INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road. Ann Arbor. MI 48106-1346 USA
313 761-4700 800 521-0600

Order Number 9219245

An interactive programming environment for solving partial differential equations using adaptive grids (with application to flame-flow interaction problems)

Kozlovsky, Gregory, Ph.D.

City University of New York, 1992

Copyright ©1992 by Kozlovsky, Gregory. All rights reserved.

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A

An Interactive Programming Environment for Solving Partial Differential Equations Using Adaptive Grids (with application to flame-flow interaction problems)

by

Gregory Kozlovsky

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1992

©1992
Gregory Kozlovsky
All Rights Reserved

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

12-18-91

Date

Professor Octavio Betancourt
Chair of Examining Committee

12-30-91

Date

Professor Stanley Habib
Executive Officer

Professor Michael Anshel
Professor Gregory Sivashinsky
Professor Michael Vulis

Supervisory Committee

The City University of New York

Abstract

An Interactive Programming Environment for Solving Partial Differential Equations Using Adaptive Grids (with application to flame-flow interaction problems)

by

Gregory Kozlovsky

Advisers: Professors Octavio Betancourt and Gregory Sivashinsky

In this work we investigate software tools and numerical methods for modeling physical phenomena with high local resolution. The following interrelated issues are addressed: adaptive grids, formulation of the finite element method on adaptive grids, and programming environments for numerical software development and scientific visualization.

We introduce a class of adaptive grids called *consistent recursive grids* and study its properties. Algorithms for refinement, unrefinement, grid traversal, and nearest node search are developed. Data structures for computer implementation of such grids are proposed.

A version of the finite element method specifically designed for consistent recursive grids is described. This version uses discontinuous trial functions in order to simplify matrix generation on interfaces of computational cells of different sizes.

We identify shortcomings of existing tools for numerical software development and

introduce the concept of ***geometric debugging***. We present an interactive programming environment which provides a way to connect geometrically based understanding with underlying data structures. The programming environment includes an integrated library of functions for dynamic recursive grids and an integrated interactive graphics module.

Finally, we address application of the developed software to flame-flow interaction problems of interest to combustion scientists, and present computational results.

Acknowledgements

I would like to thank my advisers Professors Octavio Betancourt and Gregory Sivashinsky for their advice and moral support, and the Executive Officer of the Ph.D. program at CUNY Professor Thomas Wesselcamper together with his Administrative Assistant Joseph Driscoll for helping me to beat the system. I gratefully acknowledge financial support from the US Department of Energy under grant DE-FG02-88ER13822 and from the National Science Foundation under grant CTS-8910903.

My special gratitude to Dr. Vardy Amdursky from the IBM Scientific Center in Haifa, Israel, who discovered a potential numerical analyst in me and helped me to become one, and to members of his group — Ilan Efrat and Boris Bachelis.

I would like to thank my friends Ruben and Joyce Michel, Bernard Phlips, Sarah Coles, Heather Reynolds, Reesa Gringorten, and Richard Ferguson for helping me during hard times.

Special appreciation goes to a scientifically oriented beauty with quick dark eyes who provided me with the amount of unhappiness necessary to be creative.

Contents

1 Introduction **1**

2 Adaptive Grids **5**

2.1 Definitions and terminology 9

2.2 Properties of consistent grids 14

2.3 Grid data structures 16

2.4 Grid creation, refinement, and unrefinement 18

2.5 Grid traversal and subsets 22

2.6 Search for neighbours 22

3 The finite element method on consistent recursive grids **24**

3.1 Basis functions in space 24

3.2 Weight functions 26

3.3 Upstream correction 26

3.4 Preassembling stiffness matrices 28

4 The interpreter **29**

4.1 Interpreter input language 29

4.1.1 Lexical conventions 31
4.1.2 Fundamental data types 31
4.1.3 Derived data types 32
4.1.4 Polymorphic arguments 33
4.1.5 Conversions 33
4.1.6 Expressions 33
4.1.7 Declarations 33
4.1.8 Statements 34
4.1.9 External variables 34
4.1.10 Function declarations 34

4.2 Script files 35
4.3 Interpreter system functions 36
4.4 Interface between compiled C functions and interpreter data space . . 37
4.5 Problem-specific objects 39
4.6 Functions for handling index lists 41
4.6.1 Creating and changing lists 41
4.6.2 Walking through index lists 42
4.7 Save/restore facility 43

5 The adaptive grid library **45**

5.1 Specifying directions, orientations, and positions 45
5.2 Grid-specific data types 47
5.3 External variables 47

5.3.1 Grid current status information 48

5.3.2 Refine/unrefine function interface 49

5.3.3 Refine/unrefine decision interface 50

5.4 Grid management functions 51

5.5 Grid traversal 53

5.5.1 Grid traversal using the tree structure 54

5.5.2 Grid traversal using internal supercell lists 55

5.6 Search for neighbours 56

5.6.1 In-line functions to find connected nodes and supercells 56

5.6.2 Finding neighbouring nodes 57

5.6.3 Location functions 58

5.7 Saving/restoring the grid 59

6 The standard tool library **60**

6.1 Grid node coordinates 60

6.2 Node and supercell lists 61

6.3 Standard refinement and unrefinement rules 64

6.4 Bilinear approximation utilities 64

6.4.1 General purpose algebraic functions 65

6.4.2 Grid vector utilities 66

6.5 Preassembling stiffness matrices 66

6.5.1 External variables 67

6.5.2 Functions 69

7 Visualization tools **72**

7.1 Color spaces 72

7.2 Coordinate systems 74

7.3 Grid function display 74

7.4 Cross-sections 76

7.5 External variables 76

7.6 Visualization functions 78

7.7 Postscript printing 79

7.8 The interactive graphics module 81

7.8.1 Input processing 81

7.8.2 Space functions 83

7.8.3 Main menu 85

7.8.4 Drawing menu 87

7.8.5 Settings menu 89

7.8.6 Grid menu 89

7.8.7 Cross-section menu 90

8 Model problems **93**

8.1 Bunsen burner 93

8.1.1 Mathematical model 95

8.1.2 Discretization 97

8.1.3 Time step 99

8.1.4 Grid refinement criteria 99

8.1.5 Numerical experiments 99

8.1.6 Discussion 101

8.2 Flame propagation in shear flow 120

8.2.1 Problem formulation 120

8.2.2 Flame front velocity 122

8.2.3 Numerical experiments 123

A Implementation example: Bunsen burner model **128**

A.1 Implementation notes 129

A.2 Compiled C modules 129

A.3 Interpreter script 132

Bibliography **134**

List of Figures

2.1 Grid fragment. 10

2.2 Supercell and node types. 11

2.3 Consistency. 12

2.4 Location of neighbouring nodes relative to the central node. 13

2.5 Non-trivial geometrical configurations around a grid node. 16

2.6 Node and supercell structures. 18

2.7 One-cell consistent refinement algorithm. 20

2.8 One-supercell consistent unrefinement algorithm. 20

2.9 Optimal grid adaptation algorithm. 21

4.1 Functional chart of the environment. 30

8.1 Bunsen burner configuration. 96

8.2 Zoom of automatically generated grid for the axisymmetric Bunsen burner case. 103

8.3 Temperature distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 1$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 104

