1+n_s)\ln(T_{ref}) - \\ & \sum_{v=1}^{NVT_s}\left(\frac{\theta_{v,s}}{\left(e^{\theta_{v,s}/T_{ref}}\right)T_{ref}} - \ln(1 - e^{\theta_{v,s}/T_{ref}})\right), \end{aligned} \tag{5.19}$$

where s_{ref} is the measured entropy at reference temperature T_{ref} and pressure p_{ref} (from 5.16).

5.2 Numerical Formulation

The numerical solution of the chemically reacting fluid equations presented in the previous section is obtained by applying the finite volume method. This method is frequently used because it can guarantee that numerical truncation error does not violate conservation properties. In this study, the method is applied to the species conservation equation (5.7), the momentum conservation equation (5.2), and the energy conservation equation(5.4). Since the model of interest is inviscid, μ, k, and $\tilde{f}$ are assumed to be zero.[1] Thus, the model equations expressed in integral form for an inviscid gas can be be written as follows

$$\begin{aligned}
&\int_{\Omega(t)} \frac{\partial}{\partial t}\rho_s \; dV + \int_{\Omega(t)} \operatorname{div}(\rho_s \tilde{u}) \; dV = \int_{\Omega(t)} \dot{w}_s \; dV, \\
&\int_{\Omega(t)} \frac{\partial}{\partial t}(\rho \tilde{u}) \; dV + \int_{\Omega(t)} \operatorname{div}(\rho \tilde{u}\tilde{u} + p\tilde{\tilde{I}}) \; dV = 0, \\
&\int_{\Omega(t)} \frac{\partial}{\partial t}(\rho e_0) \; dV + \int_{\Omega(t)} \operatorname{div}\left((\rho e_0 + p)\,\tilde{u}\right) \; dV = 0.
\end{aligned} \tag{5.20}$$

Before performing the numerical integrations and developing the finite volume formulation, it is helpful to transform equations (5.20) into a form that is more readily usable in a numerical method. In particular, the integration of a time derivative and the computation of the divergence operator is expensive to perform numerically. The equations can be transformed in such a way as to remove these terms by making use of two identities. The first of these identities describes the process of taking time derivatives of integrals over general moving and deforming domains[41]. This identity is given as

$$\frac{d}{dt}\int_{\Omega(t)} f(\tilde{x}, t) \; dV = \int_{\Omega(t)} \frac{\partial f}{\partial t} \; dV + \int_{\partial\Omega(t)} f \tilde{u}_{\Omega} \cdot \tilde{n} \; dS, \tag{5.21}$$

where Ω represents the general control volume, $\partial\Omega$ is its surface, $\tilde{u}_\Omega$ is the surface velocity, and $\tilde{n}$ is the outward pointing surface normal. The second identity, Green's theorem[38], is also used to replace the divergence operators with surface integrals, more readily tackled by numerical means.

[1] A correction term will be described that ensures that the computed solution is the limit of the solution to the original equations as μ tends to zero. For now, the distinction between the limit and simply assigning μ a value of zero will be ignored.

Green's theorem is expressed as

$$\int_{\Omega} \mathrm{div}\tilde{u}\ dV = \int_{\partial\Omega} \tilde{u}\cdot\tilde{n}\ dS. \tag{5.22}$$

The application of these identities to the model equations (5.20) yields the following transformed model equations

$$\begin{aligned}
&\frac{d}{dt}\int_{\Omega(t)} \rho_s\ dV - \int_{\partial\Omega(t)} \rho_s(\tilde{u}_\Omega\cdot\tilde{n})\ dS + \int_{\partial\Omega(t)} \rho_s(\tilde{u}\cdot\tilde{n})\ dS = \int_{\Omega(t)} \dot{w}_s\ dV,\\
&\frac{d}{dt}\int_{\Omega(t)} \rho\tilde{u}\ dV - \int_{\partial\Omega(t)} \rho\tilde{u}(\tilde{u}_\Omega\cdot\tilde{n})\ dS + \int_{\partial\Omega(t)} (\rho\tilde{u}\tilde{u} + p\tilde{\tilde{I}})\cdot\tilde{n}\ dS = 0,\\
&\frac{d}{dt}\int_{\Omega(t)} \rho e_0\ dV - \int_{\partial\Omega(t)} \rho e_0(\tilde{u}_\Omega\cdot\tilde{n})\ dS + \int_{\partial\Omega(t)} (\rho e_0 + p)\,(\tilde{u}\cdot\tilde{n})\ dS = 0.
\end{aligned} \tag{5.23}$$

To facilitate the economical description of the numerical solution method, the integral conservation equations (5.23) are rewritten in terms of a conservative fluid state variable. In this form, the conservation equations are written for a cell c, as follows

$$\frac{d}{dt}\int_{\Omega_c(t)} Q\ dV + \int_{\partial\Omega_c(t)} F\ dS = \int_{\Omega_c(t)} \dot{W}\ dV, \tag{5.24}$$

where the conservative state variable, Q, inviscid flux function, F, and chemistry source term, $\dot{W}$, are given by

$$Q = \begin{bmatrix} \rho_1 \\ \vdots \\ \rho_s \\ \vdots \\ \rho_{NS} \\ \rho\tilde{u} \\ \rho e_0 \end{bmatrix},\quad F = \begin{bmatrix} \rho_1(\tilde{u}-\tilde{u}_\Omega)\cdot\tilde{n} \\ \vdots \\ \rho_s(\tilde{u}-\tilde{u}_\Omega)\cdot\tilde{n} \\ \vdots \\ \rho_{NS}(\tilde{u}-\tilde{u}_\Omega)\cdot\tilde{n} \\ (\rho\tilde{u}(\tilde{u}-\tilde{u}_\Omega)+p\tilde{\tilde{I}})\cdot\tilde{n} \\ [\rho e_0(\tilde{u}-\tilde{u}_\Omega)+p\tilde{u}]\cdot\tilde{n} \end{bmatrix}, \text{and}\quad \dot{W} = \begin{bmatrix} \dot{w}_1 \\ \vdots \\ \dot{w}_s \\ \vdots \\ \dot{w}_{NS} \\ 0 \\ 0 \end{bmatrix}. \tag{5.25}$$

5.2.1 Numerical Approximations to Spatial Integrals

The numerical integration of equation (5.24) begins with approximations to volume and surface integrals. For the volume integrals a second order midpoint rule is used. For example, the numerical integration of Q for using the midpoint rule is expressed as

$$\int_{\Omega_c(t)} Q(\tilde{x}, t) dV = Q_c(t)\mathcal{V}_c(t), \tag{5.26}$$

where $Q_c(t)$ is the value of Q at the cell's centroid, and $\mathcal{V}_c(t)$ is defined by

$$\mathcal{V}_c(t) = \int_{\Omega_c(t)} dV. \tag{5.27}$$

The numerical integration of the surface integral in equation (5.24) is accomplished by summing the contributions of each of the NF faces of cell c. Each individual contribution is again approximated using the midpoint rule. The flux function itself will require additional numerical treatments that will be discussed in later sections. For now, assume that the flux can be approximated by a function, $\hat{F}(Q_l, Q_r)$, of conservative values to the left and right of the face. Given this, the numerical integration of F is given by the equation

$$\int_{\partial\Omega_c(t)} F dS = \sum_{f=1}^{NF_c} \int_{\partial\Omega_{c,f}(t)} F dS \approx \sum_{f=1}^{NF_c} \mathcal{A}_{c,f}(t)\hat{F}(Q_{l,f}, Q_{r,f}), \tag{5.28}$$

where the area of the face, $\mathcal{A}_{c,f}(t)$, is defined as

$$\mathcal{A}_{c,f}(t) = \int_{\partial\Omega_{c,f}(t)} dS. \tag{5.29}$$

At this point equation (5.24) is numerically approximated by the equation

$$\frac{d}{dt}\left[\mathcal{V}_c(t)Q_c(t)\right] + \sum_{f=1}^{NF_c} \mathcal{A}_{c,f}(t)\hat{F}(Q_{l,f}, Q_{r,f}) = \mathcal{V}_c(t)\dot{W}_c(t). \tag{5.30}$$

Notice that the differential term that remains in this equation applies to the product of volume and conservative state vector for the cell. However, the variable that is the objective of these calculations is $Q_c(t)$, not $\mathcal{V}_c(t)Q_c(t)$. This problem is solved by applying the chain rule, as in

$$\frac{d}{dt}Q_c(t)\mathcal{V}_c(t) = Q_c(t)\frac{d}{dt}\mathcal{V}_c(t) + \mathcal{V}_c(t)\frac{d}{dt}Q_c(t). \tag{5.31}$$

The derivative of volume with respect to time can be converted into a spatial integral through the use of identity (5.21):

$$\begin{aligned} Q_c \frac{d}{dt}\mathcal{V}_c(t) &= Q_c \frac{d}{dt}\int_{\Omega_c(t)} dV \\ &= Q_c \int_{\partial\Omega_c(t)} \tilde{u}_\Omega \cdot \tilde{n} dS \\ &\approx Q_c \sum_{f=1}^{NF_c} \mathcal{A}_{c,f}(t)(\tilde{u}_{\Omega,f} \cdot \tilde{n}_{c,f}). \end{aligned} \tag{5.32}$$

This term, known as the geometric conservation law[42], is necessary for correct time integration when mesh deformation is present. Given this, the solution method can now be described in terms of a system of ordinary differential equations of the form

$$\frac{d}{dt}Q_c = R_c, \tag{5.33}$$

where R_c is given by the expression

$$R_c = \frac{1}{\mathcal{V}_c}\left[\mathcal{V}_c \dot{W}_c - \sum_{f=1}^{NF_c} \mathcal{A}_{c,f}\hat{F}(Q_{l,f}, Q_{r,f}) - Q_c \sum_{f=1}^{NF_c} \mathcal{A}_{c,f}(\tilde{u}_{\Omega,f} \cdot \tilde{n}_{c,f})\right]. \tag{5.34}$$

Equations (5.33) and (5.34) describe a system of ordinary differential equations that numerically model the time evolution of the fluid dynamics equation when simultaneously satisfied for all cells in the mesh. To represent this fact, the cell subscript c will be dropped. Thus Q_c represents the fluid state for cell c, while Q represents the fluid states of all cells in the mesh. For example, while equation (5.33) represents the cell by cell differential equations, the global system of equations is given by

$$\frac{d}{dt}Q(t) = R(Q(t), t). \tag{5.35}$$

5.2.2 Numerical Approximations to Temporal Integrals

Numerical integration of the system of equations in (5.35) can be performed by a variety of methods. For example, a Runge-Kutta scheme can be applied to obtain a solution to this set of equations. For this implementation, a family of schemes as described by Janus [43] is used.

These schemes are given by the time discretized equation

$$\frac{(1+\psi)\Delta Q^n - \psi\Delta Q^{n-1}}{\Delta t} = (1-\theta)R^n(Q^n) + \theta R^{n+1}(Q^{n+1}), \tag{5.36}$$

where

$$\Delta Q^n = Q^{n+1} - Q^n.$$

Equation (5.36) can represent a set of implicit time integration schemes, including the second order three point backward ($\theta = 1$, $\psi = \frac{1}{2}$), backward Euler ($\theta = 1$, $\psi = 0$), and Crank-Nicholson ($\theta = \frac{1}{2}$, $\psi = 0$). It also includes explicit schemes such as forward Euler ($\theta = 0$,$\psi = 0$).

The solution of equation (5.36) for Q^{n+1} given Q^n is the goal of the time integration procedure. However, the solution is complicated by the fact that $R^{n+1}(Q^{n+1})$ is a non-linear function with a non-trivial inverse. Instead of solving equation (5.36) directly, a common approach is to employ the Newton iterative method for the solution of the non-linear homogeneous equation given by

$$\mathcal{L}(Q^{n+1}) = Q^{n+1} - Q^n - \frac{\Delta t}{1+\psi}\left[(1-\theta)R^n(Q^n) + \theta R^{n+1}(Q^{n+1})\right] - \frac{\psi}{1+\psi}(Q^n - Q^{n-1}) = 0. \tag{5.37}$$

The vector form of the Newton method is used to obtain the zero of the vector valued function, $\mathcal{L}(Q)$. Consequently, the Newton method should converge to the vector value of Q^{n+1}. The Newton method proceeds by iteratively solving the equation

$$\mathcal{L}'(Q^{n+1,p})(Q^{n+1,p+1} - Q^{n+1,p}) = -\mathcal{L}(Q^{n+1,p}), \quad p \geq 0, \tag{5.38}$$

where the Newton iteration is initialized using the previous time-step values, thus $Q^{n+1,p=0} = Q^n$. Assuming $\mathcal{V}$, $\mathcal{A}$, and u_Ω are not functions of Q, then the Jacobian, $\mathcal{L}'(Q^{n+1,p})$, is given by the equation

$$\mathcal{L}'(Q^{n+1,p}) = I - \frac{\theta \Delta t}{1+\psi}\left[\frac{\partial}{\partial Q} R^{n+1}(Q)\right]$$
$$= I - \frac{\theta \Delta t}{(1+\psi)}\left[\frac{\partial \dot{W}}{\partial Q} - \sum_{f \in faces} \frac{\mathcal{A}_f^{n+1}}{\mathcal{V}^{n+1}} \frac{\partial \hat{F}(Q_{l,f}, Q_{r,f})}{\partial Q} - I \sum_{f \in faces} \frac{\mathcal{A}_f^{n+1}}{\mathcal{V}^{n+1}} (\tilde{u}_{\Omega,f} \cdot \tilde{n}_f)\right]. \tag{5.39}$$

The Jacobian in equation (5.39) is a sparse matrix with dense sub-blocks. The off-diagonal blocks are terms generated by the flux Jacobians, due to their functional dependence on neighboring cells. The assembly of the matrix and the solution of the resulting linear system of equations is covered in later sections.

5.2.3 Numerical Flux Algorithms

Care must be taken in the numerical treatment of the inviscid flux function F of equation (5.24) in order to avoid oscillatory behavior around solution discontinuities such as shocks. One approach to removing these numerically generated oscillations is to add dissipation to the numerical fluxes that damp high frequency signals (of the order of the mesh size). Unfortunately, there is a tradeoff between this damping and the accurate resolution of discontinuities. Although this is an area of current research, it is generally agreed that characteristic based methods are better suited to resolve solution discontinuities. Specifically, the characteristic based flux difference split schemes are chosen for the numerical treatment of inviscid fluxes. These schemes are formulated as a two step process. First, the solution variables are extrapolated from cell centers to the left and right sides of faces. Then the flux at the face is computed based on the left and right states in an attempt to resolve the correct discontinuous behavior. Many of these flux formulations involve finding an approximate solution to the Riemann problem, and thus are termed approximate Riemann methods. The two methods presented here are Roe's flux and the HLLE flux.

For the flux functions presented here it is assumed that the local face coordinates are oriented with respect to the left and right sides of the face: left is the negative coordinate direction. The variable u_f is the velocity of the fluid with respect to the face coordinates and is given by

$$u_f = (\tilde{u} - \tilde{u}_\Omega) \cdot \tilde{n}. \tag{5.40}$$

5.2.3.1 Roe Averaged Fluxes

The Roe averaged flux formulation is based on a linearization of the Riemann problem The form suitable for chemically reacting flow problems has been developed by Cinnella [44]. The equations described here are presented in [45] as well.

In the following equations, the jump in a variable f from the left to right states will be denoted as

$$[\![f]\!] = f_r - f_l, \tag{5.41}$$

while the average of the left and right values is represented as

$$\langle f \rangle = \frac{f_l + f_r}{2}. \tag{5.42}$$

Given this notation, the Roe flux is written as

$$\hat{F}(Q_l, Q_r) = \langle F \rangle - \frac{1}{2}([\![\hat{F}]\!]_A + [\![\hat{F}]\!]_B + [\![\hat{F}]\!]_C), \tag{5.43}$$

where the averaged function, F, is the function described in equation (5.24), while $\hat{F}$ represents the flux of the Roe approximate Riemann solver.

The $[\![\hat{F}]\!]_A$ term is given by

$$[\![\hat{F}]\!]_A = \left([\![\rho]\!] - \frac{[\![p]\!]}{\hat{a}^2}\right) |\hat{\lambda}_A| \begin{bmatrix} \hat{\rho}_1 \\ \vdots \\ \hat{\rho}_s \\ \vdots \\ \hat{\rho}_{NS} \\ \hat{\tilde{u}} \\ \hat{h}_0 - \hat{a}^2/(\hat{\gamma} - 1) \end{bmatrix} + \hat{\rho}|\hat{\lambda}_A| \begin{bmatrix} [\![\rho_1/\rho]\!] \\ \vdots \\ [\![\rho_s/\rho]\!] \\ \vdots \\ [\![\rho_{NS}/\rho]\!] \\ [\![\tilde{u}]\!] - [\![u_f]\!]\tilde{n} \\ \Theta \end{bmatrix}, \tag{5.44}$$

where

$$\Theta = \hat{\tilde{u}} \cdot [\![\tilde{u}]\!] - \hat{u}_f [\![u_f]\!] - \sum_{s=1}^{NS} \hat{\psi}_s \left[\!\!\left[\frac{\rho_s}{\rho}\right]\!\!\right]. \tag{5.45}$$

Similarly, the terms associated with $[\![F]\!]_B$ and $[\![F]\!]_C$ are given by

$$[\![\hat{F}]\!]_{B,C} = \frac{1}{2\hat{a}^2}\left([\![p]\!] \pm \hat{\rho}\hat{a}[\![u_f]\!]\right)|\hat{\lambda}_{B,C}| \begin{bmatrix} \hat{\rho}_1 \\ \vdots \\ \hat{\rho}_s \\ \vdots \\ \hat{\rho_{NS}} \\ \hat{\tilde{u}} \pm \hat{a}\tilde{n} \\ \hat{h}_0 \pm \hat{u}_f \hat{a} \end{bmatrix}. \tag{5.46}$$

The Roe averaged values for the flow variables are given as

$$\hat{\rho} = \sqrt{\rho_l \rho_r}, \quad \hat{u}_f = \frac{\langle u_f \sqrt{\rho}\rangle}{\langle \sqrt{\rho}\rangle} \quad \hat{\tilde{u}} = \frac{\langle \tilde{u}\sqrt{\rho}\rangle}{\langle \sqrt{\rho}\rangle}, \quad \hat{q}^2 = \hat{\tilde{u}} \cdot \hat{\tilde{u}}, \quad \hat{h}_0 = \frac{\langle h_0 \sqrt{\rho}\rangle}{\langle \sqrt{\rho}\rangle}, \quad \hat{T} = \frac{\langle T\sqrt{\rho}\rangle}{\langle \sqrt{\rho}\rangle},$$
$$\hat{\rho}_s = \frac{\langle (\rho_s/\rho)\sqrt{(\rho)}\rangle}{\langle \sqrt{\rho}\rangle}, \quad \hat{R} = \sum_{s=1}^{NS} \hat{\rho}_s R_s, \quad \hat{c}_v^* = \sum_{s=1}^{NS} \hat{\rho}_s c_{vs}^*, \quad \hat{c}_{vs}^* = \frac{1}{[\![T]\!]} \int_{T_l}^{T_r} c_{vs} dT,$$
$$\hat{\gamma} - 1 = \frac{\hat{R}}{\hat{c}_v^*}, \quad \hat{e}_s = \frac{\langle e_s \sqrt{\rho}\rangle}{\langle \sqrt{\rho}\rangle}, \quad \hat{\psi}_s = \frac{R_s \hat{T}}{\hat{\gamma} - 1} - \hat{e}_s + \frac{\hat{q}^2}{2}.$$

The Roe averaged speed of sound, $\hat{a}$, is given by the equation

$$\hat{a}^2 = (\hat{\gamma} - 1)\left[\hat{h}_0 - \frac{\hat{q}^2}{2} + \hat{c}_v^* \hat{T} - \sum_{s=1}^{NS} \hat{\rho}_s \hat{e}_s\right]. \tag{5.47}$$

Similarly, the eigenvalues of the Roe matrix are given as

$$\hat{\lambda}_A = \hat{u}_f, \quad \hat{\lambda}_{B,C} = \hat{u}_f \pm \hat{a}. \tag{5.48}$$

Recall that the solution to the inviscid equations is actually the limit of the Navier-Stokes equations as viscosity tends to zero. If care is not taken, it is possible for numerical methods to admit solutions that are not actually the limit of the Navier-Stokes equations. For inviscid fluid flows, non-physical expansion "shocks" can develop in the flow that are not limit solutions to the Navier-Stokes equations. In particular, the Roe scheme is susceptible to such problems. Solutions to this problem involve increasing the dissipation of the scheme when conditions of expansion discontinuities are present. The particular correction that is used for this numerical formulation is developed by Harten and Hyman[46]. See Leveque [47], pages 152–153, for a general overview of the method. The basic approach adds a correction to the eigenvalues when an expansion is present, that is when $\lambda_l < 0$ and $\lambda_r > 0$. The correction, which adds sufficient diffusion to remove the expansion discontinuity, is computed by the equation

$$|\hat{\lambda}|_{fix} = |\hat{\lambda}| + \frac{\lambda_l(|\hat{\lambda}| - \lambda_r) + \lambda_r(|\hat{\lambda}| - \lambda_l)}{\lambda_r - \lambda_l}. \tag{5.49}$$

5.2.3.2 HLLE Flux

In some cases, the Roe averaged fluxes can introduce non-physical instabilities in the flow solutions. For example, in very high speed (*e.g.* Mach 10) blunt body problems, the Roe scheme can produce a carbuncle phenomena (a non-physical bump) in the forward region of the bow shock[48]. In addition, the Roe scheme can exhibit unstable behavior in strong planar shock fronts that align with the mesh, such as the Mach disk in an over-expanded rocket nozzle plume. These problems appear to be related to multi-dimensional effects, whereby the Roe scheme produces insufficient numerical dissipation in the faces that are orthogonal to strong shocks. Several schemes do not have these problems, for example the approximate Riemann flux of Eindfelt. Eindfelt's HLLE flux[49] is able to handle many of these cases, and can resolve many shock structures to within a few cells, much like the Roe scheme. However, the HLLE flux is more diffusive and tends to smear other structures of the flow. The HLLE flux is implemented for finite-rate chemistry by the flux equation

$$\hat{F}(Q_l, Q_r) = \frac{b^{(+)}F(Q_l) - b^{(-)}F(Q_r)}{b^{(+)} - b^{(-)}} + \frac{b^{(+)}b^{(-)}}{b^{(+)} - b^{(-)}}(Q_r - Q_l), \tag{5.50}$$

where the terms $b^{(+)}$ and $b^{(-)}$ are defined using the Roe eigenvalues $\hat{\lambda}_B$ and $\hat{\lambda}_C$:

$$b^{(+)} = \max(0, (u_f + a)_r, \hat{u}_f + \hat{a}), \text{ and} \tag{5.51}$$

$$b^{(-)} = \min(0, (u_f - a)_l, \hat{u}_f - \hat{a}). \tag{5.52}$$

5.2.3.3 Adaptive Riemann Solver

There is a tradeoff between the Roe and HLLE flux formulations. While the Roe flux is less diffusive and generally produces more accurate solutions, it is not robust. On the other hand, the HLLE flux is robust, but less accurate. One approach that offers a compromise is the adaptive Riemann solver proposed by Quirk[48]. In this approach, faces that are resolving a sufficiently strong shock are identified. Then any cell that has a face resolving a strong shock switches all of its fluxes to the HLLE scheme. In Quirk's approach, a strong shock is detected when a face has a sufficient pressure jump from the left to right sides. The criterion is given by the equation

$$\frac{|p_r - p_l|}{\min(p_l, p_r)} > \alpha, \tag{5.53}$$

where α is a problem specific parameter controlling the detection of strong shocks within the problem. The advantage of this adaptive Riemann solver approach is that it preserves the high accuracy of the Roe scheme while fixing some of its robustness problems in regions of strong shocks.

5.2.3.4 Extrapolating Left and Right States at a Face

The approximate Riemann fluxes described in the previous sections require left and right states. For first order formulations these values simply correspond to the left and right cell values that correspond to either sides of the face. For higher order schemes, these left and right states are extrapolated from neighboring cell values. The approach that is used in the present study is applicable to meshes composed of hexahedral cells. The primitive variables (pressure, density and velocity) are extrapolated to the faces using the Monotone Upstream-centered Schemes for Conservation Laws (MUSCL) extrapolation approach[50] as illustrated in figure 5.1. The extrapolated values are computed using the relationships

$$
\begin{aligned}
q_l &= q_{cl} + \frac{1}{2}\Psi(r_l)(q_{cl} - q_{cl,cl}), \text{ and} \\
q_r &= q_{cr} + \frac{1}{2}\Psi(r_r)(q_{cr} - q_{cr,cr}),
\end{aligned} \tag{5.54}
$$

where $\Psi(r)$ is a limiter function, with r defined as

$$
\begin{aligned}
r_l &= \frac{(q_{cr} - q_{cl})}{(q_{cl} - q_{cl,cl})}, \text{ and} \\
r_r &= \frac{(q_{cl} - q_{cr})}{(q_{cr} - q_{cr,cr})}.
\end{aligned} \tag{5.55}
$$

There are a variety of limiter functions that are described in numerous texts on numerical methods for conservation law equations. For more details refer to Hirsch [51] pages 552–556. Van Leer's limiter[52] was chosen for the simulations presented in this study.

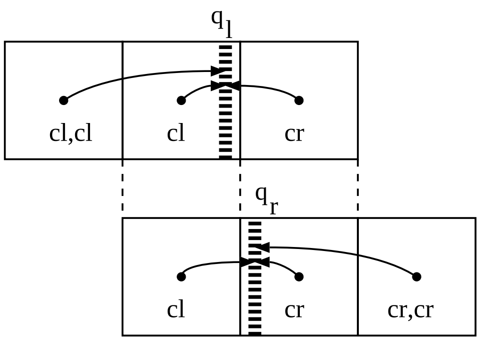


Figure 5.1: MUSCL Extrapolation Variable Dependencies

In general, the MUSCL extrapolation scheme described here is only applicable to meshes consisting of hexahedra. For more generalized meshes the extrapolation can be based on properly limited cell gradients. For example, the limiters of Barth [53] or Venkatakrishnan [54] may be applied.

5.2.4 Linear System Solution

The solution of the Newton iteration step given by equation (5.38) requires solving a linear system of equations of the form

$$Ax = b, \tag{5.56}$$

where

$$\begin{aligned} A &= \mathcal{L}'(Q^{n+1,p}), \\ x &= (Q^{n+1,p+1} - Q^{n+1,p}), \\ b &= -\mathcal{L}(Q^{n+1,p}). \end{aligned}$$

The matrix, A, of this linear system is given by equation (5.39) and is typically a matrix with sparse structure that is composed of dense sub-blocks. Most of the terms of equation (5.39) contribute to the diagonal block of the matrix, while the flux terms, being a function of left and right Q values, contribute to off diagonal terms. Flux Jacobian terms are treated as Jacobians of the first order flux functions in an effort to increase the sparseness of matrix A. The matrix A can be factored into a lower, upper, and diagonal blocks, as in

$$A = L + D + U. \tag{5.57}$$

For first order flux Jacobians, each internal face of the mesh contributes a term to the diagonals of the cells to either side, while also contributing one block to the lower matrix, L, and one block to the upper matrix U. The system of equations is solved using a symmetric Gauss-Seidel method. Iterative methods of this form are rather efficient at solving systems of equations produced by finite volume schemes applied to hyperbolic equations such as the inviscid fluid dynamics equations presented here. The symmetric Gauss-Seidel iterative solver works by a two pass method. These two passes, a forward and a backward pass, are a consequence of the solution of the factored equations

$$(L+D)x^{*\,i+1} + Ux^{i} = b, \text{ and}$$
$$Lx^{*\,i+1} + (D+U)x^{i+1} = b, \tag{5.58}$$

where $x^{*\,i+1}$ is the result of the forward pass of the symmetric Gauss-Seidel iteration. The iteration is initialized with the first pass of a block Jacobi iterative method given by

$$x^{0} = D^{-1}b. \tag{5.59}$$

In the solution of equations (5.58) and (5.59), a dense GAXPY (general A x plus y) LU method[55] is employed to invert the diagonal blocks of D. This LU factorization is performed once before the Gauss-Seidel iteration proceeds and is used in both passes.

CHAPTER VI

RULE SPECIFICATIONS FOR THE INVISCID FINITE-RATE CHEMISTRY SOLVER

Chapter III introduced a specification language for numerical computational methods, while chapter V developed a model for inviscid flows that include finite-rate chemistry. This chapter will demonstrate how the grid is specified as a set of facts, and how the components of the solver derived in chapter V can be specified using the constructs described in chapter III.

6.1 Representing the Numerical Mesh

The discretization of space represented by a numerical mesh is one of the more fundamental data structures required by finite volume, finite element, and finite difference schemes. In general, a spatial discretization consists of a collections of points that have spatial positions distributed over the region of interest. In addition to these spatial points, there are edges, faces, and cells[1]. These components are defined with respect to their relationship to one another. For example, an edge is defined by two points in the mesh, while a face is defined by a set of edges, and a cell is defined by a set of faces, as illustrated in figure 6.1. There are many links between these entities, such as the relationship between faces and the cells on either side, or the nodes that form a cell. The specific relationships that are required depend on the numerical method being employed and how it is formulated. For example, a finite difference method may be edge-based, while a finite volume method may be cell- and face-based. In addition, the ordering of relationships between these mesh entities may have topological significance, such as the correspondence of node orderings to face normal vectors, as shown in figure 6.2. All of these issues must be defined in the fact database, where the particular relationships that are required depend on the solution algorithm methodology and solver design decisions (such as the decision to use a point-wise or reduction rule).

[1]In the finite element community, cells are often called elements or zones.

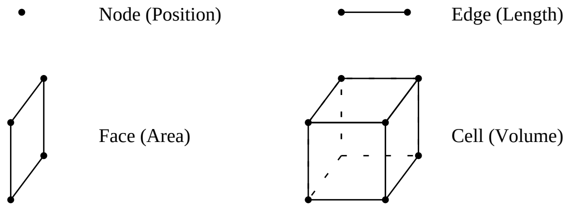


Figure 6.1: Space Discretization Entities

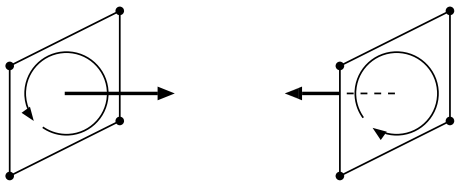


Figure 6.2: Relationship Between Node Ordering and Face Orientation

The numerical algorithm developed in chapter V reduces to a set of volume and surface integrals, where volume integrals are computed for each cell based on cell values, and surface integrals are computed based on extrapolated values for the left and right sides. Thus a face based data structure would appear to be more natural. In addition, the area computation of a face requires access to nodal positions. Since the current computational methods are only considered for hexahedral cells, only faces described by a four node circuit are required. It is assumed that this nodal mapping will be stored in a map identified as `qnds`. Nodes are ordered in the mapping `qnds` such that the normal vector points in the direction of the right side of the face, as illustrated in figure 6.2. The maps to a face's left and right cells are provided by maps `cl`, `cr`, `cll`, and `crr`, consistent with the MUSCL extrapolation scheme given in equation (5.54). In addition, it is assumed that ghost cells will be created such that boundary faces also have a left and right cell. Finally, some cell based calculations require access to the faces of a given cell, and so six mapping to the six faces of the hexahedra are also provided. Table 6.1 summarizes the facts sufficient to describe the mesh for the finite volume algorithm derived earlier.

In addition to these facts, rules for the computation of mesh-related properties such as face areas and cell volumes are required. The rule signatures for these grid-related computations are listed in table 6.2.

Table 6.1: Mesh Data Descriptions

Name	Value Type	Role	Domain
`pos`	3-D vector	position vector	nodes
`qnds`	map	maps to 4 nodes of a quadrilateral face	faces
`cl`	map	maps to left cell of a face	faces
`cr`	map	maps to right cell of a face	faces
`cll`	map	maps to left higher order cell	faces
`crr`	map	maps to right higher order cell	faces
`north`	map	north face of cell	cells
`south`	map	south face of cell	cells
`east`	map	east face of cell	cells
`west`	map	west face of cell	cells
`top`	map	top face of cell	cells
`bottom`	map	bottom face of cell	cells

Table 6.2: Grid Related Rule Signatures

Rule	Description
$area \leftarrow qnds \rightarrow pos$	Compute area vector
$fpos \leftarrow qnds \rightarrow pos$	Compute face center
$vol \leftarrow (north, south, east, west, top, bottom) \rightarrow (area, fpos)$	Compute cell volume

6.2 Rules for General Chemistry Models

The evaluation of the chemistry and thermodynamic models requires an initialization phase, where information is imported from a chemistry database. The particular information that is imported depends on an input parameter (a text string) that describes which chemistry model to apply. There are two basic functions that are required in this evaluation: an equation of state that solves relationships between various thermodynamic state variables, and a reaction rate calculator. These two calculators are identified by the variables `eos` and `reactor`. In addition, there is a simple calculator, named `qvi`, which describes the layout of information in the conservative state vector (`qvi` is short for q-vector information). There is a singleton rule that transforms the model specified by the user into these calculators by initializing them with the appropriate chemistry database information. The equation of state is solved for a particular conservative q-vector by a rule that creates the `eos_state` variable, which contains the thermodynamic state information obtained by solving the state equations. In addition, a rule that computes the reaction rates, Kf, and the equilibrium constant, Kc, as identified by the variable `rates` is required for the computation of the chemistry source terms. These rules signatures, which describe the computation of the chemistry terms of the equations, are summarized in table 6.3.

Table 6.3: Chemistry Related Rule Signatures

Rule	Rule Type	Description
$qvi, eos, reactor \leftarrow model$	singleton	Create model calculators
$eos\_state \leftarrow qvi, eos, q$	point-wise	Evaluate EOS
$rates \leftarrow eos, eos\_state, reactor$	point-wise	Evaluate reaction rates

6.3 Flux Computation and Space Integration Rules

The evaluation of spatial integrals (neglecting the geometric conservation term) provide the residual equation (5.34). These spatial integrals are collectively identified as a source term variable, defined by

$$src = \mathcal{V}\dot{W} - \sum_{f=1}^{NF} \mathcal{A}_f \hat{F}(Q_{l,f}, Q_{r,f}). \tag{6.1}$$

Since additional terms (such as viscous effects) will require modifying equation (6.1) by adding additional terms, it would make sense to treat this calculation with the unit/apply rule semantics. Using this approach, new terms can be added to the `src` terms as needed, by adding another apply rule to the rule database. Since these rules would require additional information (such as transport properties), they would not be applied unless the user supplied the necessary parameters.

The first term of equation (6.1) requires an evaluation of cell volumes and the species production rates given by equation (5.13), while the second term requires a summation of fluxes about the faces of a cell. While the first term is a function of local cell properties, the second is a function of local and neighboring cell properties. This increased dependence gives rise to several options regarding the flux integration method: 1) the flux integration can be treated as a local sum performed for cells, or 2) the flux integration can be treated from a face perspective where the flux contributions are added from faces to cells. The cell centered computation can be performed using the cell to face mappings of table 6.1. While this is a convenient approach, it has the disadvantage that it does not easily generalize to arbitrary cell types. On the other hand, the face-based approach makes no assumptions about cell shape and so is a much more general approach. In the face-based approach, the integration of the flux term is divided into two rules: one for summing to the left face, and the other for summing to the right. In the application of this rule, care is taken to follow the conventions of Green's theorem that assumes outward pointing normals. Thus the left cell is consistent with this formulation, while the right cell requires a sign change in the normal vector (see figure 6.2). Table 6.4 lists a summary of the rules required to compute equation (6.1).

Table 6.4: Space Integral Rule Signatures

Rule	Rule Type	Description
$qp \leftarrow qvi, eos, eos\_state, q$	point-wise	primitive q vector
$qp\_l \leftarrow (cl, cr, cll) \rightarrow qp$	point-wise	left MUSCL state
$qp\_r \leftarrow (cl, cr, crr) \rightarrow qp$	point-wise	right MUSCL state
$qf \leftarrow area, eos, qp\_l, qp\_r$	point-wise	Roe flux
$src \leftarrow CONSTRAINT(vol)$	unit	initialize src = 0
$cl \rightarrow src \leftarrow qf$	apply	left cell flux term
$cr \rightarrow src \leftarrow qf$	apply	right cell flux term
$src \leftarrow eos, reactor, eos\_state, q, rates, vol$	apply	chemistry source terms

6.4 Assembling the Jacobian Matrix

The Jacobian used in the Newton method, described by equation (5.39), consists of a diagonal block matrix combined with off-diagonal matrices formed from the differentiation of the flux function. If the flux function, $\hat{F}(Q_{left}, Q_{right})$, is defined by the conservative variables on the left and right side of the face, then the Jacobian of this function will consist of two flux Jacobian matrices, given by

$$\begin{aligned} fjp &= \mathcal{A}^{n+1}\frac{\partial \hat{F}(Q_{left}, Q_{right})}{\partial Q_{left}}, \\ fjm &= \mathcal{A}^{n+1}\frac{\partial \hat{F}(Q_{left}, Q_{right})}{\partial Q_{right}}. \end{aligned} \tag{6.2}$$

The variables, `fjp` and `fjm`, are Jacobians of the flux functions located at faces and are components of the overall Jacobian matrix given in equation (5.39). The diagonal blocks of the Jacobian matrix can be constructed using a formulation parallel to the definition of the *src* given in equation (6.1). Using this example, the diagonal contributions from *src* are given by

$$(srcJ)_c = \mathcal{V}^{n+1}\frac{\partial W_c}{\partial Q_c} - \sum_{f\rightarrow left=c}(fjp)_f + \sum_{f\rightarrow right=c}(fjm)_f. \tag{6.3}$$

From this, the diagonal blocks of the Jacobian are formed by computing

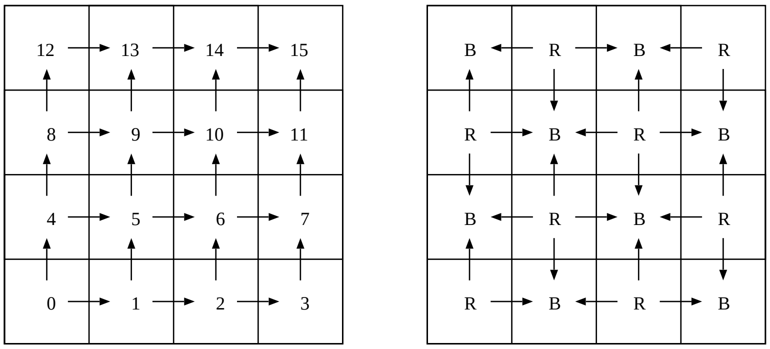
$$D_c = I - \frac{\theta\Delta t}{\mathcal{V}^{n+1}(1+\psi)}\left[(srcJ)_c - I\sum_{f\in faces}\mathcal{A}_f^{n+1}(\tilde{u}_{\Omega,f}\cdot\tilde{n}_f)\right]. \tag{6.4}$$

While `D` of equation (6.4) describes the diagonal of the Jacobian, the off-diagonal terms are formed from the `fjp` and `fjm` terms. While a face contributes `fjp` to the left face diagonal, it also contributes `fjp` to the right face's row equation. The `fjm` Jacobian contributes an off-diagonal term in a similar fashion. As a result the orientation of the left and right sides of faces and the matrix structure are related. Since the local orientation of the face normal is an arbitrary choice, it is possible to select the orientation of faces such that the Jacobian matrix always has `fjm` terms in its upper diagonal and `fjp` terms in its lower diagonal.

The arrangement of equations in the Jacobian matrix dictates the placement of off-diagonal blocks into either the lower or upper portion of the matrix. For dense matrices, each ordering of equations uniquely specifies which matrix elements exist in the upper and lower portions of the matrix. In sparse matrices, however, families of orderings can correspond with the same upper and lower partition of off-diagonal terms. These partial orderings of matrix equations are described using colorings of the matrix connectivity graph[2]. Once the equations (cells) of the matrix are colored, then one equation occurs before another equation if its color is less than the other. Thus if the face left-right orientation corresponds to the matrix coloring, then there will be a corresponding relationship between the location of the `fjp` and `fjm` terms. For example, if the faces are constructed such that the left side always faces the lowest color cell as illustrated in figure 6.3, then the matrix will be formed such that the lower blocks of the matrix only contain `fjp` terms, while the upper term of the matrix only contains `fjm` terms. Once the lower and upper portions of the matrix are identified, it is possible to form a row based data structure for the matrix based on the face connectivities. This is accomplished by transposing the face to cell

[2]The non-zero entries in the sparse matrix represent an edge in the matrix connectivity graph such that a non-zero term at A_{ij} represents an edge from equation i to equation j.

mappings `cl` and `cr` to obtain the cell to face mappings `lower` and `upper`, as illustrated in figure 6.4. Note, that while only one cell may be contained in the `cl` map for a face, many faces may be contained in the `lower` map for a cell.