8.4 Concentration distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 1$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 105

8.5 Reaction rate distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 1$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 106

8.6 Temperature (T), concentration (C), and reaction rate (Ω) profiles for the axisymmetric Bunsen burner at $r = 0$ (a) and $r = 7.5$ (b) ($\nu = 1$, $\alpha = \pi/6$, $Le = 1$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 107

8.7 Temperature distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 1.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 108

8.8 Concentration distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 1.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 109

8.9 Reaction rate distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 1.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 110

8.10 Temperature (T), concentration (C), and reaction rate (Ω) profiles for the axisymmetric Bunsen burner at $r = 0$ (a) and $r = 7.5$ (b) ($\nu = 1$, $\alpha = \pi/6$, $Le = 1.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 111

8.11 Temperature distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 112

8.12 Concentration distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 113

8.13 Reaction rate distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 114

8.14 Temperature (T), concentration (C), and reaction rate (Ω) profiles for the axisymmetric Bunsen burner at $r = 0$ (a) and $r = 7.5$ (b) ($\nu = 1$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 115

8.15 Temperature distribution for the slot Bunsen burner ($\nu = 0$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 116

8.16 Concentration distribution for the slot Bunsen burner ($\nu = 0$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 117

8.17 Reaction rate distribution for the slot Bunsen burner ($\nu = 0$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 118

8.18 Temperature (T), concentration (C), and reaction rate (Ω) profiles for the slot Bunsen burner at $r = 0$ (a) and $r = 7.5$ (b) ($\nu = 0$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). 119

8.19 Flame propagation in shear flow configuration. 120

8.20 Contours of temperature distribution for the shear flow ($Le = 1.5$, $0 \leq x \leq 32$, $0 \leq y \leq 20$) with superimposed adaptive grid. 125

8.21 Concentration distribution for the shear flow ($Le = 1.5$, $0 \leq x \leq 32$, $0 \leq y \leq 20$). 126

8.22 Reaction rate distribution for the shear flow ($Le = 1.5$, $0 \leq x \leq 32$, $0 \leq y \leq 20$). 127

Chapter 1

Introduction

Numerical modeling of physical phenomena involves much more than just programming the equations. The most time-consuming and demanding work starts after bugs causing compilation and run-time errors are corrected. The obtained solution must be verified and if, as it almost always happens, the solution is wrong, the remaining bugs must be found and eliminated. These bugs may be either programming errors, or conceptual errors in the discretization of the equations or in the solution process. Human understanding of an error in the solution is most often based on spatial perception. The origins of the error must be traced to the code through numerical data. We shall refer to this part of the numerical software development process as *geometric debugging.*

The simplest solution — printing numerical data using standard output operators — is far from satisfactory. The volume of data can be very large and a lot of time can be wasted on finding and interpreting a few relevant numbers. It is often unclear in advance which part of the data have to be examined, and at what stage of the

computation. Therefore, the traditional way of developing numerical software using a compiled language, such as Fortran or C, requires frequent recompilations after very small changes in the output operators and/or the solution process. Moreover, numerous utilities may have to be written in order to input parameters of the problem, save/restore solutions, output the data in more palatable way, and help examine them. In our experience, only a relatively small fraction of code written to model a physical phenomena is problem-specific numerical code. The situation becomes even worse when the data structures are complex, as is the case with adaptive grids. In our opinion, it would hardly be possible to develop numerical solvers based on adaptive grids without well-designed software tools.

As a solution for this problem we propose an interactive programming environment which provides a way to connect geometrically based understanding with underlying data structures. The proposed programming environment does not attempt to solve problems for the user; rather it is an integrated set of tools which, as we hope, would drastically reduce the amount of technical, non-problem-specific programming and provide facilities for interactive geometric debugging. We hope that this approach (undoubtedly inspired by the C language) will be both general and efficient. Higher level domain-specific software packages can be written on top of this environment.

The core of the interactive programming environment consists of three parts — the interpreter, the interactive graphics module, and the library of adaptive grid functions — which are closely integrated. Additional functionality is provided by the library of standard tools. The user should write numerically intensive computational functions in Fortran or C and link them with the interpreter. User-written programs can access

the interpreter data space. With this approach we can combine the convenience of an interpreter with the efficiency of a compiled language.

The input language of the interpreter is a subset of C with some additions. These additions include grid objects, grid vectors, and space functions. Grid objects automatically change with the grid refinement and unrefinement. Grid vectors and space functions have special meanings for the interactive graphics module.

The main functions of the interactive graphics are grid vector display, and the computational domain scanning using the mouse. *Scan vectors* mode shows the values of all the grid vectors at the current cursor location in a pop-up window. New values can be assigned to the grid vectors by simply overtyping the old values. *Scan function* mode allows the user to see the output of the chosen space function at the current cursor location. Space functions are user-supplied functions which can use the following information as their input: the coordinates of the current point in the computational domain, which corresponds to the current cursor location; the index of the top-level node closest to the current point; and the index of the grid cell containing the current point. In addition, space functions as well as any other user-supplied functions can access the interpreter variables and make any necessary computations. Space functions can print their output in a pop-up window using special operators. Their principal use is for geometrical debugging.

The adaptive grid library works with the class of recursive grids called *consistent* grids. Consistent grids are in a sense the most regular of irregular grids. Consistency simplifies grid data bookkeeping, derivation, and programming of numerical methods. The functions of the library create grids; refine and unrefine them, automatically

maintaining consistency; provide access to grid nodes and cells; and find neighbours for a given node or cell.

The library of standard tools provides additional functionality. It includes, among others, functions which maintain lists of grid nodes and grid cells of different geometrical types, and functions which help to pre-assemble finite element matrices for recursive consistent grids.

Chapter 2

Adaptive Grids

When solving partial differential equations, it is often necessary to resolve small areas with high activity.[1] Flame fronts are an example of such areas. High local activity can also be found in boundary layers. It may be not computationally feasible to use grids which are uniformly spaced throughout the computational domain to resolve these small high activity areas. One increasingly common solution in this situation is the use of adaptive non-uniform grids (Benkhaldoun et al., 1988, Berger & Oliger, 1984, Dannenhofer & Baron, 1986, Ewing, 1986, Rheinboldt & Mesztenyi, 1980). Adaptive non-uniform grids have different densities in different parts of the computational domain, as dictated by requirements of the accuracy of the numerical method. Because of the potential that adaptive gridding has for reducing computational costs, it is one of the forefront areas in computational physics.

Many types of non-uniform grids are used in numerical computing. In general, the choice of a grid type represents a trade-off between simplicity of the grid and its ability to cover the computational domain economically with the locally required resolution.

[1] What exactly constitutes *high activity* is problem dependent. It may mean a high gradient, a large second derivative of the solution, or something else.

In deciding which type of grid to employ for a given problem, much depends on the geometrical configuration of the area(s) in need of high resolution. In many cases, relatively simple grids, such as tensor-product grids or composite uniform grids, can do the job efficiently. We will describe, without the pretence of being comprehensive or rigorous, several typical situations where relatively uncomplicated grids can be used. For simplicity, we restrict our exposition to two-dimensional case.

The high activity area is centered around a singular point. A radial grid is formed by the intersection of circles centered at the point with rays emanating from it. A radial grid is actually a tensor product grid and, as such, can be implemented using simple data structures. The density of such a grid naturally increases with proximity to the central point.

The high activity area is concentrated around a line. Non-uniform tensor product grids are well suited to this case. By suitable mapping of the computational domain the case of a simple curve can be accommodated.

The high activity area can be covered by a rectangle of the size comparable to that of the high activity area. Composite uniform grids can be employed as well as non-uniform tensor product grids. The advantages of the former should be weighed against the increase in the complexity of computer implementation.

In the case of flame-flow interaction problems, the shape of the high activity area is usually more complicated than in the cases outlined above. In addition, in order to be efficient the above approaches should rely on certain geometrical shapes of the area to be refined, and a "black-box" solver must incorporate some sort of pattern recognition ability, a significant complication to be avoided.

Another important consideration is the ability of the grid to change with the solution. In the simplest application of non-uniform gridding, a variably spaced grid is established initially and then held fixed through the course of the calculation. This approach to grid generation is used in many industrial finite element packages. It requires *a priori* knowledge of the general behaviour of the solution (usually an intuition of an experienced engineer) and may require many months of labour. Dynamic adaptive gridding methods, on the other hand, change the grid during the solution process according to the accumulated information. These methods refine the grid in the high activity areas, leave the grid coarse in the areas of little activity, and unrefine the grid in the areas where refinement is no longer needed. Automatic mesh refinement and unrefinement is absolutely necessary in case of time-dependent problems with moving areas of high activity, or when the location of the areas requiring high resolution is not known in advance. For flame-flow interaction problems, which are highly nonlinear, a large number of time steps have to be performed. A desirable property of a dynamic grid in this case is that a small change in the solution causes a small change in the grid. Tensor-product grids and, in general, composite uniform grids do not possess this property (at least when uniform subgrids are large).

For a numerical solution of flame-flow interaction problems, adaptive grids should satisfy the following requirements:

- Handle areas with complex geometry
- Have the ability to move refined area(s)
- Be general enough to serve as a "black box"
- Allow for high degree of local refinement

We choose to use *consistent recursive grids* which satisfy all of the above requirements. Except of restrictions imposed by the requirement of *consistency*, each computational cell of a recursive grid can be subdivided into four subcells independently of the other cells. We call a recursive grid *consistent* if the ratio between the linear sizes of the adjacent computational cells does not exceed two.

Consistent recursive grids have many valuable properties.[2] They are "tight" in the sense that the number of nodes which are not needed for the required local resolution but have to be present because of the grid structure is small. Furthermore, a local change in the solution causes a local change in the recursive grid. In addition, recursive consistent grids have a certain regularity in that the number of possible grid configurations around a grid node is small. Consistency simplifies grid data book-keeping, derivation, and programming of numerical methods.

Consistent dynamic recursive grids require complicated data structures and grid algorithms. They are clearly more difficult to implement than adaptive tensor-product grids. The implementation of numerical methods on consistent recursive grids may be non-trivial. In return, the number of nodes and cells can be reduced, compared to less "tight" grids, causing a corresponding reduction in computational effort and memory requirements. The ease of adapting the grid to small changes in the solution can also be an advantage. On the other hand, tensor-product and uniform composite grids provide better opportunity for vectorization. It is far from clear, at this point, which grids are preferable for which type of problems.

---

[2]Some of these properties will be quantified in subsequent sections.

Will massively parallel computers make adaptive grids obsolete? Unlikely, for we will never be able to model the whole universe at once. The age-old approach of concentrating our attention on one object and taking into account its interaction with the environment using simpler models will stay with us. Adaptive gridding can be viewed as a particular application of this approach.

In our work, for consistent recursive grids we designed data structures, developed and implemented algorithms for refinement, unrefinement, grid traversal, and nearest node search. The implementation was done in the form of a general purpose library of grid handling functions written in C. Our exposition here is restricted to two-dimensional grids, although the approach presented in the following sections can be generalized to three-dimensional grids.

2.1 Definitions and terminology

The main unit for building our grids is a *supercell*. Geometrically, a supercell is a rectangle divided by lines parallel to its sides into four equal subrectangles. Each of these subrectangles we will call a *grid cell* or *computational cell*. Grid nodes are formed by the vertices of the cells. Nine nodes are *associated* with each supercell.

A grid starts its life as a *root grid*, which consists of a number of non-overlapping supercells of equal size completely covering a rectangle.[3] An elementary grid refinement is performed by creating a supercell exactly covering an existing cell (*refining*

---

[3] We are not concerned with more elaborate root grids, because a root grid is inexpensive and a non-rectangular computational domain can be economically covered using appropriate grid refinements.

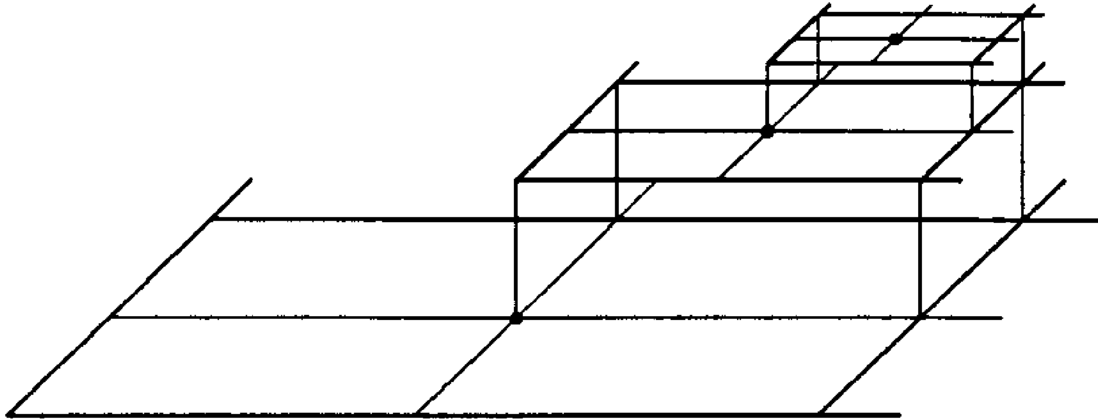


Figure 2.1: Grid fragment.

the cell). The refined cell is called the *parent* cell of the new supercell, which, in turn, is called its *descendant*. Refinement can later be reversed by destroying a previously created supercell (*unrefining the cell*).

Each supercell has a *grid level* number associated with it and its cells. The grid level of a root supercell is zero. The grid level of a non-root supercell is equal to the level of its parental cell plus one.