Canonical Coloring

Red-Black Coloring

Figure 6.3: Mesh Colorings

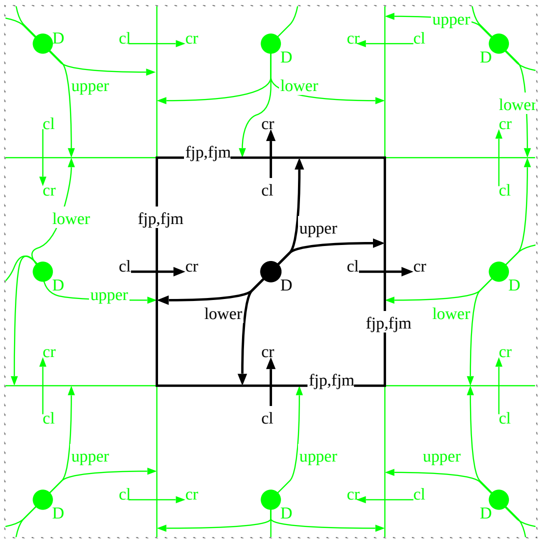


Figure 6.4: Cell Perspective of Matrix Assembly

The final matrix assembly using the `cl`, `cr`, `upper`, and `lower` maps is illustrated in figure 6.5. Notice that the data structure is a row oriented data structure. One can move from cells to blocks that are on the same row and then from those off-diagonal blocks to their corresponding column. This data structure is all that is needed to describe the Gauss-Seidel iterative algorithm given in equation (5.58).

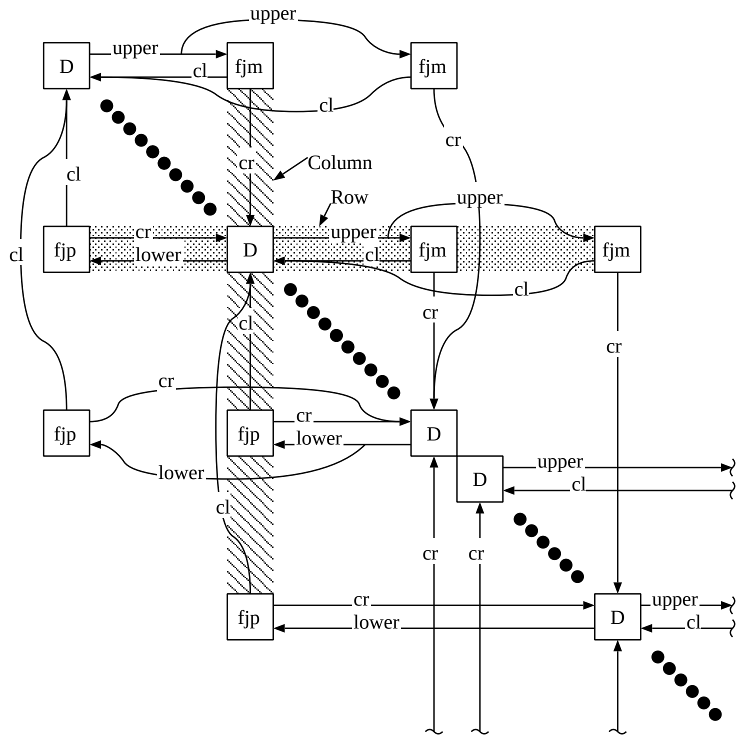


Figure 6.5: Matrix Assembly Data Structure Cut-Away

6.5 Performing a Gauss-Seidel Iteration

The solution of the linear system using the symmetric Gauss-Seidel algorithm described by equation (5.58) can be obtained by using a recursive rule specification. For this discussion only the forward pass of the symmetric Gauss-Seidel algorithm will be illustrated. This algorithm evaluates the expression $x^{k+1} = (L+D)^{-1}(Ux^k + b)$, where $b = \mathcal{L}(Q^{n+1,p})$, and L, D, and U are the lower, diagonal, and upper components of the Jacobian matrix, respectively. This method can be written as the point-wise recurrence relation

$$x_i^{k+1} = D_i^{-1}\left(b_i - \sum_{j=1}^{i-1} L_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} U_{ij}x_j^k\right). \tag{6.5}$$

For the data structure described in figure 6.5, equation (6.5) can be rewritten as

$$x_i^{k+1} = D_i^{-1}\left[b_i + \frac{\theta\Delta t}{\mathcal{V}^{n+1}(1+\psi)}\left(\sum_{(i,j)\in lower}(fjp)_j x_{cl_j}^{k+1} - \sum_{(i,j)\in upper}(fjm)_j x_{cr_j}^{k}\right)\right]. \qquad (6.6)$$

Note that in this equation there is a sign change for the `fjm` term, which compensates for the sign change of the face normal, consistently with Green's theorem. This recurrence relation can be represented by the recursive rule specification, given by the signature

$$x^{k+1} \leftarrow D, b, lower \rightarrow fjp, lower \rightarrow cl \rightarrow x^{k+1}, upper \rightarrow fjm, upper \rightarrow cr \rightarrow x^k. \qquad (6.7)$$

The specification given by equation (6.7) provides sufficient information to generate a schedule for the forward pass of the symmetric Gauss-Seidel algorithm. For example, consider how the χ function might be evaluated for this recursive specification for a red-black coloring, as illustrated in figure 6.3. In this case all red cells have only left faces pointing to them, while all black cells have only right faces pointing to them. Thus red cells have a null mapping for `lower`, while black cells have a null mapping for `upper`. In the first step of the iterative algorithm evaluating χ, only the red cells will be evaluated, since the lower map is null only for these cells and no entities have the attribute `x{k+1}`. For the second iteration of this scheduling process, all black cells can be scheduled since the red cells produced the necessary `x{k+1}` attribute. A following iteration will determine that no new attributes can be assigned and the schedule of the Gauss-Seidel algorithm will be complete. A two step concurrent red-black algorithm will be the result. Interestingly, if instead a regular mesh is given with a canonical coloring, then the derived schedule will match the concurrent hyper-plane algorithm used in the NAS parallel benchmark LU[56], as illustrated in figure 6.6.

6.6 Summary

This chapter provided an overview of the specification required to build a complex numerical application. The data structures required to represent a finite volume discretized mesh for an implicit solver are shown. These data structures are consistent with the fact data types described in chapter III. In addition, a recursive rule specification is used to describe a general Gauss-

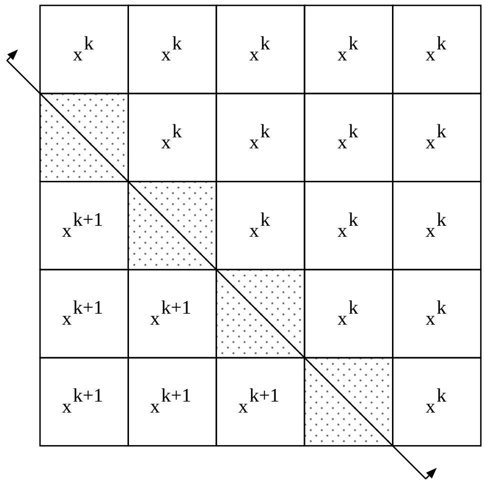


Figure 6.6: Execution Pattern for the Concurrent Hyper-plane Algorithm

Seidel algorithm. This specification illustrates the ability of the concurrent schedule generation algorithm described in chapter IV to provide automatic generation of concurrent programs, based on high level specifications.

CHAPTER VII

RESULTS

This chapter is divided into two main sections. The first one features a series of test cases, demonstrating that a correct implementation of the inviscid finite-rate chemistry solver has been achieved. All of the results of this section have been computed using the rule-based specification described in chapter VI. The computations for these results were performed on a Silicon Graphics Power Challenge server using 195mhz R10K processing units. The second section measures the performance of the scheduling algorithm on a variety of grids, as well as the overall performance of the resulting numerical simulation.

7.1 Finite-Rate Solver Results

Three chemistry models are utilized in this presentation: the ideally dissociating oxygen model described in Vincenti and Kruger [39], a similar dissociating oxygen model that includes the vibrational energy term for diatomic oxygen, and the five species air model of Kang and Dunn[57]. These models, the associated thermodynamic variables, and the reaction rates are included in appendix B.

7.1.1 Shock Relaxation for Ideally Dissociating Oxygen Model

Shocks consist of a very thin region of fluid where significant changes in the fluid properties take place. In the inviscid formulation of the Navier-Stokes equations these shocks exist as discontinuous solutions. In reality these structures have a finite thickness, usually on the order of several mean free paths for strong shocks. Chemical reactions, on the other hand, require many more molecular interactions than can occur in the narrow shock structure. As a result, a strong shock is typically followed by a relaxation region where the flow adjusts from the frozen shock conditions (no chemistry) to conditions of chemical equilibrium. For a standing shock, this problem can be reduced to a set of ordinary differential equations (ODEs) in space. Since these ODEs can be solved very accurately using standard numerical algorithms such as Runge-Kutta methods, a Runge-Kutta solution will be used to provide a reference solution that is compared to a solution generated by the finite volume method described in chapter V. The Runge-Kutta method used to integrate the reference ODEs is a fifth order Cash-Karp algorithm augmented with adaptive step-size controls[58]. The adaptive step-sizes provided by the Runge-Kutta solution are used to generate the grid for the finite volume scheme. The resulting grid is composed of 4839 cells. The ideally dissociating oxygen model is used for this comparison.

The test case presented here depicts the relaxation zone behind a normal shock where the free stream conditions are given by a velocity of $u_\infty = 4000\ m/sec$, a pressure of $p_\infty = 70\ Pa$, and a density of $\rho_\infty = 0.0002\ Kg/m^3$. The conditions immediately behind the shock, and the inflow conditions to the finite volume scheme, are given by a velocity of $u_{frozen} = 671.4\ m/sec$, a pressure of $p_{frozen} = 2732.9\ Pa$, density of $\rho_{frozen} = 0.00119\ Kg/m^3$ and a temperature of $T_{frozen} = 8838.34\ K$. The mixture of diatomic and monatomic oxygen is given by a mass fraction of monatomic oxygen of 0.000007. The outflow boundary condition used in the finite volume scheme is given by the prescribed equilibrium pressure $p_{equilibrium} = 3037.2\ Pa$.

The results of the relaxation region simulation are shown in figures 7.1 and 7.2. The spatial dimension is plotted on a logarithmic axis to best illustrate the details of the relaxation structure. As is shown from the plots, the Runge-Kutta and finite volume results are in excellent agreement. This simulation result indicates that the basic features, such as reaction rates and equilibrium constant computations, spatial advection terms, as well as chemistry source term integration, are correct. Unsteady, time accurate simulations will be demonstrated in the next section.

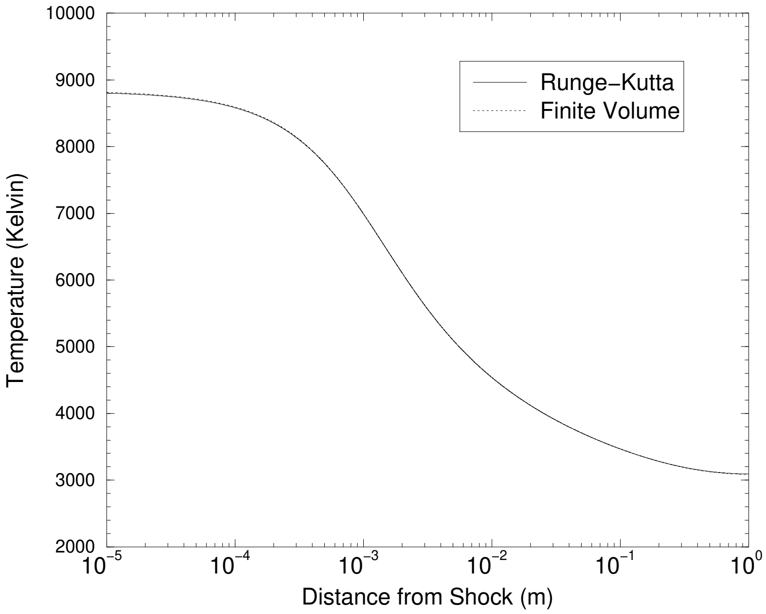


Figure 7.1: Relaxation Zone Temperatures

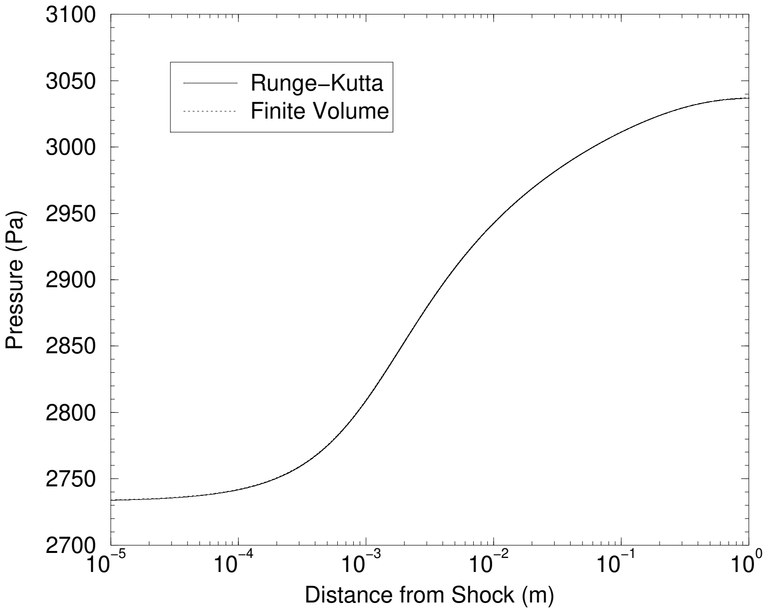


Figure 7.2: Relaxation Zone Pressures

7.1.2 Dissociating Oxygen-Shocktube Test Case

The shocktube scenario is a well known test case for unsteady numerical computations. This problem contains many of the basic discontinuous structures that are found in inviscid flow fields. For the ideal gas case, the solution to the shock tube problem can be determined exactly and provides a reasonable benchmark for numerical algorithms. The shock tube problem was solved with the ideal gas model, establishing that the numerical solver is able to handle the basic flow. In addition, a dissociating oxygen model was used to simulate a shocktube scenario. Although an exact reference solution does not exist for this case, a simulation of this type can be used to ascertain the qualitative correctness of the code. In addition, the results of this numerical experiment can be compared to the results of others, such as those produced by Westmoreland[59]. Initial tests indicated that one of Westmoreland's reaction rates was incorrect. However, utilizing the same undocumented reaction rates, the two simulations produced similar results. The results presented here use the correct reaction rates, although the simulation time illustrated has been adjusted so that the final results are qualitatively similar to those from Westmoreland[59].

The initial conditions consist of a driver side density of $\rho = 1.26062\ Kg/m^3$, temperature of $T = 3000\ K$, pressure of $p = 1 \times 10^6\ Pa$, and degree of dissociation[1] of $\alpha = 0.0176349$. The driven side of the shock tube is described by a density of $\rho = 0.192364\ Kg/m^3$, a temperature of $T = 2000\ K$, a pressure of $p = 1 \times 10^5\ Pa$, and a degree of dissociation of $\alpha = 0.000329186$. To maintain time accuracy, the second order three point backward time integration was performed using five Newton iterations. In addition, four Gauss-Seidel iterations were performed in the iterative solver. A time-step of $0.2\ \mu\ sec$ was employed on a grid composed of 1000 cells spanning an interval of $0.08\ m$.

The solution of the shocktube simulation is presented $240\ \mu\ sec$ after the diaphragm has burst. These results are presented in figures 7.3 through 7.5. Superimposed on these plots are frozen and equilibrium simulation results. The frozen results are obtained by setting the chemistry source terms to zero, while the equilibrium results are obtained by setting the reaction rates to sufficiently large constant values. Figure 7.3 clearly indicates the relaxation region behind the right moving shock. Notice that the relaxation temperature begins at the frozen temperature and approaches the equilibrium solution. Also notice that the exothermic recombination reactions

[1] The degree of dissociation is the ratio of the number of monatomic atoms to the total number of atoms in the mixture. For this simple case, this parameter reduces to the mass fraction of dissociated oxygen.

elevate the temperature in the expansion fan for both the equilibrium and finite-rate cases. The pressure, as illustrated in figure 7.4, remains essentially unchanged regardless of the chemical model. This is a typical result for dissociating reactions in gas dynamics. Finally, the plots of the degree of dissociation in figure 7.5 reveals the role dissociation plays in the relaxation region.

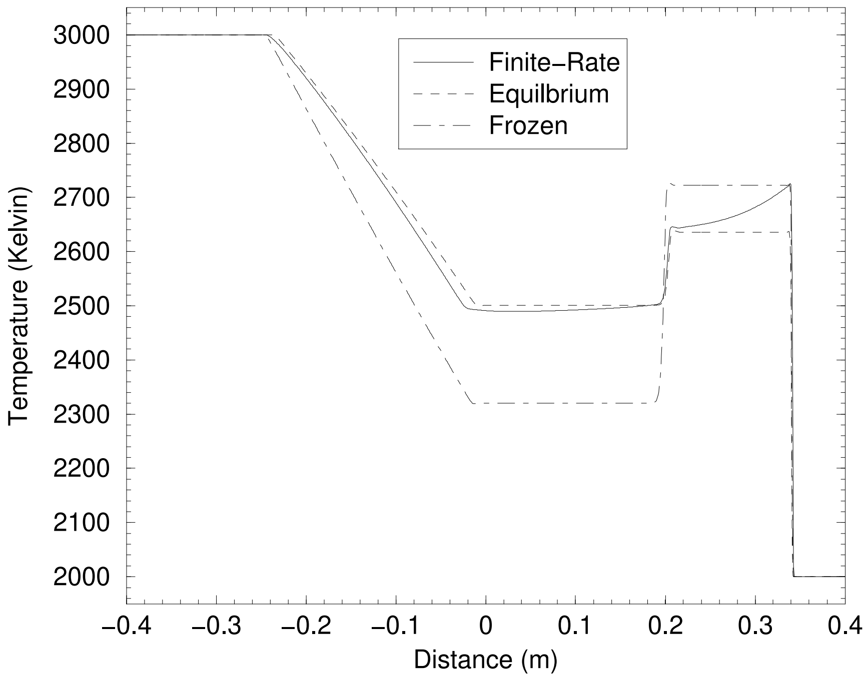


Figure 7.3: Temperature Distribution (Dissociating Oxygen Shocktube)

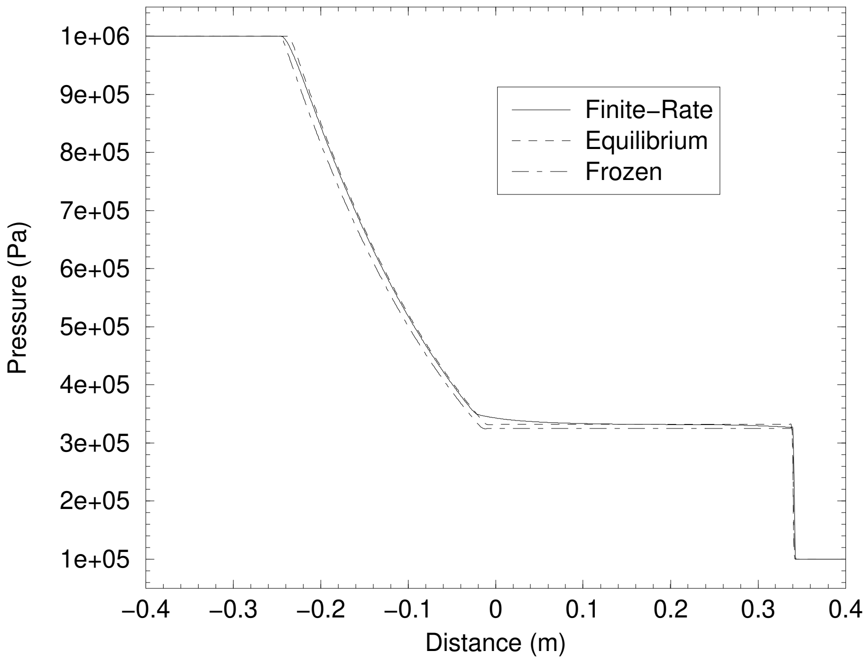


Figure 7.4: Pressure Distribution (Dissociating Oxygen Shocktube)