Nodes associated with cells on different grid levels and having identical coordinates are considered distinct and are called *co-located nodes*. A set of nodes co-located with a given node includes the node itself. There is only one node with the given coordinates on any grid level and, therefore, a node may be associated with more than one supercell on the same level. The grid level of a node is equal to the grid level of its associated supercells.

A *terminal cell* is a cell which does not have the descendant supercell. A *terminal supercell* has all four of its subcells terminal. An *active supercell* contains at least one terminal cell. A *terminal node* is a node which is not co-located with a node on a

higher level.

It will be useful to classify the nodes by their geometrical location in an associated supercell:

x-node - node in the center of a supercell

c-node - node in one of the corners of a supercell

t-node - node on a side of a supercell which is associated exclusively with this supercell (no adjacent supercell on the same grid level)

s-node - located as t-node but is associated with two adjacent supercells

Node types are illustrated in Figure 2.2.

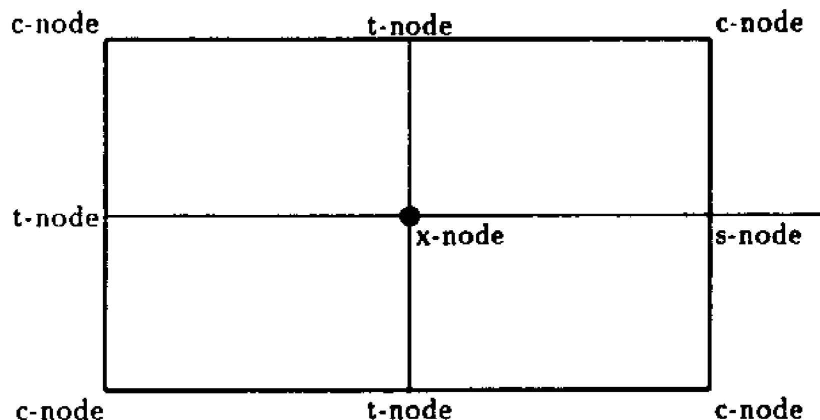


Figure 2.2: Supercell and node types.

Another important distinction is between *internal* cells and nodes, and *border* cells and nodes which are the cells adjacent to and the nodes located on the border of the computational domain. At this point, we can introduce an additional class of nodes: r-node — any node which is not an internal t-node. The name r-node stands for *regular node*.

The difference between x-nodes, c-nodes, s-nodes, and t-nodes is important from the point of view of grid data structures and basic grid algorithms. For numerical

algorithms x-nodes, c-nodes, and s-nodes are all the same, except that, as we will see later, there can be more variety in the configuration of the grid around c-nodes. However, numerical methods have to be formulated differently on the r-nodes and on the internal t-nodes.

Two grid cells are called *neighbours* or adjacent cells if they share at least one pair of co-located nodes. A pair of grid nodes are *neighbours on level k* if both of them have co-located nodes which are associated with the same cell of level k. Two terminal nodes are called *neighbours* if both nodes are co-located with nodes associated with a terminal cell.

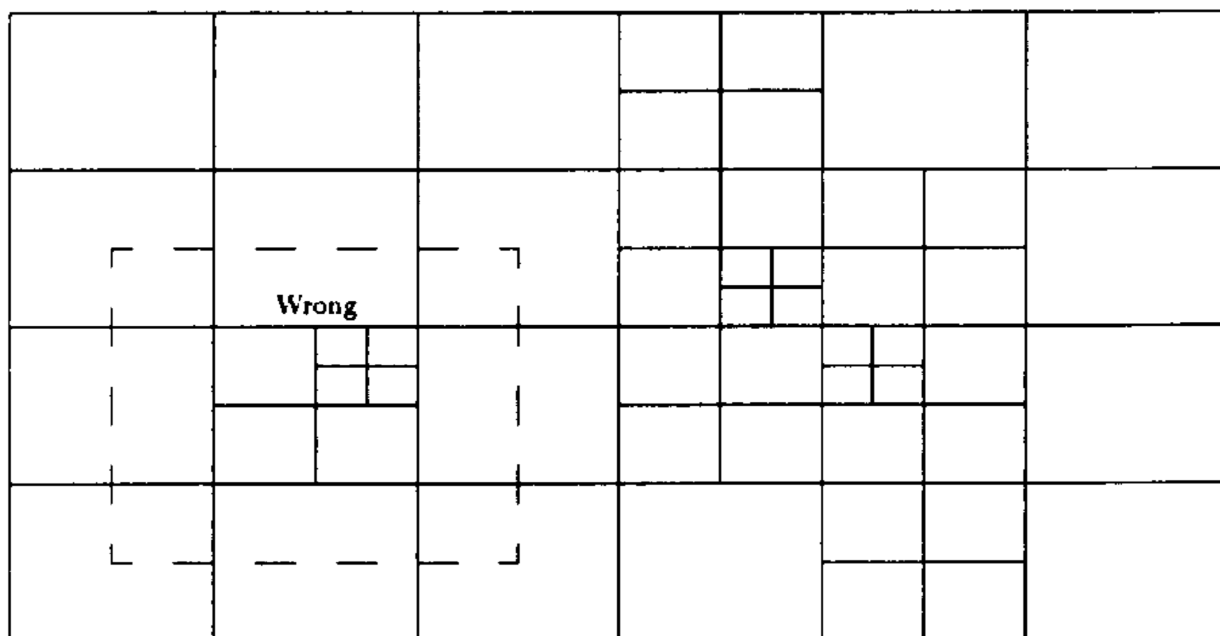


Figure 2.3: Consistency.

A grid is called *consistent* if the linear size of any adjacent terminal grid cells differs by at most a factor of two. Consistency is important from several points of view. Consistent grids are, in a sense, the most regular of irregular grids. The

number of possible local geometric configurations around every terminal node or cell is small. This simplifies grid data structures and basic grid algorithms (refinement, unrefinement, grid traversal, and nearest node search). Consistency also simplifies derivation and programming of numerical methods. In addition, numerical analysts believe that numerical methods "do not like" abrupt changes in the grid size.

A grid can be represented as a forest of quad-trees, with supercells as tree nodes. This tree structure allows for certain important operations to be performed fast. Unfortunately, grid nodes do not form a tree. Therefore, even when we use only nodes for our numerical methods, we have to keep both supercells and nodes in our data structures.

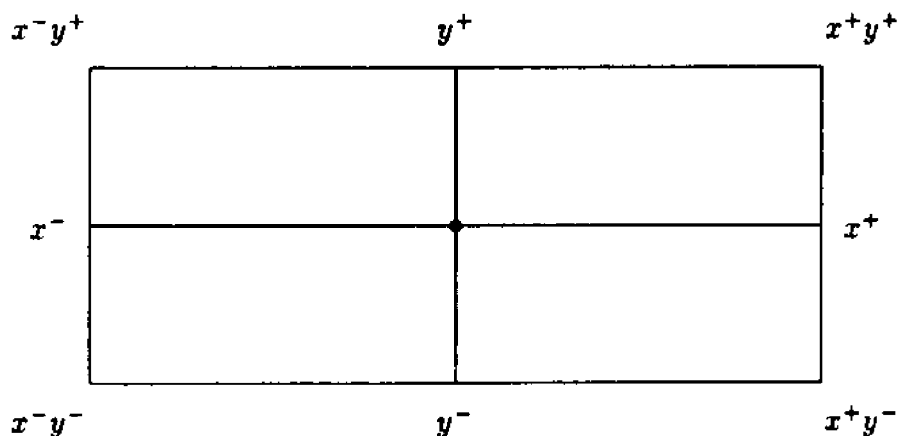


Figure 2.4: Location of neighbouring nodes relative to the central node.

We will frequently need to refer to the relative location of neighbouring grid components (nodes, cells, and cell interfaces). For the lack of better terminology we will say that nodes are located in the *directions* x^-, x^+, y^-, y^+, and in the *orientations* x^-y^-, x^+y^-, x^-y^+, x^+y^+ relative to the central node. A look at Figure 2.4 should make the meaning of the notation clear. The same notation will be used for the

relative positions of cells, for positions of cells relative to nodes, and vice versa.

Now we can precisely define the *local geometrical configuration*. The geometrical configuration around a terminal node is a correspondence between the orientations and the relative levels of the terminal cells adjacent to the node, where the relative level of a cell is an integer obtained by subtracting the central node grid level from the cell grid level. A geometrical configuration is identified by its *configuration number*.

2.2 Properties of consistent grids

Consistent grids have the following properties, most of which result directly from the definitions.

Property 1 *There is a one-to-one correspondence between the supercells and their associated x-nodes.*

Property 2 *The terminal cells surrounding a terminal node are on at most two different grid levels — the level of the node and one level below it. At least one of these cells is associated with the node and is, therefore, on the same grid level. Moreover, all four terminal cells surrounding a terminal x-node or a terminal s-node are on the same level. A terminal internal t-node is surrounded by three cells — two cells associated with it and one on the level below. Only a terminal c-node can have several different grid configurations around it.*

Proof. Follows from the definition of consistency. □

Theorem 1 *A node associated with a terminal cell is either terminal or has a co-located terminal node on the next grid level.*

Proof. Consider the terminal node p_t co-located with the given node p. According to Property 2 the terminal cells surrounding a terminal node can be on at most two different levels. Therefore, the highest level cell around p_t can be at most one level above the level of the given terminal cell. The same is true for the levels of nodes p and p_t which are associated with the aforementioned cells. □

Theorem 2 *Grid levels of two neighbouring terminal nodes differ at most by one.*

Proof. By definition two neighbouring terminal nodes are co-located with nodes of a terminal cell. Therefore, according to Theorem 1, the grid level of each node is either equal to the level of the cell, or is one level above it. □

Theorem 3 *An internal t-node of a consistent grid is always terminal.*

Proof. Given a t-node p on level l, suppose there is a node $\hat{p}$ on level $l+1$ co-located with node p. By the definition of an internal t-node, there is a terminal cell on level $l-1$ adjacent to node p. On the other hand, there is a cell of level $l+1$, associated with node $\hat{p}$, which is also adjacent to node p. Thus, the starting assumption would violate the grid consistency. □

Theorem 4 *A c-node located on the same side of a supercell with an internal t-node is always terminal.*

Proof. Analogous to the proof of Theorem 3. □

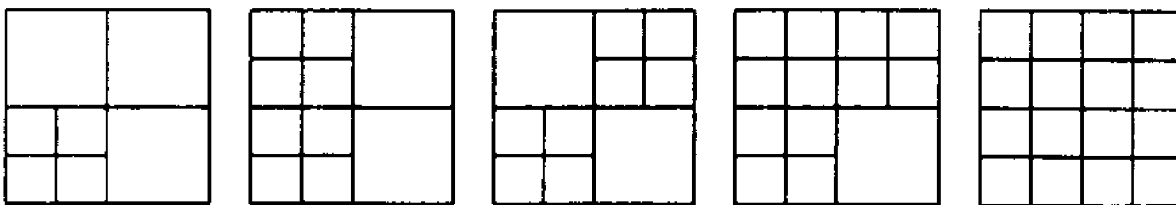


Figure 2.5: Non-trivial geometrical configurations around a grid node. The remaining configurations can be obtained by rotation.

Theorem 5 *The number of possible distinct geometrical configurations around an internal terminal r-node is fifteen (see Figure 2.5).*

Proof. It follows from Property 2 that terminal cells of only two different sizes can surround a terminal node and that the case where all the surrounding cells are on the level below the node level is not feasible. The theorem follows from the fact that there are four terminal cells around an internal terminal r-node. □

2.3 Grid data structures

The design of grid data structures must be a compromise between the amount of memory required and the support for the efficient execution of certain basic operations on the grids. An educated guess is that, given the current state of computer hardware and the type of problems to which recursive adaptive grids are likely to be applied, efficiency is more important than memory. The following is a list of basic grid operations which must be efficiently supported by grid data structures.

Grid refinement/unrefinement. Given a consistent recursive grid, a set of cells to be refined, and a set of supercells allowed to be unrefined, find a modified consistent

grid in which all the necessary cell refinements are performed and as many supercells as possible are unrefined.

Grid traversal. Traverse subsets of nodes in different order. Subsets of nodes can be defined by node types (such as t-nodes or r-nodes), grid levels, and node terminality. Among frequently used node orderings are the x-y, y-x orderings, and the nested dissection ordering.

Nearest neighbours search. Given a node, find its neighbouring nodes and/or neighbouring cells. Given a cell, find its neighbouring cells and/or associated nodes.

Point location relative to the grid. Given a point in the computational domain, find the node at the given grid level which is the closest to the point and the cell which contains the point.

Multigrid methods support. Given a node, find the co-located nodes on the adjacent grid levels. This operation is required in order to transfer grid vector values between different grid levels.

Two arrays are used to store the information about the nodes, the supercells, and their interconnections. The array `ctree` keeps the instances of the structure `SUPERCELL`, and the array `gnodes` contains the instances of the structure `GNODE`. Each instance of these structures is identified by its index in the corresponding array. The dynamic allocation module keeps track of the free slots in `ctree` and `gnodes`. When a new supercell or node is created, a slot for it is allocated; when a supercell or a node is deleted its slot is returned to the list of the free slots. The definitions of node and supercell structures are listed in Figure 2.6 with some comments.

The experience of writing this code shows that there is excessive information in

```
typedef struct supercell { /********** supercell structure *********/
    float ctr[2];         /* coordinates of the supercell center     */
    gn_index ctrnode;     /* central node of the supercell           */
    sc_index parcell;     /* index of the parent supercell           */
    int  load;            /* work load associated with cell sub-tree */
    int  maxlevel;        /* maximum level of descendants            */
    char pos;             /* position in the parent supercell        */
    char level;           /* supercell's level (root level is 0)     */
} SUPERCELL;

typedef struct gnode {     /********* grid node structure *********/
    gn_index nbr[4];       /* indexes of 4 neighbouring nodes     */
    gn_index fine;         /* co-located next level node          */
    union {
        gn_index coarse;   /* co-located previous level node      */
        sc_index scell;    /* associated supercell for x-node     */
    } u;
    unsigned char gtype;   /* node type and inconsistency bit     */
    unsigned char gdir;    /* encoded locations of adjacent cells */
} GNODE;
```

Figure 2.6: Node and supercell structures.

the present structures which can be kept instead as optional cell or node vectors.