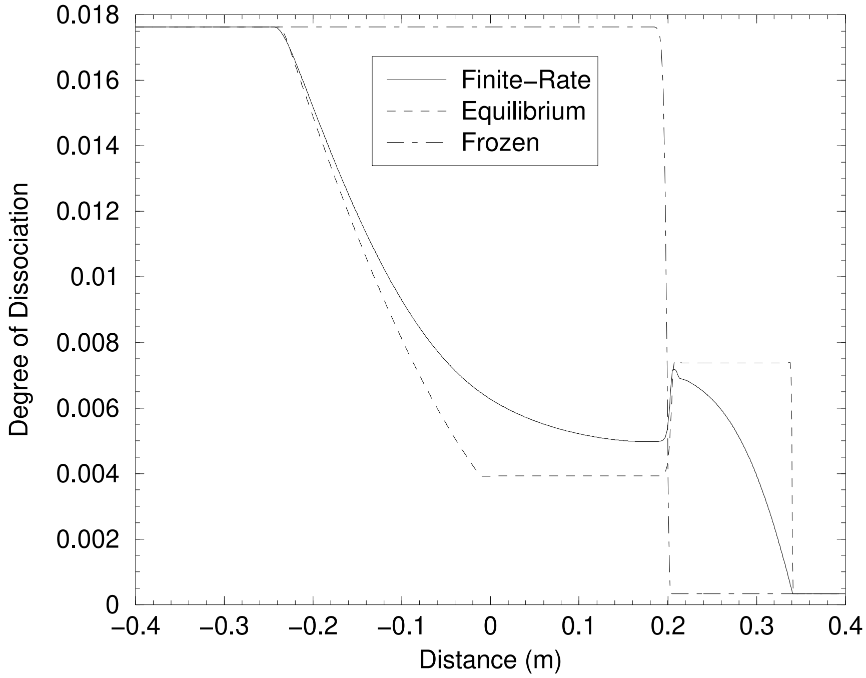


Figure 7.5: Degree of Dissociation (Dissociating Oxygen Shocktube)

7.1.3 High Temperature Supersonic Nozzle

This simulation considers the flow of high temperature air in a supersonic nozzle. The geometry of the nozzle is given by the area relationship:

$$A = \frac{\pi}{16}\left[1 + \sin\left(\frac{\pi x}{2L}\right)\right]^2, \tag{7.1}$$

where L is the nozzle length. For this example, L is two meters. A 100 by 40 point grid is used for this simulation, as illustrated in figure 7.6. The five-species model of Kang and Dunn is employed. Frozen computations were performed by setting the chemistry source terms to zero, while equilibrium computations were performed using large constant reaction rates. The results are compared with the computed values of Westmoreland[59] and Cinnella[44].

The supersonic inlet boundary conditions are given by a pressure of $p = 1.945 \times 10^5\ Pa$, a temperature of $T = 9001\ K$, and a density of $\rho = 0.0385433\ Kg/m^3$. The mixture is set to the equilibrium composition given by the mass fractions 1.33836×10^{-5}, 0.233854, 0.0515657, 0.713442, and 0.00112502 for $O2$, O, $N2$, N, and NO, respectively. The velocity of the inlet flow is 4000 m/sec.

As the temperature contours indicate in figure 7.7, this simple geometry produces a rather complex flow-field structure. The initial sharp turn at the nozzle inlet creates an expansion fan that intersects the centerline at approximately 30 centimeters from the nozzle inlet. The expansion is followed by a weak compression induced by the flow along the inwardly curving nozzle wall. This compression wave intersects the centerline at approximately 80 centimeters leading to an increasing temperature along the centerline in this region. As can be seen in the NO contours in figure 7.8, these fluid dynamic structures are also coupled to the production and destruction of species within the flow. Thus, this test case represents a rather complex flow situation.

The validation of the solution method is accomplished by comparing the centerline flow properties to the results of Westmoreland[59] and area averaged properties to the results of Cinnella[44]. In general, the centerline results agree extremely well with the work of Westmoreland. The area averaged curves also match well with Cinnella's results. These results are summarized in figures 7.9 thru 7.20.

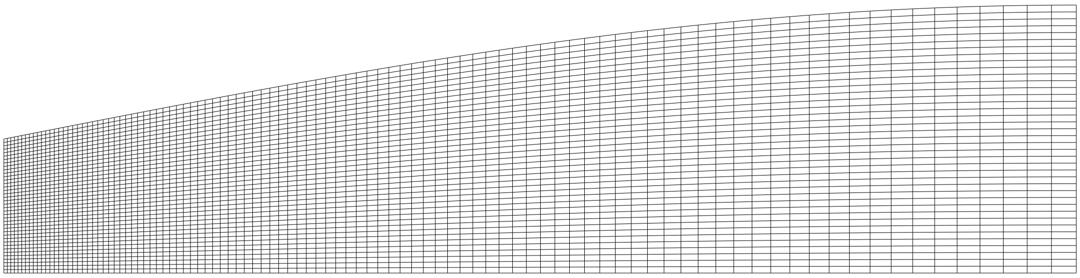


Figure 7.6: Supersonic Nozzle Grid

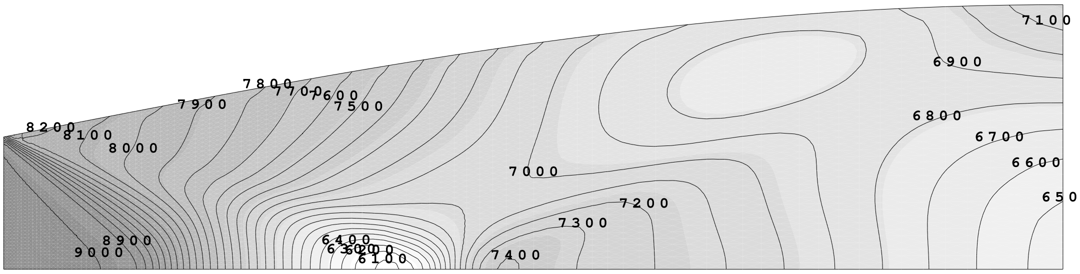


Figure 7.7: Temperature Contours (Supersonic Nozzle, 5 Species Air)

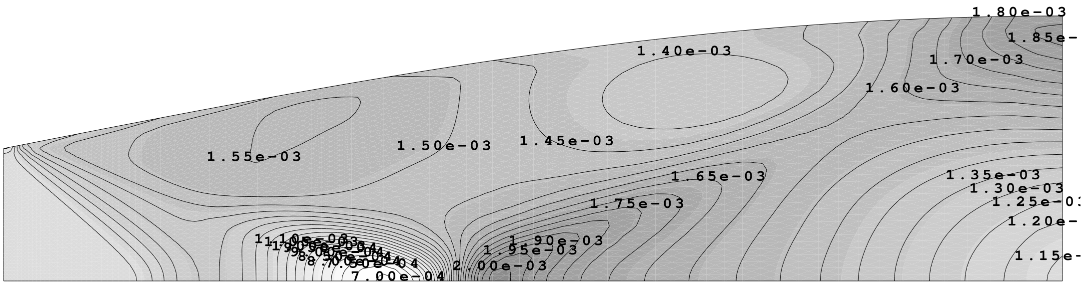


Figure 7.8: NO Mass Fraction Contours (Supersonic Nozzle, 5 Species Air)

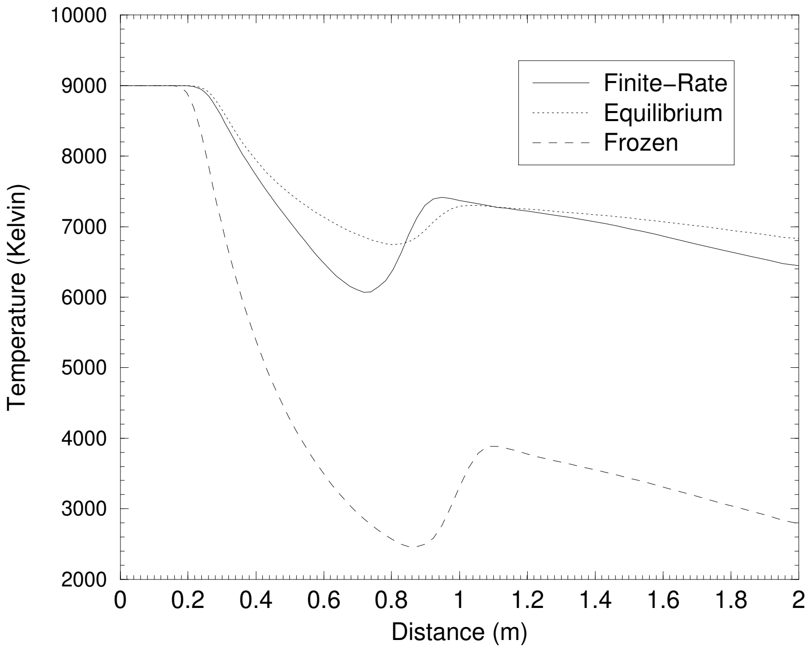


Figure 7.9: Centerline Temperatures (Supersonic Nozzle, 5 Species Air)

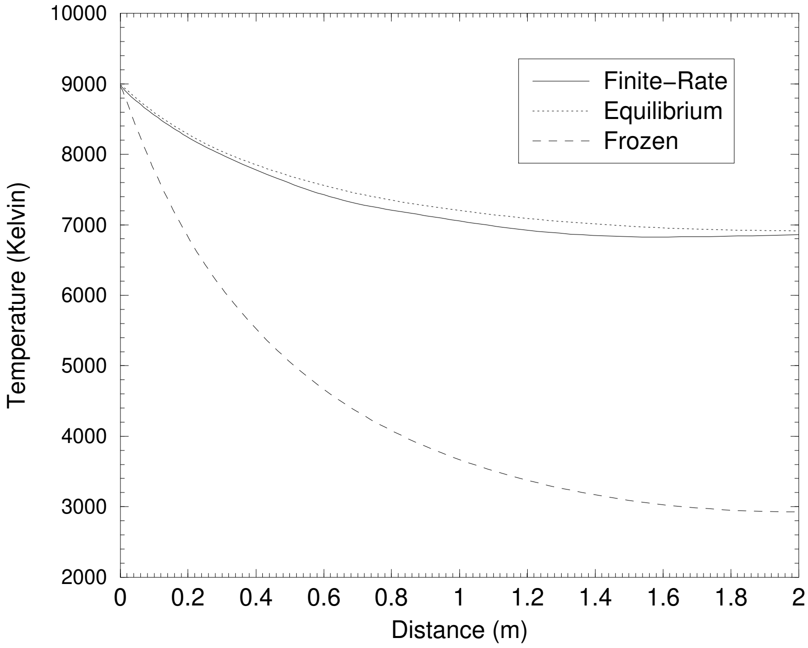


Figure 7.10: Area Averaged Temperatures (Supersonic Nozzle, 5 Species Air)

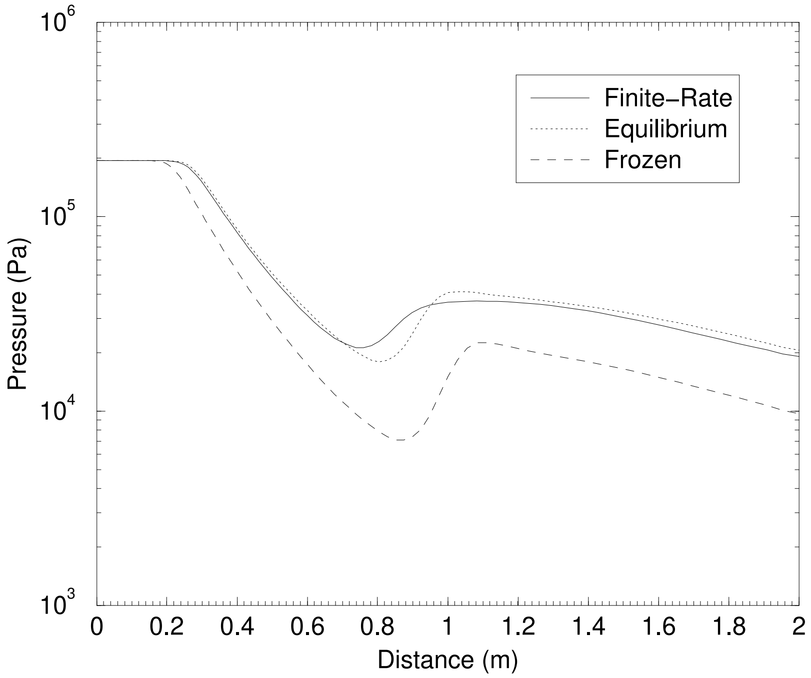


Figure 7.11: Centerline Pressures (Supersonic Nozzle, 5 Species Air)

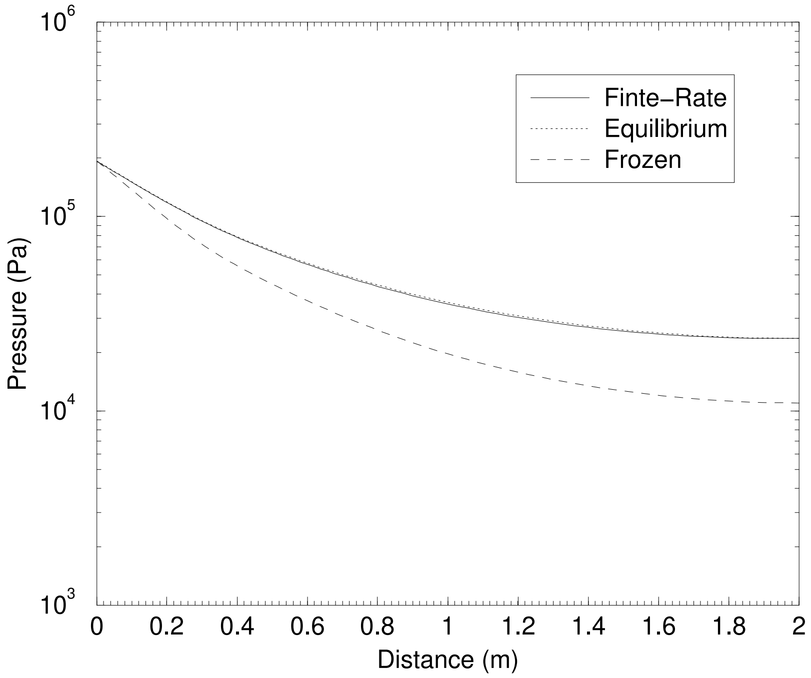


Figure 7.12: Area Averaged Pressures (Supersonic Nozzle, 5 Species Air)

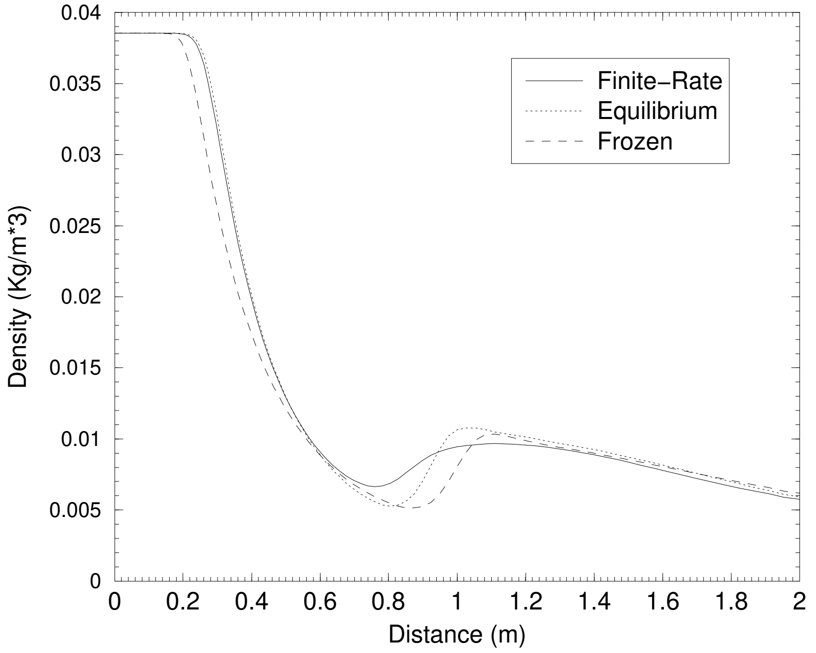


Figure 7.13: Centerline Densities (Supersonic Nozzle, 5 Species Air)

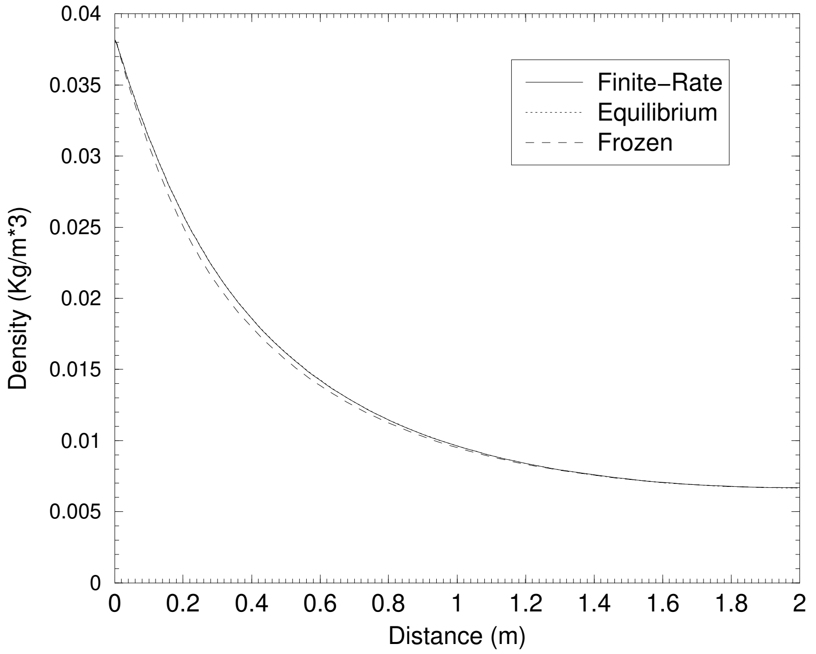


Figure 7.14: Area Averaged Densities (Supersonic Nozzle, 5 Species Air)

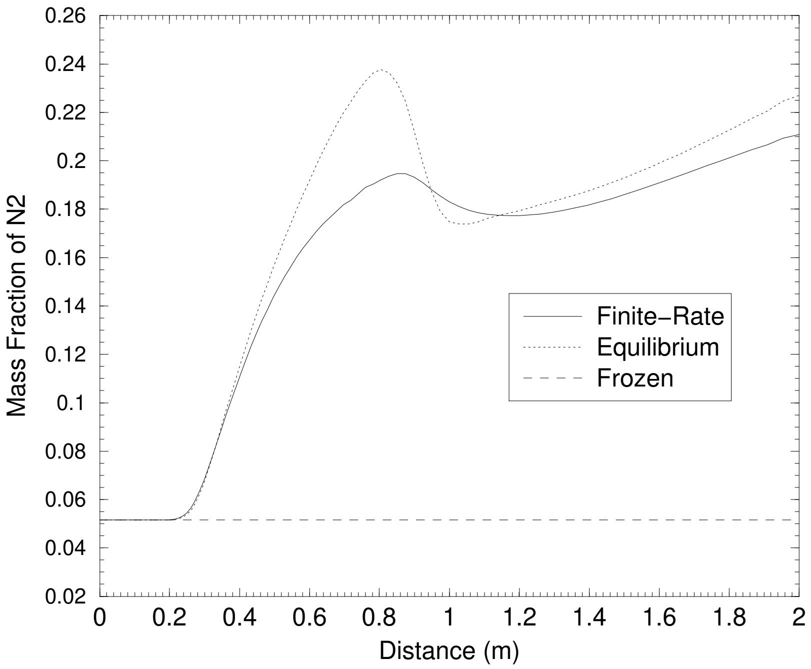


Figure 7.15: Centerline N2 Mass Fractions (Supersonic Nozzle, 5 Species Air)