2.4 Grid creation, refinement, and unrefinement

Grid adaptation is done in steps which may be interspersed with numerical computations. One step of grid adaptation is a solution to the following problem.

Definition 1 Grid adaptation problem.

Given a grid $\mathcal{G}$, a set of its cells $\mathcal{R}$ to be refined, and a set of its supercells $\mathcal{U}$ which are allowed to be deleted, find a minimal consistent grid in which all the cells of $\mathcal{R}$ are refined and all the supercells not belonging to $\mathcal{U}$ are preserved. A minimal grid is a grid

from which no supercells can be deleted without violating one of the above conditions. It is easy to see that because of the consistency requirement the minimal grid may contain some supercells of $\mathcal{U}$ marked for deletion, and some cells not belonging to $\mathcal{R}$ will be refined.

Theorem 6 *For the given grid $\mathcal{G}$ and the given sets $\mathcal{R}$ and $\mathcal{U}$ there exists a unique minimal grid.*

Proof. Suppose there are two different minimal grids $\mathcal{G}_1$ and $\mathcal{G}_2$. Without restricting generality, we can presume that the set $\mathcal{G}_1 \setminus \mathcal{G}_2$ is not empty and that the supercell $\tilde{s}$ is among its supercells on the highest grid level. Then, the parent cell of $\tilde{s}$ does not belong to the set $\mathcal{R}$, otherwise it would have been refined in $\mathcal{G}_2$. For the same reason, the supercell $\tilde{s}$ is not among the supercells of $\mathcal{G}$ which cannot be deleted. Finally, the deletion of $\tilde{s}$ will not cause inconsistency in $\mathcal{G}_1$, because it is absent in $\mathcal{G}_2$ and there are no supercells in $\mathcal{G}_1 \setminus \mathcal{G}_2$ which are above the grid level of $\tilde{s}$. Therefore, contrary to the initial conjecture, $\mathcal{G}_1$ is not a minimal grid. □

Let `create(s)` be the function which creates the supercell covering the cell s. Then, the recursive function `refine(s)` presented in Figure 2.7 refines the terminal cell s, preserving the grid consistency. Moreover, the grid remains consistent during the intermediate steps of the algorithm, because a supercell is created only after all of the neighbours on the grid level below its level have been created.

Let `destroy(s)` be the function which deletes the supercell covering the cell s. Then, the recursive function `unrefine(s)` presented in Figure 2.8 unrefines the cell s if possible without violating the grid consistency.

```
refine(s)
{
    For each neighbouring cell s_n, do {
        If s_n is one level below s, refine(s_n)
    }
    create(s)
}
```

Figure 2.7: One-cell consistent refinement algorithm.

```
unrefine(s)
{
    boolean nounrefine = FALSE
    If s is non-terminal, nounrefine = TRUE
    For each neighbouring cell s_n, do {
        If s_n is more than one level above s, nounrefine = TRUE
    }
    If nounrefine is FALSE, destroy(s)
}
```

Figure 2.8: One-supercell consistent unrefinement algorithm.

Because of consistency, the solution to the grid adaptation problem is global in the sense that refinement or unrefinement at a particular location of the grid may depend on the situation at a different location. A naive approach to the problem can lead to repeated unrefinements and refinements of the same cell. This would not only require unnecessary computing, but might also lead to the loss of numerical data associated with temporarily deleted nodes and cells. We propose an algorithm presented in Figure 2.9 which guarantees that this does not happen. The following notations are used: l is the current grid level, l_{max} is the maximal grid level, $\mathcal{R}_l$ and $\mathcal{U}_l$ are the subsets of the sets $\mathcal{R}$ and $\mathcal{U}$, respectively, consisting of supercells on the grid level l.

```
adapt()
{
    For l = 0,... l_max, do {
        For each supercell s ∈ R_l, refine(s)
    }
    For l = l_max,...0, do {
        For each supercell s ∈ U_l, unrefine(s)
    }
}
```

Figure 2.9: Optimal grid adaptation algorithm.

Theorem 7 *The grid adaptation algorithm is optimal in the sense that the resulting grid is minimal and that no cell refined at an intermediate stage of the algorithm is later unrefined.*

Proof. Let $\tilde{c}$ be the highest level cell which was refined and then unrefined. Because it was unrefined, $\tilde{c}$ cannot belong to $\mathcal{R}$, and, therefore, it was refined because there was a neighbouring cell $\tilde{c}^+$ on the grid level above that had to be refined. By the definition of $\tilde{c}$, $\tilde{c}^+$ cannot be among the cells later unrefined. Hence, $\tilde{c}$ too could not have been unrefined without violating consistency.

Suppose now that the grid $\mathcal{G}_a$ resulting from the application of the grid adaptation algorithm is not minimal. Let $\tilde{c}$ be a highest level cell which is refined in $\mathcal{G}_a$ but not in the minimal grid. Then, $\tilde{c}$ does not belong to the set $\mathcal{R}$. Therefore, $\tilde{c}$ either was created because there was a neighbouring cell $\tilde{c}^+$ on the grid level above its level that had to be refined, or $\tilde{c}$ belongs to the set $\mathcal{U}$ of the original grid. In the first case, by the definition of $\tilde{c}$, $\tilde{c}^+$ must be refined in the minimal grid, and, therefore, $\tilde{c}$ must also be refined in the minimal grid, which contradicts the definition of $\tilde{c}$. In the second case, nothing prevents $\tilde{c}$ from being deleted during the unrefinement phase of the

algorithm. □

2.5 Grid traversal and subsets

Parameters of a numerical method can be node based, cell based or cell interface based. We will use the word grid component to refer to any of these three types. Most often numerical methods work with node based parameters.

A grid is needed to support operations performed on data based at grid nodes, cells, or interfaces. Programs performing these operations must have the ability to traverse the grid. The order of grid traversal can be important. Grid components can be ordered by grid level, by geometrical order, by component type (such as t-nodes and r-nodes), or by any combination of the above orders.

Two sets of tools are provided for grid traversal — the basic grid traversal functions and index lists. Basic grid traversal functions (BGTF) traverse the grid using its internal tree structure. Index lists require some additional memory but are much more flexible than BGTF. They allow a set of grid components to be split into any number of subsets and maintain a specific order inside every subset. Several standard splittings are provided with the system. Users can create new splittings and orderings.

2.6 Search for neighbours

The grid data structure contains indexes of the neighbouring nodes on the same grid level for every direction. It also contains indexes of the co-located nodes on the adjacent levels. Therefore, all the neighbouring nodes can be found in $O(1)$

operations. The connections between nodes and supercells are made through x-nodes. Because of this, the connections between nodes can be used to find neighbouring cells as well.