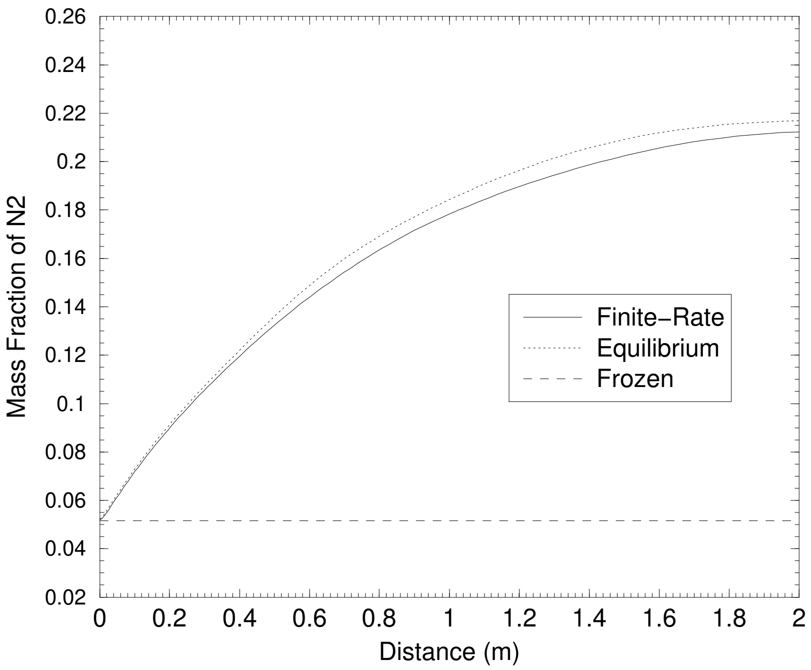


Figure 7.16: Area Averaged N2 Mass Fractions (Supersonic Nozzle, 5 Species Air)

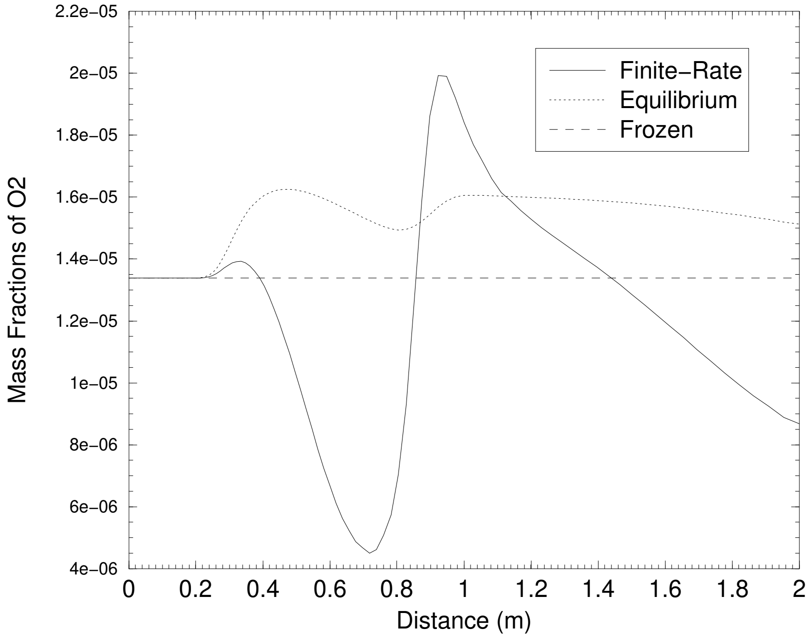


Figure 7.17: Centerline O2 Mass Fractions (Supersonic Nozzle, 5 Species Air)

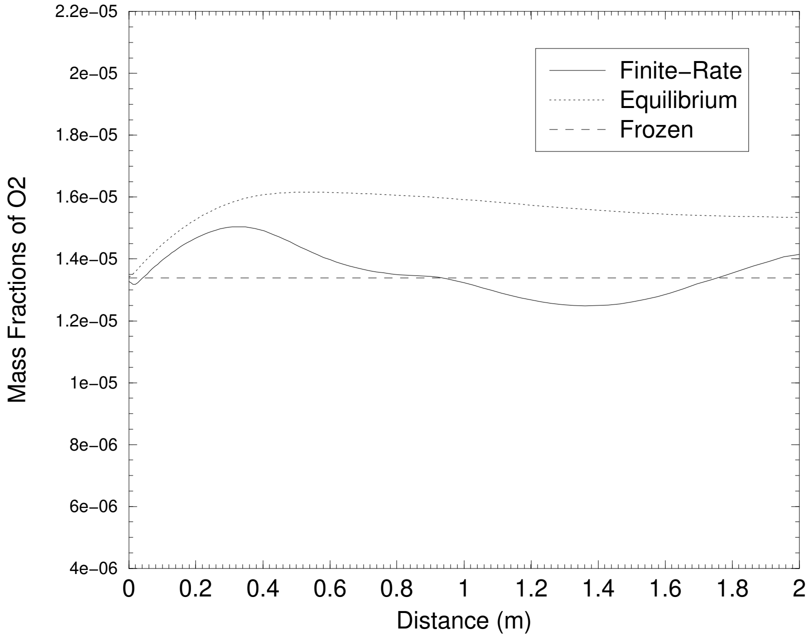


Figure 7.18: Area Averaged O2 Mass Fractions (Supersonic Nozzle, 5 Species Air)

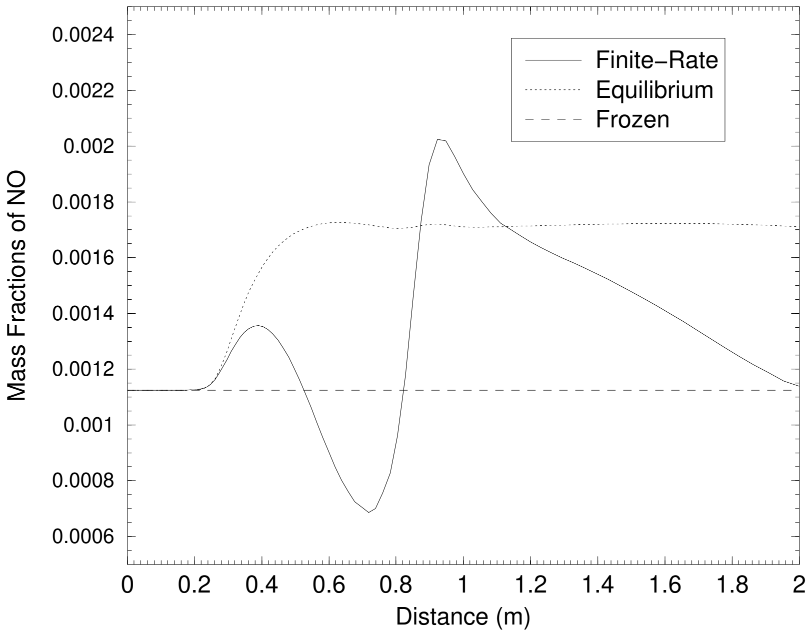


Figure 7.19: Centerline NO Mass Fractions (Supersonic Nozzle, 5 Species Air)

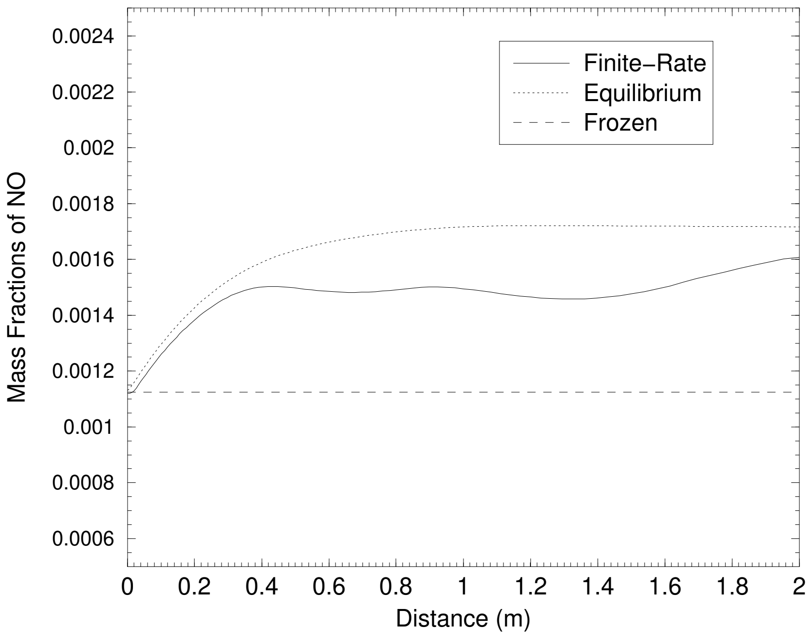


Figure 7.20: Area Averaged NO Mass Fractions (Supersonic Nozzle, 5 Species Air)

7.1.4 Mach 10 Blunt Cone

The final case that is used to evaluate the solver is a hypersonic blunt-body case. The geometry is defined by a blunted 9-deg half-angle cone. The nose radius for this geometry is $r = 6.35\ cm$. The free stream conditions are given by a Mach 10 flow with an ambient pressure of $p = 2.65 \times 10^4\ Pa$ and temperature of $T = 223\ K$. Air is considered as a mixture of 22 percent oxygen and 78 percent nitrogen by mass. A solution to this geometry for the ideal gas and equilibrium chemistry model was obtained by Cox[60]. Four cases are run for this example: 1) an ideal gas model, 2) a frozen chemistry model that includes vibrational terms, 3) an equilibrium calculation that is performed using large constant reaction rates, and 4) a finite-rate computation using the five-species air model of Kang and Dunn discussed earlier. The grid is a 71 by 40 point mesh, with 71 points distributed along the solid boundary, as illustrated in figure 7.21.

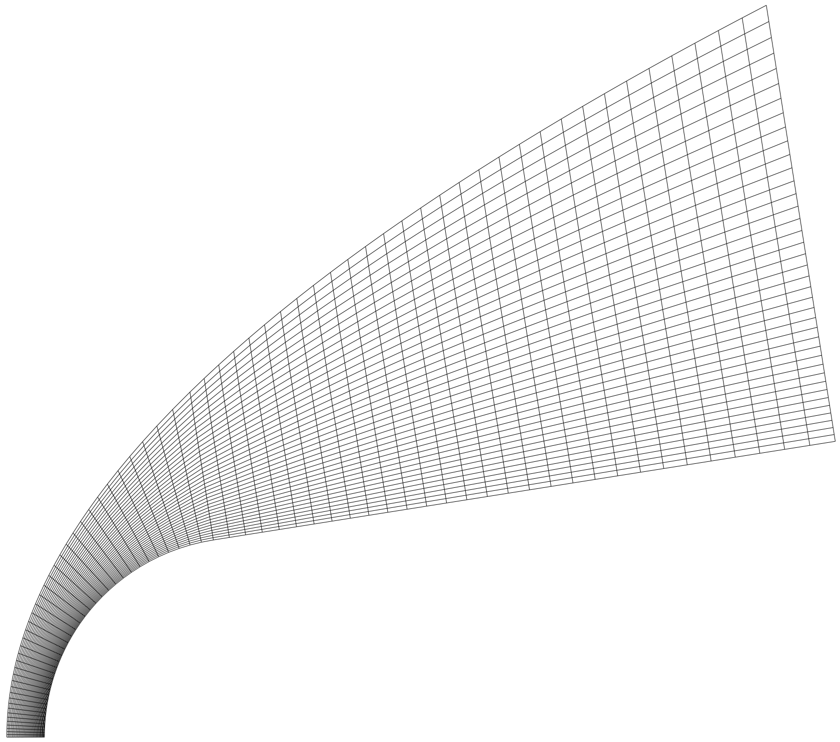


Figure 7.21: Blunt Cone Simulation Mesh

A comparison of the results of the ideal gas and finite-rate models are shown in the temperature contours of figure 7.22 and the density contours of 7.23. The importance of including chemistry source terms in high speed flow simulations is evident in these results. The ideal gas model predicts a much higher temperature in the stagnation region, while the finite-rate chemistry results show cooler temperatures and higher densities. This is due to the presence of endothermic dissociation reactions occurring in the stagnation region of the flow. In addition, the location of the bow shock is significantly closer to the body in the finite-rate case. These results are similar to the comparisons made by Cox[60] for equilibrium chemistry simulations.

The results along the stagnation streamline more clearly illustrate the general trends as shown in figure 7.24. These results closely match the results of Cox, with the exception that the shocks are located slightly closer to the body. However, these differences appear to be on the order of the coarse mesh spacing used in the simulations of Cox. The finite-rate results show a slightly greater bow shock standoff than the equilibrium results. In addition, the finite-rate results are able to capture the relaxation region behind the bow shock, although a more refined mesh would be required to correctly capture the peak temperature. The finite-rate temperature is nearly identical to the equilibrium value at the stagnation point. The frozen calculations show that the vibrational contributions play a significant role in the cooling of the flows in the stagnation region.

The temperature results along the body also show good agreement with the predictions of Cox. The equilibrium computations match extremely well, while the ideal gas computations predict a slightly lower shoulder temperature. The finite-rate temperature matches the equilibrium results near the stagnation point, but the finite-rate results become cooler as the flow accelerates around the slope of the body. This is due to "freezing" of the flow, which inhibits exothermic recombination reactions in the cooler regions when using the more realistic finite-rate model.

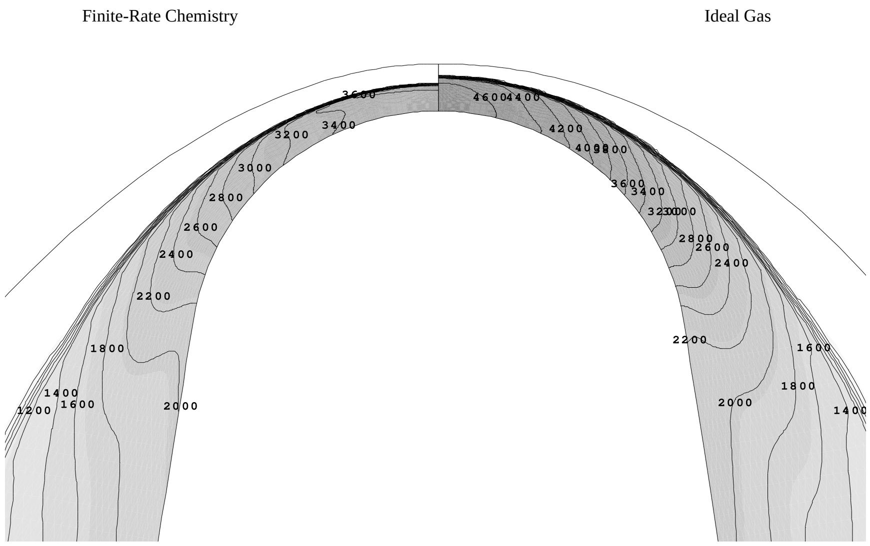


Figure 7.22: Blunt Cone Temperature Contours

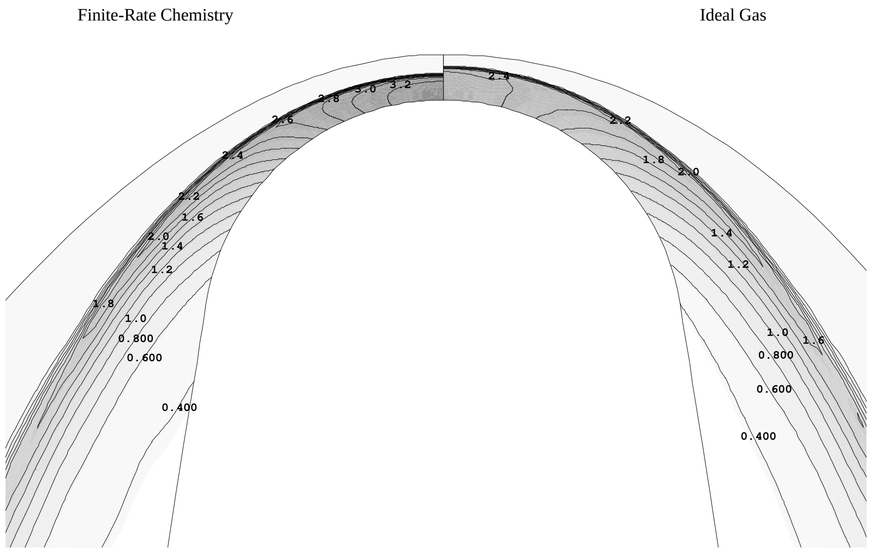


Figure 7.23: Blunt Cone Density Contours

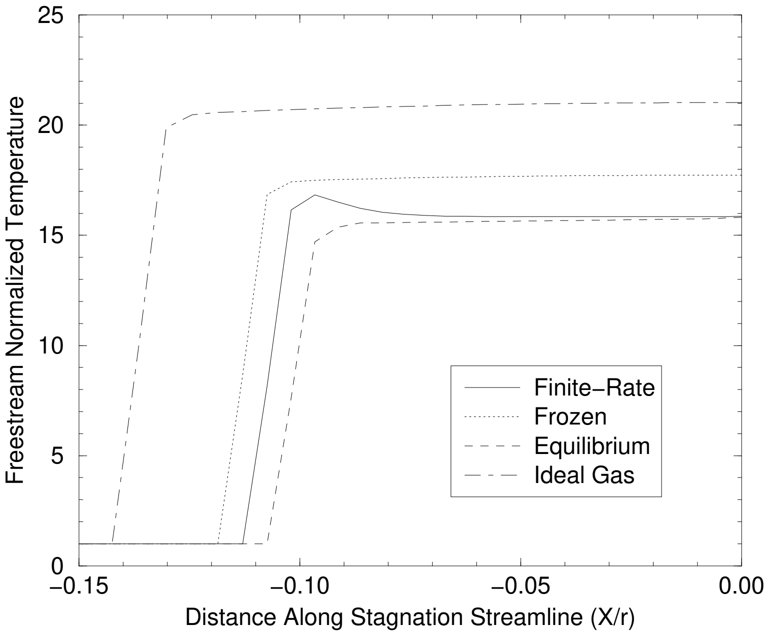


Figure 7.24: Blunt Cone Stagnation Streamline Temperatures

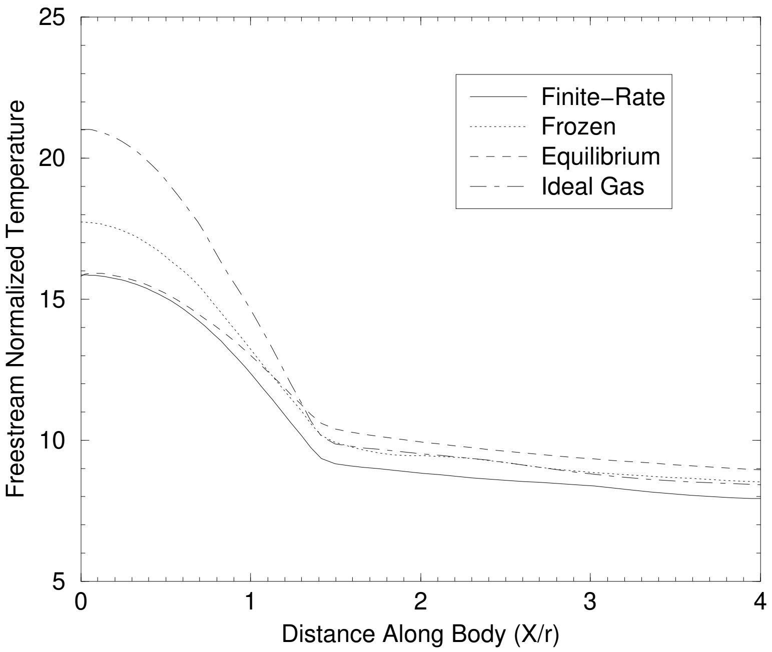


Figure 7.25: Temperatures along Blunt Cone Surface

7.2 Scheduling Performance Results

Since the generation of an execution schedule is not directly required by the simulation, but rather is required by the present approach, the time required to generate an execution schedule can be considered as computational overhead. It is necessary that the scheduling time remain small, so that the approach remains an economical alternative to more traditional methods. This section will measure the scheduling time on a selection of representative grids and determine empirically the trends in scheduling cost (time).

Three basic types of grids are considered for this simulation: a one-dimensional grid consisting of N cells, a two-dimensional grid consisting of $N \times N$ cells and a three-dimensional grid consisting of $N \times N \times N$ cells. The problem size, n, is given by the number of cells in the simulation. The scheduling time for grids of sizes from 10^3 to 10^6 cells are measured at exponential intervals. The resulting scheduling times are plotted in the logrithmic axis plot of figure 7.26. For grids of size $30,000$ cells or less, all of the grids produced similar scheduling times. For larger grids, however, there was a divergence of results, with the one-dimensional grids having the shortest scheduling times and the three-dimensional grids having the largest. The largest scheduling time measured was approximately 15 minutes for a three-dimensional grid consisting of 10^6 cells. Since most simulations on a grid of this size are likely to require processing time on the order of CPU-days, this is a respectably small scheduling time.

One concern is the rate of growth of the three-dimensional scheduling time. To gain an understanding of this growth, a least-squares curve fit was used to obtain a power law approximation to the curves. Since the growth appears after grids of approximately $30,000$ cells, only grids larger than this are considered in this curve fitting. The resulting curve fits appear as lines in figure 7.26. From these curve fits, an estimation of the complexity of the algorithm can be made. These estimations are listed in table 7.1. From this measurement it is determined that while the scheduling time for the one-dimensional grid increases as $O(n)$, the three-dimensional grid increases at approximately $O(n^{1.5})$.

One source of additional scheduling overhead with the three-dimensional meshes is that although the final deduction produces sets of entities over contiguous intervals, the intermediate results may not remain contiguous. For example, if the Gauss-Seidel algorithm performs computations in diagonal planes across the computational grid, then the existential deduction

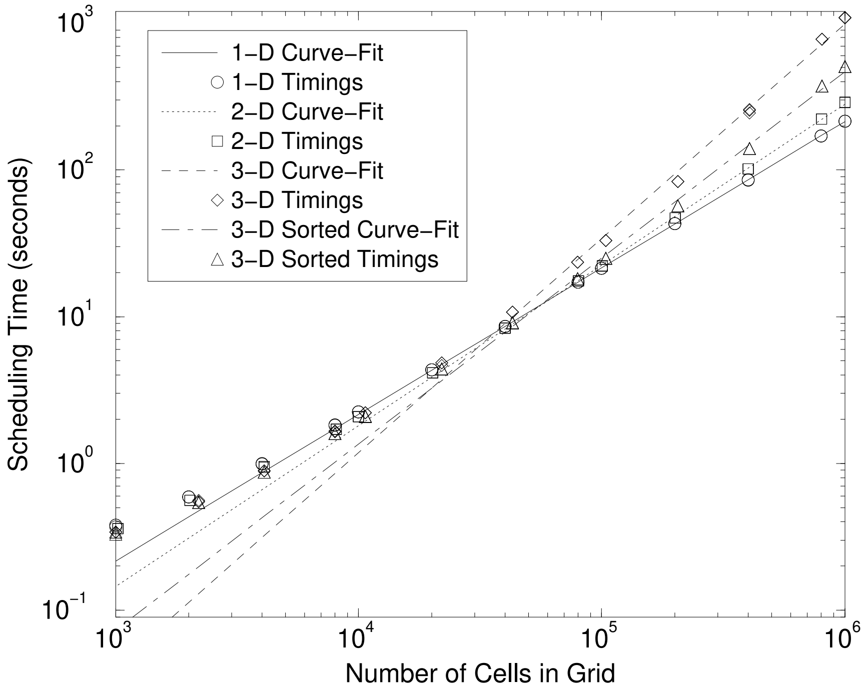


Figure 7.26: Scheduling Time v.s. Grid Size

Table 7.1: Measured Scheduling Order of Complexity

Grid Type	Complexity
1-D	$O(n^{0.998})$
2-D	$O(n^{1.096})$
3-D	$O(n^{1.461})$
3-D Sorted	$O(n^{1.259})$

phase will compute non-contiguous sets of entities in the Gauss-Seidel recursive rule evaluation. As a result, the representation of sets of entities as a number of intervals no longer remains bounded, consequently, the assumption of $O(1)$ cost for set unions and intersections no longer applies. To test this hypothesis, the cells in the three-dimensional grid were sorted according to the reverse Cuthill McKee[61] ordering. This ordering, which can be computed in $O(n)$ time, has the advantage that it matches the concurrent Gauss-Seidel algorithm order of evaluation. Note that this does not change the algorithm, only the association of entity identifiers with cells. The resulting sorted three-dimensional grid is also plotted in figures 7.26. As listed in table 7.1, this sorting reduced the order of the three-dimensional scheduling time from $O(n^{1.5})$ to $O(n^{1.26})$.

In addition to measuring the scheduling time for this rule-based approach, it is also important that the resulting schedules execute with comparable performance to other approaches. To estimate the execution performance, a nozzle simulation was run using both the rule-based code and a Fortran based ideal gas solver due to Janus[43]. The hardware counters available on the SGI R10K processors are used to determine the actual floating point operation rates for the simulations. Both an ideal gas and a full chemistry model are measured for the rule-based solver. The measurements of execution time includes the time required to generate the schedule. Thus this represents the actual flop rate including all costs associated with the rule-based approach. Table 7.2 lists the results of this performance measurement and shows that although the rule-based approach is somewhat less efficient than the Fortran solver of Janus[43], it still produces a performance competitive with more traditional design approaches. It is also important to note that the solver of Janus is structured, while the solver presented here is a much more flexible unstructured solver.

Table 7.2: Performance Results on Nozzle Simulation

Code	Model	Flops	Time (sec)	Rate (M-Flops)
Rule-Based	Ideal Gas	41.41e9	1174	35.3
Rule-Based	Finite-Rate	252.89e9	5803	43.6
Fortran	Ideal Gas	33.80e9	749	45.1

CHAPTER VIII

CONCLUSION

This study presented an implementation strategy for developing high performance numerical simulation software. This strategy consists of specifying numerical computations as a series of rules. These rules are much like the guarded horn clauses of Strand[28], except that the clause specification includes mapping operators. These mapping operators provide the fundamental method for describing mesh topology. Since these mapping semantics are included in the rule specification, the approach described here does not suffer from the same problems in distribution of array values across processors. Thus, the work presented here can be thought of as a Strand that has been adapted to the needs of finite-element and finite-difference methods.

Like the CHAINS[31] model, the one described here is at a high semantic level. The rule specifications described in chapter III closely matches the semantics of the derived numerical model equations. Unlike the CHAINS model, the specification described here facilitates the description of iteration and linear solver algorithms. Thus the specification described here strikes a balance between the power and abstraction of the model.

Like the formal specification approach of Birken[32, 33], the specification approach described here is able to provide guarantees concerning the logical consistency of the derived simulation program. Unlike the work of Birken, the scheduling time required by this system is minimal, since it does not invoke expensive heuristic searches utilizing generic theorem provers. Instead, the approach presented here employs run time scheduling to produce similar results, with scheduling times that are comparable to a single simulation iteration.

In addition to the development of a specification language for finite-element and finite-difference schemes, this study developed simulation software for flows involving chemical non-equilibrium. This simulation software was implemented using the developed specification language to illustrate the effectiveness in the specification of real applications. The resulting software was validated using a variety of test cases demonstrating successful implementation.

In addition, an interesting generalization of the Gauss-Seidel algorithm was developed utilizing recursive rule specifications, allowing the development of concurrent Gauss-Seidel algorithms for unstructured meshes. Performance measurements also indicate that the overhead associated with the rule specification methodology is minimal.

While the results of this study look quite promising, there is much left to address. While the specification system described here only specify rules that compute values, the specification of rules that compute entities or entity relationships will be needed if adaptive schemes are considered. In this case, scheduling rules for execution becomes much more complex since partial schedules need to be generated as part of the scheduling algorithm. As a result, the overall performance of the simulation system will be more sensitive to scheduling costs. In addition, care will be needed in order to preserve the monotonicity of the deductions if rules are allowed to generate entities as a result. More research in all of these areas will be essential before fully adaptive schemes can be captured by a rule specification approach.

APPENDIX A

NUMERICAL SOLUTIONS OF PARTIAL DIFFERENTIAL EQUATIONS

Modeling physical phenomena using numerical methods begins with the determination of a mathematical description of the problem at hand. There are many such models in engineering and the sciences, for example the Navier-Stokes equations, the Maxwell Equations, the shallow water equations, and so on. In many cases these models are represented by partial differential equations, or PDEs. A partial differential equation for a given function $u(x, y, \cdots)$ defined over a domain Ω with boundary Γ is given by a relation of the form

$$F(x, y, \cdots, u, u_x, \cdots, u_{xx}, u_{xy}, \cdots) = 0, \tag{A.1}$$

where F is a function of the independent variables $x, y, \cdots$, of the function u, and of a finite number of its partial derivatives. Typically the dependent variables represent physical properties of the models, for example energy, momentum, mass, etc. In the mathematical interpretation these dependent variables simply become functions of the independent variables. The problem is to identify or approximate these functions. Assuming that these approximations are accurate and that the model equations represent underlying physical processes, these functions will closely model the behavior of physical systems.

A.1 Discretization of the Domain

The numerical solution of partial differential equations requires a finite representation of the physical domain. Generally this finite representation is called a discretization. There are four primary discretization approaches: the finite difference methods[62], finite element methods[63], spectral methods[64], and finally particle methods[65][66]. Finite difference methods are based on approximate representations of neighborhoods of discrete points in the domain. Finite element methods (of which finite volume is a sub-case) are based on an integral view of the domain.

Spectral methods discretize partial differential equations in a transformed space (such as the frequency domain under the Fourier transform). Particle methods such as Monte-Carlo [65] or Lattice Gas [66] approaches are usually applied to problems where local thermodynamic equilibrium can not be assumed. Here, finite difference and finite element discretization methods will be discussed in more details.

The finite difference and finite element discretizations can be characterized by a finite set of points or "nodes" in the domain Ω and on the boundary Γ, and by the domain topology, which is represented in a discrete sense by associations or connectivities between these nodes. In the finite difference discretization the topology of the domain is represented by nodal connectivities. In this representation, each node is associated with neighboring nodes, so that a discrete interpretation of the neighborhood of that node can be constructed. In the finite element discretization the domain is divided into volumes that do not overlap and completely cover the domain. Volumes are identified by their shape (e.g. tetrahedral, hexahedral, prism) and relationships to nodes. Often relationships to faces or edges of the volumes are also used in finite element computations. This will be described in more detail in later sections.

The discretization produced by either the finite difference or finite element methods, including nodes and topology (connectivity), is referred to as a mesh. The topology of a mesh is formulated using a labeling of the nodes. Meshes can be categorized into two general categories: regular or irregular. In regular meshes the node labels imply topological connectivity, usually through a multidimensional label. In computer applications this label usually is implemented through array indexing. Regular meshes can occur in both triangular (tetrahedral) and quadrilateral (hexahedral) meshes in 2-d (3-d) as indicated in figure A.1. In this figure the nodes of the regular meshes can be labeled with two indexes, (i, j), such that the neighboring nodes can be identified via simple analytic expressions, $(i \pm 1, j \pm 1)$. For irregular meshes it is impossible to describe the topology implicitly from the labeling and so the topology must be specified explicitly by connectivity lists. Numerical solvers based on the regular mesh assumption using multidimensional arrays are typically identified as structured solvers, while solvers based on connectivity lists are identified as unstructured solvers. Notice that the unstructured solver is general in that it can be applied to either irregular or regular meshes, while the structured solver is only applicable to regular meshes (although block structured solvers can accommodate limited degrees of irregularity in the mesh).

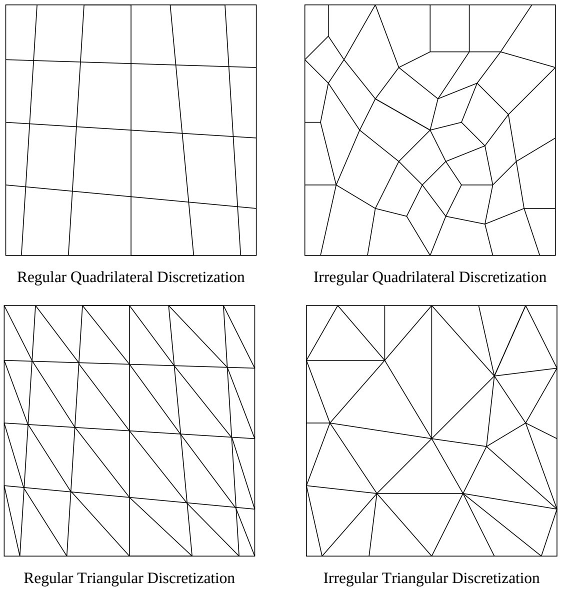


Figure A.1: Regular and Irregular Discretizations of a Square Domain

A.2 Finite Differences

In the finite difference approximation the domain Ω and boundary Γ are discretized such that the domain is represented by N discrete nodes and the boundary is represented by M discrete nodes. Typically each of these nodes are labeled by integers. For example, all of the nodes in the domain can be labeled by integers from 1 to $N+M$, where the domain nodes are labeled from 1 to N, while the boundary nodes are labeled from $N+1$ to $N+M$. The labeling of the nodes is necessary for the specification of the difference equations, although the labeling procedure itself is an arbitrary specification. Basically, although labeling is a necessary step, any labeling that provides a unique identifier to each node is equally valid. The first step of the finite difference approximation is to satisfy the PDE only at the discrete nodes rather than on the continuum of the domain. For example, the linear convection/diffusion equation with a constant wave velocity a and a constant diffusivity ν is given by

$$u_t + au_x = \nu u_{xx} \text{ on domain } \Omega, \tag{A.2}$$

and its finite difference formulation becomes

$$(u_t)_i + a(u_x)_i = \nu(u_{xx})_i \text{ where } i = 1, \cdots, N. \tag{A.3}$$

So far, this specification of the finite difference method is incomplete. It does not include treatments of boundary conditions or reveal how partial derivatives are evaluated.

A.2.1 Approximating spatial derivatives

The basic theory behind approximating spatial derivatives in finite difference equations rests on the assumption that the solution to the PDE is sufficiently smooth that a truncated Taylors series can accurately approximate it. In this view, spatial derivatives at nodes are estimated by finding the derivatives of a polynomial function that approximates the solution at neighboring nodes. Thus the derivatives at a node can be approximated by a function of that node's dependent and independent variables and neighboring nodes' dependent and independent variables. For this discussion we define the neighborhood set of node i to be the set of node labels that will be included in the approximation of the dependent variables at node i. The neighborhood set for node i will be denoted as Q_i. Providing neighborhood sets for each node in the domain results in a directed connectivity graph that describes the topology of the mesh for the finite difference equations. In general this connectivity graph can be represented by a connectivity matrix, where the rows of the matrix represent the neighborhood sets. This connectivity matrix is usually sparse and symmetric. Using a polynomial function to approximate dependent variable derivatives yields expressions for the derivatives that are linear combinations of the dependent variables in a node neighborhood set. Given this strategy, the finite difference approximation for the linear convection diffusion equation given in (A.3) can now be written as

$$(u_t)_i + a \sum_{j \in Q_i} (c_j)_i u_j = \nu \sum_{j \in Q_i} (d_j)_i u_j \text{ where } i = 1, \cdots, N. \tag{A.4}$$

In this equation the coefficients c_j and d_j are produced by the polynomial interpolation of points in the neighborhood set of node i, assuming that node i is contained within the neighborhood set. This view captures the concept of stencils found in finite difference texts.

A.2.2 Approximating Temporal Derivatives

Thus far the description of the discretization of the domain has been in terms of its spatial dimension. Usually the temporal dimension is discretized separately: each nodal value is assumed to progress in time through a sequence of time steps. The current "known" time is denoted by n, whereas the next time step is labeled as $n+1$. Given this notation, a node is completely specified when both its label and time-step are provided. By convention the node is labeled with a superscript that represents the time level of the variable and a subscript that represents the spatial discretization label of the variable. For example, u_i^n represents the value of node i at time level n. The formulation of the time derivatives proceeds in a similar fashion to that of the spatial derivatives, except that there is a choice regarding where in time the temporal derivative is evaluated. For example in explicit schemes the time derivative is evaluated at time level n while implicit schemes evaluate the derivatives at time level $n+1$ or $n+\frac{1}{2}$. For example, using the previously established notation, an explicit formulation would be expressed as

$$\sum_{j=n-k}^{n+1} b_j u_i^j + a \sum_{j \in Q_i} (c_j)_i u_j^n = \nu \sum_{j \in Q_i} (d_j)_i u_j^n \text{ where } i = 1, \cdots, N, \tag{A.5}$$

while a simple implicit formulation would be expressed as

$$\sum_{j=n-k}^{n+1} b_j u_i^j + a \sum_{j \in Q_i} (c_j)_i u_j^{n+1} = \nu \sum_{j \in Q_i} (d_j)_i u_j^{n+1} \text{ where } i = 1, \cdots, N. \tag{A.6}$$

In a similar fashion the Crank-Nicholson method, which expands the temporal derivative about $n+\frac{1}{2}$, would be expressed as

$$\sum_{j=n-k}^{n+1} b_j u_i^j + \frac{a}{2} \sum_{j \in Q_i} (c_j)_i (u_j^n + u_j^{n+1}) = \frac{\nu}{2} \sum_{j \in Q_i} (d_j)_i (u_j^n + u_j^{n+1}) \text{ where } i = 1, \cdots, N. \tag{A.7}$$

Notice that in the previous notations the difference equations can be expressed as matrices. The sums over each node's neighborhood set can be interpreted as a vector dot product. The combination of all of the relations can be interpreted as a matrix vector product. If the vector of all nodal values is denoted as $\mathbf{u}$ then a vector form of equation (A.7) can be given as

$$A\mathbf{u}^{n+1} = B_n \mathbf{u}^n + B_{n-1}\mathbf{u}^{n-1} + \cdots + B_{n-k}\mathbf{u}^{n-k}. \tag{A.8}$$

In equation (A.8), A, B_n, $\cdots$, B_{n-k} are matrices. For explicit formulations A is a diagonal matrix while in implicit formulations A is a general matrix that must be inverted in order to solve for $\mathbf{u}^{n+1}$.

A.2.3 Boundary Condition Treatments

So far the discussion of the finite difference method has only considered treatment of the nodes that are within the domain and has ignored the boundary nodes. In order for PDE problems to be well posed additional information is supplied at the boundaries. These boundary conditions must be incorporated into the solution process for a complete description of the method. The typical boundary conditions that are posed are classified into two principal types: Dirichlet and Neumann. In Dirichlet boundary conditions the value of the dependent variable is prescribed at the boundary surface Γ. These are perhaps the most straightforward boundary conditions to apply. Assuming a Dirichlet boundary condition that specifies that $u = f(\mathbf{x}, t)$ on Γ one can derive the discrete formulation as

$$u_i^n = f(\mathbf{x}_i, t^n) \text{ where } i = N+1, \cdots, N+M \tag{A.9}$$

In Neumann boundary conditions, derivatives of the dependent variables with respect to the independent variables are specified on the boundary. In this case it is not sufficient to prescribe the values on the boundary nodes, since these values are dependent on the solution of nodes within the domain. There are two approaches that are typically taken to satisfy Neumann boundary conditions. The first follows similar themes to the approximation of derivatives in the finite difference method. In this first approach a polynomial function is used to extrapolate the solution to the boundary nodes. For example, if $u_x = f(x, t)$ is specified on the boundary, then a neighborhood set of the boundary node can be used to satisfy the boundary condition by an equation such as

$$f(x_i, t^n) = \sum_{j \in Q_i} (c_j)_i u_j^n. \tag{A.10}$$

This equation can be used to obtain the solution variable by solving for u_i as in this equation

$$u_i^n = \frac{1}{(c_i)_i}\left(\sum_{j\in Q_i, j\neq i}(c_j)_i u_j^n - f(x_i, t^n)\right) \text{ where } i = N+1, \cdots, N+M. \tag{A.11}$$

One disadvantage of this method is that the boundary treatment is typically one order less accurate than the order of the derivative approximation. This implies that the neighborhood set required at the boundary will often be larger (in the sense of distance) than that required within the domain.