Chapter 3

The finite element method on consistent recursive grids

For the spatial discretization on consistent recursive grids we will use the finite element method (Strang & Fix, 1973, Zienkiewicz, 1977). A particular version of the finite element method is defined by the choice of the basis functions and the weight functions.

3.1 Basis functions in space

A major problem in using recursive grids is the treatment of the interface between computational cells of different sizes. To solve this problem, we developed a version of the finite element method for consistent recursive grids. Discontinuous trial functions are used in this version in order to simplify matrix generation at the grid nodes adjacent to computational cells of different sizes.

We shall seek unknown functions in the approximate form

$$\phi = \sum a_i \phi_i \tag{3.1}$$

where ϕ_i are basis functions of space variables x and y, and a_i are node based parameters to be computed.

A basis function ϕ_i is a piecewise bilinear function which equals 1 at node i and equals 0 at the rest of the nodes. Let $\mathcal{J}_i$ be the set of elements (cells) which have node i as one of their vertices. Then ϕ_i is non-zero only over the elements belonging to $\mathcal{J}_i$, and is a bilinear function over each element of $\mathcal{J}_i$. The support of the function ϕ_i, defined as the sum of the elements belonging to $\mathcal{J}_i$, will be denoted by S_i. In our case, where the elements are rectangles in two-dimensional space, the number of elements in $\mathcal{J}_i$ is four for an internal r-node and two for a t-node. Assuming that the coordinate system starts at node i, its axes are parallel to the sides of the elements, and the element $j \in \mathcal{J}_i$ is located in the first quadrant, the restriction of a basis function ϕ_i over the element j can be written as

$$\phi_i = \frac{h_i^x - x}{h_i^x} \; \frac{h_i^y - y}{h_i^y} \tag{3.2}$$

where h_i^x and h_i^y are the x and y dimensions of the element j.

When elements of $\mathcal{J}_i$ have different sizes, or the node i is a t-node, the function ϕ_i is apparently discontinuous. The continuity of the expansion (3.1) is preserved by imposing the following linear constraints on the admissible set of parameters a_i

$$a_i = \frac{1}{2}(a_{i^+} + a_{i^-}) \tag{3.3}$$

for every internal t-node, where i^+ and i^- are the nodes on the ends of the cell side

in the middle of which the t-node i is located.

An alternative approach would be to eliminate the t-node based basis functions altogether and use continuous basis functions at the r-nodes. These functions would have larger support than basis functions ϕ_i, which would lead to a more complicated discretization pattern and, ultimately, would also complicate programming.

3.2 Weight functions

As weight functions we take normalized linear combinations of basis functions

$$\psi_i = \mu_i(\phi_i + \frac{1}{2}\sum_{k\in\mathcal{T}_i}\phi_k) \tag{3.4}$$

where $\mathcal{T}_i$ is the set (possibly empty) of t-nodes adjacent to the node i, and μ_i is chosen so that the integral of ψ_i over its support Q_i is unity. Note that ψ_i are continuous functions of x and y.

3.3 Upstream correction

It is well known that, when convective term is present in a differential equation, for reasons of stability an upstream discretization must be used. There are two difficulties in applying the upstream discretization as described in Zienkiewicz (1977) to our problem. First, it must be generalized to the case of two-dimensional non-uniform grids. Second, the choice of a smooth upstream weighting correction used by Zienkiewicz would lead to an unnecessary additional discretization error in the free

term. The upstream corrected weight functions have the form

$$\psi_i^u = \psi_i + \tau^x\theta_i^x + \tau^y\theta_i^y \tag{3.5}$$

Here, generalized functions θ_i^x and θ_i^y are the upstream corrections for velocity components in the x and y directions respectively. The coefficients τ^x, τ^y are between 0 and 1 (we use $\tau = 1$ when the corresponding velocity component is present in the equations and $\tau = 0$ when it is absent).

We will write down only the definition of θ_i^x for the case when the x velocity component at node i is positive; the definition of θ_i^y and the definitions for a negative velocity component are completely analogous. Consider the partition of Q_i by the line $x = x_i$ where x_i is the x coordinate of the node i. Let Q_i^{x+} and Q_i^{x-} be parts of Q_i located in the direction of increasing and decreasing x, respectively. Then

$$\theta_i^x = \theta_i^{x+} = \zeta_i^x\delta'(x - x_i)\psi_i \tag{3.6}$$

where δ is the delta function, and the coefficient ζ_i^x is chosen in such a way that

$$\int_{Q_i^{x+}} \theta_i^{x+} = -\int_{Q_i^{x+}} \psi_i \tag{3.7}$$

This choice eliminates the contribution of the downstream part of Q_i in the discretization coefficients when $\tau = 1$.

It can be shown that the upstream correction (3.6) does not affect the discretized coefficients of the second degree terms and the chemical production term.

3.4 Preassembling stiffness matrices

Consistent recursive grids in two-dimensional space can have only fifteen different geometrical configurations at an internal node. Therefore, a discrete operator approximating a differential operator with constant coefficients will be completely determined by two parameters: the configuration number and the grid level. As a result, the rows of a stiffness matrix can be preassembled for every configuration and for every level.

Chapter 4

The interpreter

The interpreter forms the foundation of the interactive programming environment. The components of the environment and their mutual dependencies are schematically shown in Figure 4.1. The parts of the interpreter are located inside the smaller dashed rectangle in the center.

At present only arithmetical expressions, assignments, and function calls are implemented. The rest, derived type definitions and control operators, are rarely needed. A function which has some logical complexity can always be implemented as a compiled C module. The interpreter is intended mainly for declaring data, examining them, and calling functions performing operations on the data.

We present here an informal description of the input language of the interpreter, leaving out the precise definition of the grammar.

4.1 Interpreter input language

For the most part, the interpreter mimics C lexical conventions and syntax (Kernighan & Ritchie, 1988). We will highlight the differences from C, rather than providing a