An alternative formulation for Neumann boundary conditions uses the idea of a "ghost" node which is a non-physical node introduced just outside of the domain. In this approach the ghost nodes are assigned values such that the finite difference approximations applied at the domain boundary will produce the desired result. Then the same finite difference method that is applied within the domain is applied on the domain boundary using the ghost node in its neighborhood set. One advantage of this formulation is that the resulting boundary condition can achieve a more accurate representation with less computational effort (that is a smaller neighborhood set on the boundary).

In the end, the ghost node is just a tool that is used to formulate the boundary condition and can be eliminated in the final solution methodology. This will result in a formulation similar to equation (A.11). In practice the ghost node is often not eliminated, for example when the evaluation of the finite difference method is sufficiently complex that duplication of this code in the boundary condition evaluation unnecessarily complicates it. In these cases the ghost node must be explicitly labeled. Assuming that there is a one-to-one correspondence between boundary nodes and ghost nodes, it is easy to see that this labeling can be accomplished by using integers $N+M+1, \cdots, N+2M$.

The boundary condition treatments described here can also be included in the matrix form of equation (A.8). For Dirichlet boundary conditions the size of the vector $\mathbf{u}$ is the number of nodes within the domain. For Neumann boundary conditions the vector $\mathbf{u}$ also contains boundary nodes and possibly ghost nodes.

A.2.4 Impact of Non-Linear Terms

The previous discussions of the finite difference method were limited to linear PDE's. Unfortunately, many real world applications are governed by non-linear PDEs, which include terms that are non-linear, that is they can not be expressed as a linear combination of the partial derivatives of a function value. The viscous Burgers' equation, given as

$$u_t + uu_x = \nu u_{xx} \tag{A.12}$$

is one such example. The introduction of the dependence of wave velocity on the solution in the convective term makes this equation non-linear. The question is, what impact does this have on the finite difference approximation? The introduction of non-linear terms has a great impact on the numerical derivation of the algorithm. In particular the possibility that discontinuities may spontaneously emerge in the solution requires particular care since this violates the Taylors series interpretation relied upon to construct derivatives in traditional finite difference approaches. Although much is known about the treatment of non-linearities in PDE's, the problem is still a subject of research. The impact of current knowledge on the formulation of algorithms will be presented here, rather than an in-depth study of the theoretical foundations.

The most obvious approach to solving equation (A.12) is to derive an explicit method using the methodologies already presented. For this case the straight-forward finite difference formulation of equation (A.12) becomes

$$\sum_{j=n-k}^{n+1} b_j u_i^j + u_i^n \sum_{j\in Q_i} (c_j)_i u_j^n = \nu \sum_{j\in Q_i} (d_j)_i u_j^n \text{ where } i = 1, \cdots, N. \tag{A.13}$$

Notice that in this formulation it becomes impossible to obtain a matrix vector formulation as in equation (A.8) due to the uu_x term. This will cause problems in the implicit formulation since the operator applied to $\mathbf{u}^{n+1}$ is no longer easily invertible. There is another concern due to the effects of truncation error on the above formulation. Formulating a difference equation of the form in equation (A.13) yields a scheme that is no longer conservative. This problem is usually solved by rewriting the PDE such that no dependent variables occur outside of partial derivatives. For the viscous Burgers' equation this formulation is obtained by observing that

$uu_x = \frac{1}{2}(u^2)_x$, thus the conservative form of the viscous Burgers' equation becomes

$$u_t + \frac{1}{2}(u^2)_x = \nu u_{xx}. \tag{A.14}$$

Similarly, this equation can be represented in a finite difference formulation as

$$\sum_{j=n-k}^{n+1} b_j u_i^j + \sum_{j \in Q_i} (c_j)_i (u_j^n)^2 = \nu \sum_{j \in Q_i} (d_j)_i u_j^n \text{ where } i = 1, \cdots, N. \tag{A.15}$$

Often this approach is not sufficient to obtain accurate results, and more sophisticated methods of dealing with the non-linear terms are necessary. In particular, upwinding-based schemes such as flux-splitting methods are often required when dealing with non linear connective terms[47]. These methods can be summarized by assuming that the derivatives are approximated by some, possibly non-linear, function of the dependent variables in the neighborhood set. That is to say that these schemes can be captured by assuming that there exists some function $F(u_j | j \in Q_i)$ that appropriately approximates the flux at node i. Using this notation, equation (A.13) can be rewritten as

$$\sum_{j=n-k}^{n+1} b_j u_i^j + F(u_j^n | j \in Q_i) = \nu \sum_{j \in Q_i} (d_j)_i u_j^n \text{ where } i = 1, \cdots, N. \tag{A.16}$$

Thus far this discussion has not indicated how implicit solvers for non-linear PDE's are addressed. The problem with non-linear PDE's is that the resulting difference equations can not be expressed in the matrix-vector form of equation (A.8). Using vector notation it is possible to describe the general form of the non-linear discretized problem using an approach similar to that used in equation (A.16), as represented by

$$\sum_{j=n-k}^{n+1} b_j u_i^j + \sum_{j=n-k}^{n+1} c_j \Big(F_1(u_k^j | k \in Q_i) + \cdots + F_L(u_k^j | k \in Q_i) \Big) = 0 \text{ where } i = 1, \cdots, N. \tag{A.17}$$

In this equation the functions $F_1, \cdots, F_L$ represent the (possibly) non-linear finite difference formulations for each spatial derivative term of the PDE. This can be generalized to a vector formulation similar to the matrix-vector formulation, by the equation

$$\mathbf{F}_{n+1}(\mathbf{u}^{n+1}) + \mathbf{F}_n(\mathbf{u}^n) + \mathbf{F}_{n-1}(\mathbf{u}^{n-1}) + \cdots + \mathbf{F}_{n-k}(\mathbf{u}^{n-k}) = 0. \tag{A.18}$$

In this formulation, the vector valued functions $\mathbf{F}_{n+1}, \cdots, \mathbf{F}_{n-k}$ are formed by combining the terms of equation (A.17). Although these vector valued functions are non-linear, they have the special property that their dependency graph will have the same structure as the connectivity matrix implied by the neighborhood sets Q_i. The problem with this formulation is that obtaining $\mathbf{u}^{n+1}$ requires finding the inverse of the function $\mathbf{F}_{n+1}$, which is in general difficult to do.

The usual approach used to solve equation (A.18) is to apply the vector form of the Newton method to obtain the zero of the function $\mathbf{G}(\mathbf{u}^{n+1})$ where this function is the left-hand-side of equation (A.18). This method uses the Jacobian of function $\mathbf{G}$, defined as

$$A_{ij} = \frac{\partial G_i}{\partial u_j^{n+1}}. \tag{A.19}$$

The Jacobian matrix A will have the same sparse structure as the connectivity matrix described by the neighborhood sets, that is if $j \notin Q_i$ then $A_{ij} = 0$. This Jacobian can be used to construct a Newton method for iteratively solving for $\mathbf{u}^{n+1}$. An algorithm based on the Newton method, where each iteration of the $\mathbf{u}$ vector is denoted by $\mathbf{u}_k$, is shown in figure A.2. In this algorithm the solution at time level n is used as an initial guess to start the Newton iteration. Provided that the time-steps are small enough, so that the solution at time level n is sufficiently close to that at time level $n+1$, this method will converge quadratically.

$\mathbf{u}_0^{n+1} = \mathbf{u}^n$
for $k = 0$ to NUMITERS do
    $A(\mathbf{u}_k^{n+1})\Delta\mathbf{u}_k^n = -\mathbf{G}(\mathbf{u}_k^{n+1})$
    $\mathbf{u}_{k+1}^{n+1} = \mathbf{u}_k^{n+1} + \Delta\mathbf{u}_k^n$
enddo

Figure A.2: The Newton method algorithm for finding $\mathbf{u}^{n+1}$

A.2.5 Concerns of Stability, Consistency, and Convergence

The discussion of this section has been focused principally on the structural form of algorithms produced by the finite difference method. The methods described are not guaranteed to produce results that are accurate representations of the partial differential equations upon which they are applied. In this regard there is a broad theoretical basis for demonstrating that a particular finite difference method will produce results that are representative of the partial differential equation they model. The basic theoretical approach is to show that the finite difference method will produce a solution that is increasingly close to the solution of the PDE as the number of points used in the discretization are increased (such that the largest distance between neighboring points also decreases). Proving this result is the same as proving that the finite difference scheme is convergent. Two tools that are used to prove convergence are consistency and stability analyses. Consistency verifies that the finite difference derivatives reduce correctly to the true solution derivatives as the mesh is refined, while stability verifies that numerical errors in the derivative formulations do not grow over time. By the Lax theorem a scheme for a linear PDE will be convergent if it is consistent and stable. For non-linear PDE's this stability and consistency is only a necessary condition for convergence. The theory and methodology behind these analysis methods are presented in many texts on the finite difference method (*e.g.* [62]). The point of the present discussion is to identify the basic software structures required to implement these algorithms. In this light, the descriptions of this section cover the structure of convergent and non-convergent schemes alike, although only convergent schemes are of practical interest.

A.3 Finite Elements

In the finite element discretization methodology, the domain is divided into a discrete set of regions or elements. The dependent variables are modeled by a chosen class of functions (for example polynomial functions), defined within each element. These elements are then assembled to form a solution for the entire domain. There are many methods that may be employed to obtain a mathematical description of these functions within each element. The Galerkin method will be discussed here because it represents one of the more generally applicable approaches.

The first step in the finite element method is to obtain an approximate function that represents the solution within an element. The usual approach used here is to introduce a parametric coordinate system within the element, and use shape functions (also called blending functions) to interpolate values described at the nodes. These shape functions (usually identified by the notation N_i) usually have the property of convexity, described by

$$\sum_{i=1}^{m} N_i = 1. \tag{A.20}$$

Using these shape functions, the value of the function within an element is described by a linear combination of nodal values and the shape functions. For example, the function value u is approximated by this method as

$$\tilde{u} = \sum_{i=1}^{m} u_i N_i. \tag{A.21}$$

In this equation $\tilde{u}$ is the approximated function, u_i are the values at the nodes of the element, and N_i are shape functions, that depend on the parametric coordinates of the element. These nodal values and shape functions are chosen such that the compatibility and completeness requirement are satisfied: if the PDE contains derivatives of the $(r+1)$th order, then compatibility requires that the constructed function at element interfaces must have C^r continuity, and completeness requires that the constructed function within the element must have C^{r+1} continuity (where C^r continuity indicates that the function and its derivatives up to the rth order are continuous). Also note that by this formulation the shape of the element is prescribed by the application of the shape function to the nodal coordinates.

After constructing a candidate function, $\tilde{u}$, how does one guarantee that this equation approximates the differential equation? In the Galerkin approach the method of weighted residuals is used. The residual is an indication of how poorly the sample function $\tilde{u}$ satisfies the differential operator. The residual is obtained by applying the differential operator (as described in equation (A.1)) to the sample function. Thus the residual is defined by

$$R = F(x, y, \cdots, \tilde{u}, \tilde{u}_x, \cdots, \tilde{u}_{xx}, \tilde{u}_{xy}, \cdots). \tag{A.22}$$

Note that if the sample function satisfies the PDE exactly then the residual will be identically zero everywhere in the domain. Since the sample function is constrained to be of a particular form it is unlikely that the residual will be identically zero at all points. In the method of weighted residuals the residual is driven to a small value by requiring that a weighted average of the residual over the domain vanish. Specifically, the m linearly independent weighting functions W_i satisfy the relation

$$\int_{\Omega} R\ W_i\ d\Omega = 0,\ \text{where } i = 1, \cdots, m. \tag{A.23}$$

Since the PDE under approximation will provide a zero residual at all points in the domain, this formulation is applied to each element rather than the entire domain: the element equations are formulated such that

$$\int_{\Omega^{(e)}} R^{(e)}\ W_i^{(e)}\ d\Omega = 0,\ \text{where } i = 1, \cdots, m. \tag{A.24}$$

Furthermore, in the Galerkin method the weighting function is chosen such that $W_i = N_i$. This method, combined with the compatibility and completeness restrictions on the shape functions, defines the basic foundation for many applications of the finite element method.

A.3.1 Integration by Parts

In the application of the method of weighted residuals the resulting integral equations are of a form where a differential operator is multiplied by a weighting function. This product naturally lends itself to the application of integration by parts (the integral counterpart to the product rule). For example, if the differential operator contains some derivative of a function f then the weighted residual formulation can be transformed by integration by parts into the following formulation

$$\int_a^b \frac{\partial f}{\partial x} W_i\ dx = f W_i\big|_a^b - \int_a^b f \frac{\partial W_i}{\partial x} dx. \tag{A.25}$$

The importance of this transformation is that the order of the derivatives is reduced by one. This relaxes the compatibility and completeness restrictions required by the elements, and as such implicitly reduces the order of the polynomial shape functions required to solve the problem. In two and three dimensional problems this integration by parts is satisfied by the Green and Gauss theorems, given by the equation

$$\int_\Omega \left(\vec{\nabla}\cdot\mathbf{f}\right) W_i \; d\Omega = \int_\Gamma W_i \;\left(\mathbf{f}\cdot\mathbf{n}\right) d\Gamma - \int_\Omega \mathbf{f}\cdot\vec{\nabla} W_i \; d\Omega. \tag{A.26}$$

Notice that the selection of $W_i = 1$ reduces this equation to the traditional form of Gauss's divergence theorem that is often applied in finite volume formulations.

A.3.2 Transformation to Local Coordinates

As mentioned earlier, the shape of the element is also derived from the shape functions. That is

$$\vec{x}(\vec{\xi}) = \sum_{i=1}^{m} \vec{x}_i N_i(\vec{\xi}). \tag{A.27}$$

In this formulation the position vector within an element is specified by the local parametric coordinate vector $\vec{\xi}$. Typically the parametric coordinates are defined over the interval $[-1, 1]$ or $[0, 1]$, depending on the type of shape function selected. A two-dimensional representation of this transformation process is illustrated in figure A.3. Typically the shape functions used to describe the element shape are identical to those used to describe the dependent variables within the domain. In these cases the formulation is said to be isoparametric. Generally the shape functions used to identify the parametric transformation to physical space and the shape function used to interpolate dependent variables need not be equivalent. In such cases the mapping is called super-parametric (sub-parametric) if the number of nodes used to generate the transformation function is greater than (less than) the number of nodes used in the dependent function interpolations.

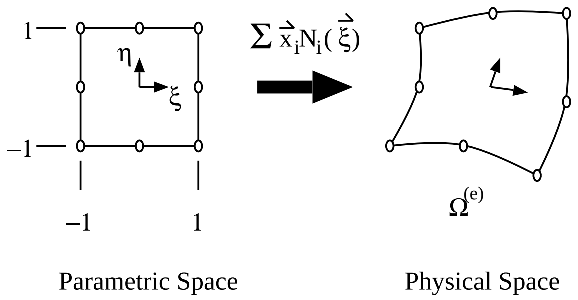


Figure A.3: An example of parametric coordinate transforms in 2-D

An important measure of the transformation space is the Jacobian of the transformation, where the Jacobian $\overline{J}$ is defined by the relation

$$\overline{J}_{ij} = \frac{\partial \xi_i}{\partial x_j}. \tag{A.28}$$

Usually the Jacobian of the inverse transformation is typically used, since the derivatives can be more directly obtained by differentiating the transform function. The Jacobian of the inverse transformation is given by

$$\overline{J}^{-1}_{ij} = \frac{\partial x_i}{\partial \xi_j}. \tag{A.29}$$

These relations can be used to obtain $\frac{\partial \xi_i}{\partial x_j}$ derivatives when needed. Given the Jacobian of the transformation, it is now possible to change the variables of integration to the parametric space using the relation that $|\overline{J}| d\Omega = d\xi_1 d\xi_2 d\xi_3$. Applying this relation one obtains

$$\int_{\Omega^{(e)}} f(\vec{x}) d\Omega = \int_{-1}^{+1} \int_{-1}^{+1} \int_{-1}^{+1} f\left(\vec{x}(\vec{\xi})\right) \frac{1}{|\overline{J}|} d\xi_1 d\xi_2 d\xi_3 \tag{A.30}$$

Finally since the function $f(\vec{x})$ typically contains derivatives of dependent variables with respect to the independent variable $\vec{x}$, a rule must be applied to transform these derivatives to the local parametric coordinates. This is performed by utilizing the following relation, obtained from the application of the chain rule,

$$\vec{\nabla}_x N_i(\vec{x}) = \overline{J} \vec{\nabla}_\xi N_i(\vec{\xi}). \tag{A.31}$$

Overviews of these transformation techniques can be found in a variety of texts on finite volume and finite element methods (*e.g.*, [67] [68]).

A.3.3 Numerical Integration Methods

The preceding section outlined the transformation of the integral equations obtained from the method of weighted residuals into the local parametric coordinates of the element. After this transformation, the resulting integrals will be of the form

$$\int_{-1}^{+1}\int_{-1}^{+1}\int_{-1}^{+1} f(\vec{\xi})d\xi_1 d\xi_2 d\xi_3, \tag{A.32}$$

where $f(\xi)$ is usually a function formed of products of shape functions. Since these functions are typically polynomials, it is possible to integrate them analytically. For situations where these polynomial formulations are too complex to be evaluated efficiently, a numerical integration is used (without loss of accuracy, since the errors indicated by these integrations can be made to be of the same order as the shape functions themselves). The typical approach to numerical integration is Gaussian quadrature, which is represented in one dimension as

$$\int_{-1}^{+1} f(\xi)d\xi = \sum_{i=1}^{n} H_i f(\xi_i), \tag{A.33}$$

where the H_i's are weight coefficients and the ξ_i's are the abscissas of integration. This formulation can be applied in each dimension independently to obtain an equivalent formulation for three dimensional integrals given by

$$\int_{-1}^{+1}\int_{-1}^{+1}\int_{-1}^{+1} f(\xi,\eta,\zeta)d\xi d\eta d\zeta = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} H_i H_j H_k f(\xi_i,\eta_j,\zeta_k). \tag{A.34}$$

One of the more commonly used integration methods is the one point quadrature. In the one point quadrature the function is only sampled at the origin in parametric space.

A.3.4 Element Assembly

Up to this point, integrations over each element have occurred, but an integration over the entire domain is necessary to obtain a solution. Here there are two basic approaches: to integrate the elements about each node to obtain an equation for that node; or to assemble the equations obtained for each element. Integrating over all of the elements that share a particular node will produce equations of a similar form to the finite difference method. That is, each nodal value will be expressed as a function of the neighboring nodal values. As with the finite difference method, if the discretized PDE is linear then these functions will be expressed as linear combinations of the neighboring nodal values. One disadvantage of this approach is that it does not allow much latitude in the run time control over the assembly process. For example, adaptive solvers where the type of shape function used changes as the execution proceeds would not be conveniently implemented using this approach, since the integrations about the nodes are performed *a priori*.

The alternate approach is to integrate over each element and then assemble the resulting equations. This assembly process represents an integration over the entire domain. In a discretized sense, this integral becomes the Riemann sum of the elements over the domain. The element integrations will produce an equation associated with each weighting function used in the method of weighted residuals. The set of equations produced can be represented in matrix form for linear PDE problems (for non-linear problems the resulting equations will need to be linearized in order to obtain a solution). These matrix equations will contain matrices of size $n \times m$ if n is the number of nodes in the element and m is the number of weighting functions. These matrices become components of the overall matrix that will be formed when the global integration is performed. In this case the local element matrices are summed to create the global matrix: coefficients associated with the connection relationship from node i to node j are summed for example. Generally this matrix will need to be inverted in order to obtain the global integration. Although the finite element discretization obtains a linear system in a way similar to the finite difference discretization, the formation of the matrix is more complex, since the element matrices do not form row representations. Instead each element contributes its part to the global matrix.

A.3.5 Lumped Matrices

Consider the integration of the u_t term in the convection/diffusion equation by applying Galerkin's method. Using this approach one obtains the following equation

$$\int_{\Omega^{(e)}} \tilde{u}_t^{(e)} \; N_i^{(e)} \; d\Omega = \sum_{j=1}^{n} \int_{\Omega^{(e)}} (u_t)_j \; N_j^{(e)} \; N_i^{(e)} \; d\Omega, \text{ where } i = 1, \cdots, m. \tag{A.35}$$

This equation can be rewritten in matrix form as $M\mathbf{u_t}$ where M, often identified as the mass matrix, is represented by

$$M_{ij}^{(e)} = \int_{\Omega^{(e)}} N_i^{(e)} \; N_j^{(e)} \; d\Omega. \tag{A.36}$$

The existence of this mass matrix complicates the process of obtaining a time integration of u. An approximation that is often performed is to sum the rows of the mass matrix to the diagonal. The resulting diagonal matrix is called a "lumped" mass matrix. Lumping is a procedure that can be applied to other matrices that occur in the finite element formulation and often leads to a structure in the program where values computed in element integrations are summed to nodes.

A.3.6 Other Details

The discussion of the finite element method thus far has assumed that the dependent variables will be specified on the nodes of the element and that the functional representation of the dependent variable within each element will be determined by a set of shape functions. With this approach, computations center around obtaining the dependent variables at nodes. The integration of element functions over the domain introduces computations that are topologically associated with the element volumes and faces (sometimes edges are also included). These computed values are only intermediate results used to obtain the values of the dependent variables on the nodes. This is not the only perspective on finite element methodologies. Other perspectives may locate some or all of the dependent variables in other locations of the mesh. For example, in many finite element formulations the dependent variables are located at element centers. Similar methodologies will locate different components of the dependent variables at different mesh locations to either increase computational efficiency or obtain local conservation properties. For example, in an incompressible solver it may be advantageous to store pressures at element centers, fluid velocities at element faces, while position vectors are described according to nodes. These variations increase the complexity of the resulting solver and make it more difficult to obtain a generalized formulation. In these cases, dependencies between dependent variables are not as straightforward to represent as the neighborhood set (or stencil) in finite differences.

In addition, it is often the case that operator splitting is employed to make the problem more tractable. For example, in the incompressible flow case it may be necessary to solve the pressure and velocity components independently. When such cases are used a sequence of problems are solved and combined (possibly with iteration) in order to obtain a solution for all dependent variables. Boundary conditions may add additional and somewhat arbitrary constraints and dependencies depending on how they are modeled. For example, in the Baldwin-Lomax turbulence model a minimization is obtained on lines orthogonal to the boundary in order to determine the transition between the inner and outer regions of the turbulent boundary layer. These issues can introduce additional complexity to the process of assembling components of the solver.

Finally, the mesh itself may be represented by a variety of element shapes. For example, hexahedral or prismatic elements may be used near boundaries and the more general tetrahedral element shape may be used in regions where no distinct orientation of the shape functions are found to be advantageous. Examples of some of the element options are shown in figure A.4. Notice that in these different element shapes there are distinctly different relationships between topological elements of nodes, faces, and elements. A usual approach to building solvers in these mixed modes may include collocating like elements. An important advantage of the collocation methods is that they organize the dependency graph into constant valence subgraphs. For example, all hexahedral elements with linear shape functions will depend on eight nodes while all linear tetrahedral elements will depend on four. This view can be important, particularly in obtaining high performance implementations. For example, although it is possible to identify the relationship between nodes and elements directly, this is usually avoided, if possible, since the number of elements that may neighbor a particular node is not fixed.

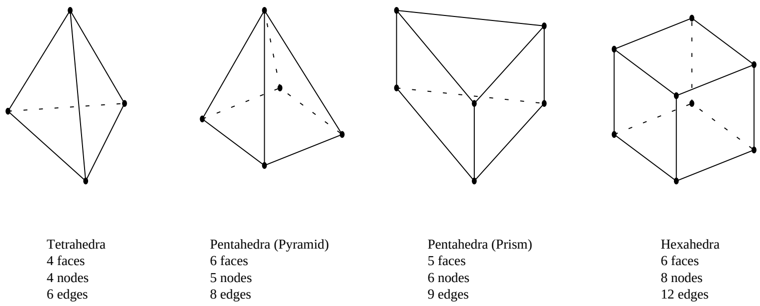


Figure A.4: Various 3-D Element Configurations

APPENDIX B

THERMODYNAMIC AND CHEMISTRY DATA

This appendix includes descriptions of the chemistry models used in the simulations of chapter VII. These list the actual model files input to the simulation software. All values presented in this appendix are in the MKS unit system. Figure B.1 lists the thermodynamic properties of all species used by the chemistry models described in this appendix. The variables in this figure correspond to the thermodynamic variables described in chapter V. Figure B.2 lists the ideally dissociating oxygen model. Notice that this model redefines the thermodynamics for species `O2`. Figure B.3 lists the dissociating oxygen model that includes the full thermodynamic model for `O2`, as listed in figure B.1. And finally, the five species air model of Kang and Dunn is listed in figure B.4. This model uses the thermodynamic properties listed in figure B.1.

```
////////////////////////////////////////////////////////////////////////
// Units:  SI Unit System                                             //
//  m       = atomic mass unit     molecular mass                     //
//  n       = no units             trans & rot contribution factor   //
//  hf      = J/Kmole              Heat of formation                  //
//  href    = J/Kmole              Reference enthalpy                 //
//  sref    = J/Kmole/Ko           Reference entropy                  //
//  Tref    = Ko                   Reference Temperature              //
//  Pref    = Pa                   Reference Pressure                 //
//  theta_v = Ko                   Vibrational Temperature            //
////////////////////////////////////////////////////////////////////////
species = {
// elements
N     = <n=1.5, hf= 4.7104e8> ;
O     = <n=1.5, hf= 2.4699e8> ;
// Reference values
N     : <href= 4.73131e8, sref=153299, Tref=298.0, Pref=101325.0> ;
O     : <href= 2.49374e8, sref=161069, Tref=298.0, Pref=101325.0> ;
// diatomic compounds
N2    = <n=2.5, hf= 0.0,      theta_v=3392.0 > ;
O2    = <n=2.5, hf= 0.0,      theta_v=2273.0 > ;
NO    = <n=2.5, hf= 8.9853e7, theta_v=2739.0 > ;
// Reference Values
N2    : <href= 0.0,       sref=191639, Tref=298.0, Pref=101325.0> ;
O2    : <href= 0.0,       sref=205163, Tref=298.0, Pref=101325.0> ;
NO    : <href= 9.03555e8, sref=210803, Tref=298.0, Pref=101325.0> ;
} ;
```

Figure B.1: Species Thermodynamic Properties

```
// ideal dissociating oxygen model.  Assumes the vibrational component of O2 is
// in the half excited state.
species = {
O2    = <n=3.0, hf= 0.0 > ;
O2    : <href= 0.0,sref=205163, Tref=298.0, Pref=101325.0> ;
O ;
O2    : < mf = 1.0 > ;
} ;
reactions = {
O2 + O2 <-> 2 O + O2 : <Kf = Arrhenius(1.9e18,-1.5,59500.0),
                        Kc = Thermodynamic> ;
O2 + O  <-> 2 O + O  : <Kf = Arrhenius(6.4e20,-2.0,59500.0),
                        Kc = Thermodynamic> ;
} ;
```

Figure B.2: Ideally Dissociating Oxygen Chemisty Model

```
species = {
O  ;
O2 : < mf = 1.0 > ;
} ;

reactions = {
O2 + O2 <-> 2 O + O2 : <Kf = Arrhenius(1.9e18,-1.5,59500.0),
                        Kc = Thermodynamic> ;
O2 + O  <-> 2 O + O  : <Kf = Arrhenius(6.4e20,-2.0,59500.0),
                        Kc = Thermodynamic> ;
} ;
```

Figure B.3: Dissociating Oxygen Chemistry Model (Uses default O2 thermodynamics)

```
// Five species air Model of Kang and Dunn
//
species = {
O2 : <mf = 0.22 > ;
N2 : <mf = 0.78 > ;
NO ;
O ;
N ;
} ;

reactions = {
O2 + O  <-> O + O + O   = <Kf = Arrhenius(9.000e16,  -1.0, 5.95e4)> ;
O2 + O2 <-> O + O + O2  = <Kf = Arrhenius(3.240e16,  -1.0, 5.95e4)> ;
O2 + N  <-> O + O + N   = <Kf = Arrhenius(3.600e15,  -1.0, 5.95e4)> ;
O2 + N2 <-> O + O + N2  = <Kf = Arrhenius(7.200e15,  -1.0, 5.95e4)> ;
O2 + NO <-> O + O + NO  = <Kf = Arrhenius(3.600e15,  -1.0, 5.95e4)> ;

N2 + M  <-> N + N + M   = <Kf = Arrhenius(1.900e14,  -0.5, 1.13e5)> ;
N2 + N  <-> N + N + N   = <Kf = Arrhenius(4.085e19,  -1.5, 1.13e5)> ;
N2 + N2 <-> N + N + N2  = <Kf = Arrhenius(4.700e14,  -0.5, 1.13e5)> ;

NO + M  <-> N + O  + M  = <Kf = Arrhenius(7.800e18,  -1.5, 7.55e4)> ;
NO + O2 <-> N + O  + O2 = <Kf = Arrhenius(3.900e17,  -1.5, 7.55e4)> ;
NO + N2 <-> N + O  + N2 = <Kf = Arrhenius(3.900e17,  -1.5, 7.55e4)> ;

NO + O  <-> O2 + N      = <Kf = Arrhenius(3.200e06,   1.0, 1.97e4)> ;
N2 + O  <-> NO + N      = <Kf = Arrhenius(7.000e10,   0.0, 3.80e4)> ;
} ;
```

Figure B.4: Five Species Air Model of Kang and Dunn

REFERENCES

[1] T. Sterling, D. Becker, D. Savarse, J. Dorband, U. Ranawak, and C. Packer, "Beowulf: A parallel workstation for scientific computation," in *Proceedings of the 1995 International Conference on Parallel Processing*, vol. 1, pp. 11–14, ICPP, August 1995.

[2] F. P. Brooks, "No silver bullet," in *Proceedings of the IFIP Tenth World Computing Conference* (H. J. Kugler, ed.), pp. 1069–1076, Elsevier Science, 1986.

[3] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*. MIT Press, 2nd ed., 1998.

[4] E. Dijkstra, "Cooperating sequential processes," in *Programming Languages* (F. Genuys, ed.), Academic Press, 1965.

[5] D. Buttenhof, *Programming with POSIX Threads*. Addison Wesley, 1997.

[6] OpenMP Architecture Review Board , "OpenMP Fortran application program interface," October 1997. Version 1.0.

[7] M. Metcalf and J. Ried, *Fortran 90 Explained*. Oxford University Press, 1992.

[8] High Performance Fortran Forum , "HPF language specification," January 1997. Version 2.0.

[9] C. Lin and L. Snyder, "ZPL: An array sublanguage," in *Languages and Compilers for Parallel Computing* (U. Banerjee and D. Gelernter, eds.), pp. 96–114, Springer-Verlag, 1993.

[10] G. Alverson, W. Griswold, C. Lin, and L. Snyder, "Abstractions for portable, scalable parallel programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, pp. 72–86, January 1998.

[11] L. G. Valiant, "A bridging model for parallel computation.," *Communications of the Association for Computing Machinery*, vol. 33, pp. 103–111, August 1990.

[12] J. Hill, B. McColl, D. Stefanescu, M. Goudreau, K. Lang, S. Rao, T. Suel, T. Tsantilas, and R. Bisseling, "BSPlib: The BSP programming library," *Parallel Computing*, vol. 24, no. 14, pp. 1947–1980, 1998.

[13] R. Jagannathan, "Dataflow models," in *Parallel and Distributed Computing Handbook* (E. Zornaya, ed.), McGraw-Hill, 1995.

[14] B. Ang, A. Carlo, S. Glim, and A. Shaw, "An introduction to the ID compiler," Tech. Rep. Computation Structures Group Memo 329, Massachusetts Institute of Technology Laboratory for Computer Science, May 1991.

[15] P. Hudak, "Report on the programming language Haskell: A non-strict, purely functional language version 1.2," Tech. Rep. R1-R164, SIGPLAN Notices, 1992.

[16] S. K. Skedzielewski, P. J. Miller, J. T. Feo, and S. M. Denton, *SISAL 90 User's Guide*. Lawrence Livermore National Laboratory, 1991.

[17] D. C. DiNucci, *A Formal Model For Architecture-Independent Parallel Software Engineering*. PhD thesis, Oregon Graduate Institute, March 1990.

[18] N. Carriero and D. Gelernter, *How to write parallel programs: A first course*. MIT Press, 1990.

[19] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide: Second Edition*. SIAM, 1995.

[20] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object-oriented numerical software libraries," in *Modern Software Tools in Scientific Computing* (E. Arge, A. M. Bruaset, and H. P. Langtangen, eds.), pp. 163–202, Birkhauser Press, 1997.

[21] R. Das, M. Uysal, J. Saltz, and Y. S. Hwang, "Communication optimizations for irregular scientific computations on distributed memory architectures," *Journal of Parallel and Distributed Computing*, vol. 22, pp. 462–479, September 1994.

[22] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T. Williams, "Array design and expression evaluation in POOMA II," in *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science, ISCOPE, Springer-Verlag, 1998.

[23] T. L. Veldhuizen, "Arrays in Blitz++," in *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science, ISCOPE, Springer-Verlag, 1998.

[24] S. R. Kohn, *A Parallel Software Infrastructure for Dynamic Block-Irregular Scientific Calculations*. PhD thesis, University of California, San Diego, 1995.

[25] M. Parashar and J. Browne, "System engineering for high performance computing software: The HDDA/DAGH infrastructure for implementation of parallel structured adaptive mesh refinement," in *Structured Adaptive Mesh Refinement Grid Methods*, IMA Volumes in Mathematics and its Applications, Springer-Verlag, 1998.

[26] H. Sagan, *Space-Filling Curves*. Springer-Verlag, 1994.

[27] P. Hilfinger and P. Colella, "FIDIL: A language for scientific programming," in *Symbolic Computation: Applications to Scientific Computing* (R. Grossman, ed.), pp. 97–138, SIAM, 1989.

[28] I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming*. Prentice-Hall, 1990.

[29] I. Foster, R. Olson, and S. Tuecke, "Productive parallel programming: The PCN approach," *Scientific Programming*, vol. 1, pp. 51–66, 1992.

[30] I. Foster, "Compositional parallel programming languages," *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 111–134, January 1999.

[31] R. S. Palmer, "Chain models and finite element analysis: An executable chains formulation of plane stress," *Computer Aided Geometric Design*, vol. 12, pp. 733–770, 1995.

[32] K. Birken, "Towards automatic parallelization of numerical software based on formal specification concepts," in *Concepts of Numerical Software* (W. Hackbusch and G. Wittum, eds.), Notes on Numerical Fluid Mechanics, Vieweg Verlag, 1998.

[33] K. Birken, "Semi-automatic parallelization of dynamic, graph-based applications.," in *Parallel Computing: Fundamentals, Applications and New Directions.* (E. D'Hollander, G. Joubert, F. Peters, and U. Trottenberg, eds.), Elsevier Science, 1998.

[34] W. Clocksin and S. Mellish, *Programming in Prolog.* Springer-Verlag, 1987.

[35] L. Sterling and E. Shapiro, *The Art of Prolog.* MIT Press, 1986.

[36] R. S. Bird and L. Meertens, "Two exercises found in a book on algorithmics," in *Program Specification and Transformation* (L. G. L. T. Meertens, ed.), pp. pp 451–457, North-Holland, 1987.

[37] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* MIT Press, 1990.

[38] Z. Warsi, *Fluid Dynamics: Theoretical and Computational Approaches.* CRC Press, 1993.

[39] W. G. Vincenti and C. H. Kruger, *Introduction to Physical Gas Dynamics.* Krieger Publishing Company, 1965. Reprinted 1986.

[40] C. Cox, *An Efficient Solver for Flows in Local Chemical Equilibrium.* PhD thesis, Mississippi State University, December 1992.

[41] A. G. Hansen, "Generalized control volume analyses with application to the basic laws of mechanics and thermodynamics," *Bulletin of Mechanical Engineering Education*, vol. 4, pp. 161–168, 1965.

[42] P. D. Thomas and C. K. Lombard, "Geometric conservation law and its application to flow computations on moving grids," *AIAA Journal*, vol. 17, no. 10, pp. 1030–1037, 1978.

[43] J. Janus, *Advanced 3-D Algorithm for Turbomachinery.* PhD thesis, Mississippi State University, May 1989.

[44] P. Cinnella, *Flux-Split Algorithms for Flows with Non-Equilibrium.* PhD thesis, Virginia Polytechnic Institute and State University, December 1989.

[45] R. Walters, P. Cinnella, D. Slack, and D. Halt, "Characteristic-based algorithms for flows in thermochemical nonequilibrium," *AIAA Journal*, vol. 30, pp. 1304–1313, May 1992.

[46] A. Harten and J. M. Hyman, "Self-adjusting grid methods for one-dimensional hyperbolic conservation laws," *Journal of Computers in Physics*, vol. 50, pp. 235–269, 1983.

[47] R. J. LeVeque, *Numerical Methods for Conservation Laws.* Birkhäuser, 1992.

[48] J. Quirk, "A contribution to the great riemann debate," tech. rep., ICASE, 1992. ICASE Report No. 92-64.

[49] B. Einfeldt, "On Godunov-type methods for gas dynamics," *SIAM Journal on Numerical Analysis*, vol. 25, pp. 294–318, April 1988.

[50] B. V. Leer, "Towards the ultimate conservative difference scheme. V. A second order sequel to Godunov's method.," *Journal of Computational Physics*, vol. 32, pp. 101–136, 1979.

[51] C. Hirsch, *Numerical Computation of Internal and External Flows: Volume 2 Computational Methods for Inviscid and Viscous Flows.* John Wiley & Sons, 1988.

[52] B. V. Leer, "Towards the ultimate conservative difference scheme. II. Monotinicity and conservation combined in a second order scheme.," *Journal of Computational Physics*, vol. 32, pp. 101–136, 1979.

[53] T. J. Barth and D. C. Jespersen, "The design and application of upwind schemes on unstructured meshes," tech. rep., AIAA, 1989. AIAA 89-0366.

[54] V. Venkatakrishnan, "On the accuracy of limiters and convergence to steady state solutions," tech. rep., AIAA, 1993. AIAA 93-0880.

[55] G. H. Golub and C. F. V. Loan, *Matrix Computations: Second Edition.* Johns Hopkins University Press, 1989.

[56] E. Barszcz, R. Fatoohi, V. Venkatakrishnan, and S. Weeratunga, "Solution of regular, sparse triangular linear systems on vector and distributed-memory multicomputers," tech. rep., NASA Ames Research Center, Report RNR-93-007, NASA Ames Research Center, Moffett Field CA 94035, April 1993.

[57] S. W. Kang and M. G. Dunn, "Theoretical and measured electron-density distributions for the ram vehicle at high altitude," tech. rep., AIAA, 1972. AIAA Paper No. 72-689.

[58] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: Second Edition.* Cambridge University Press, 1992.

[59] W. Westmoreland, "Numerical solution of the euler equations with nonequilibrium chemistry," Master's thesis, Mississippi State University, May 1997.

[60] C. Cox and P. Cinnella, "General solution procedure for flows in local chemical equilibrium," *AIAA Journal*, vol. 32, pp. 519–527, March 1994.

[61] E. H. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *proceedings 24th National Conference Association of Computing Machines*, pp. 167–172, ACM, ACM Publications, 1969.

[62] J. W. Thomas, *Numerical Partial Differential Equations: Finite Difference Methods.* Springer-Verlag, 1995.

[63] T. J. Oden and J. N. Reddy, *An Introduction to the Mathematical Theory of Finite Elements.* New York: John Wiley, 1976.

[64] S. A. Orszag, "Spectral methods for problems in complex geometries," *Journal of Computational Physics*, vol. 37, pp. 70–92, 1980.

[65] G. A. Bird, *Molecular Gas Dynamics and the Direct Simulation of Gas Flows.* Clarendon Press, Oxford, 1994.

[66] U. Frisch, B. Hasslacher, and Y. Pomeau, "Lattice-gas automata for the Navier-Stokes equation," *Physical Review Letters*, vol. 56, pp. 1505–1508, 1986.

[67] C. Hirsch, *Numerical Computation of Internal and External Flows: Volume 1 Fundamentals of Numerical Discretization.* John Wiley & Sons, 1988.

[68] K. Huebner, E. Thornton, and T. Byrom, *The Finite Element Method for Engineers.* John Wiley & Sons, 1995.