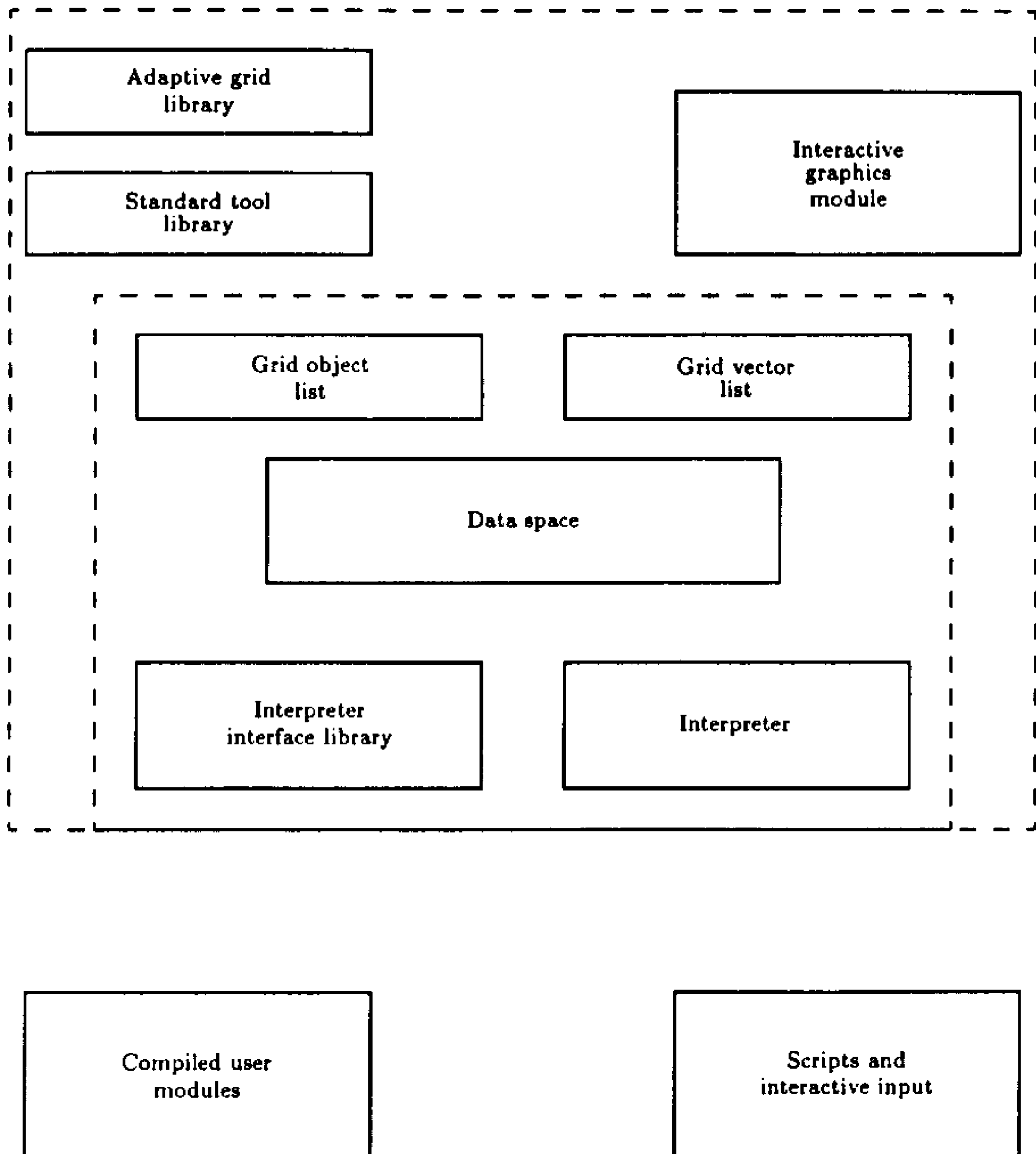


Figure 4.1: Functional chart of the environment.

detailed description of the language.

4.1.1 Lexical conventions

The rest of the line after the character # is considered a comment. Instead of a semicolon, operators are separated by a newline character.

4.1.2 Fundamental data types

The types which correspond directly to the C fundamental types are listed below followed by their C equivalents. Some of the types, namely `byte` and `index`, are also defined as derived C data types in the header file `finstd.h`.

`char`	`char`
`byte`	`unsigned char`
`short`	`short`
`ushort`	`unsigned short`
`int`	`int`
`index`	an unsigned integer type used for array indexing
`long`	`long`
`ulong`	`unsigned long`
`float`	`float`

`double` `double`

`void` `void`

4.1.3 Derived data types

Arrays of variables of any fundamental type can be declared. The number of elements in the array is stored with it and the boundaries are always checked by the interpreter during the execution. Interpreter arrays can be handled by compiled C modules through the data type `ARRAY` defined in the header file `finstd.h`.

The interpreter contains a derived type facility which would allow the user to define structures, once its front end has been implemented. At present, this facility is used internally to predefine several derived data types.

In the following list the predefined interpreter data types and the corresponding C data types are described. The corresponding C data types are defined in the header file `finstd.h`.

`iarray`

Index array. Index arrays must be accessed through the index list handling functions (Section 4.6). The corresponding C data type is `IARRAY`.

`ilist`

Index list identifier. The corresponding C data type is `ILIST`.

`rect`

A structure containing rectangle coordinates in the order `x1,x2,y1,y2`. The

corresponding C data type is `RECT`.

`point`

A structure containing point coordinates in the order `x,y`. The corresponding C data type is `POINT`.

4.1.4 Polymorphic arguments

Some of the interpreter system functions use an identifier as their argument. Such functions find out the type of the variable known under this name and act depending on the found type. The functions using this type cannot be called from a compiled user module. We will denote a polymorphic argument as a pseudo-type `ident`.

4.1.5 Conversions

Conversions between fundamental data types are done as in C, except that an overflow condition is detected and a warning printed.

4.1.6 Expressions

Arithmetic expressions are fully implemented, except for the postfix operators `.`, `->`, `++`, and `--`. Address operator `&` is implemented, but not operations on pointers.

4.1.7 Declarations

When a variable or an array is declared, a memory for it is allocated in the interpreter data space. A declaration can be followed by an `=` sign and an expression for the

optional initialization.

4.1.8 Statements

There are, at present, two types of statements: declarations and expressions. Unlike C, new variables and arrays can be declared at any place in the program. Expression statements (which include assignments) are immediately executed and the result is printed preceded by its type.

4.1.9 External variables

A number of external variables are defined in the interpreter. They can be used by compiled C modules as normal C external variables.

4.1.10 Function declarations

At present, because of MS-DOS limitations, compiled C modules are integrated with the interpreter in the following way. A special module with the user-defined function descriptions must be written. It contains the table of user-defined function pointers **usrftab** and the function **pr_usrfun** which parses calls to user-defined functions. For every user-defined function an entry in the table **usrftab** must exist. For those user-defined functions which will be called from the interpreter (for some functions only their pointers will be used), the function **pr_usrfun** must contain a parsing code. The data types, symbolic constants, and function prototypes used in the function description module are contained in the header files **intface.h** and **parser.h**. Finally,

the modules containing the user-defined functions and the module with the function descriptions must be linked with the interpreter.

The procedure of integrating user-defined functions with the interpreter must be changed to take advantage of the dynamic link facility of UNIX. The parsing of the user-defined functions must be done using function prototypes.

4.2 Script files

A sequence of interpreter commands saved in a disk file with the extension `.fn` is called a script file or a script function. Syntactically, a script file is equivalent to a function with no arguments and no return value whose name is the name of the script file without the extension. When a function is called, the interpreter tables of callable functions are searched, and then, if the function is not found, the current directory is searched for a script file with the specified name.

If a terminal error is encountered during the processing of one of the script's commands, the rest of the script file is ignored. Script files can be nested.

A name of a script file can be used as an argument when the interpreter is called. In this case, the script file is immediately executed, and, upon its completion, the interpreter is ready to accept interactive input, unless the run of the interpreter was ended by the `quit` system command encountered in the script file. This feature is used for initialization scripts and for batch runs.

4.3 Interpreter system functions

These functions provide information about the current state of the interpreter data space and perform other system functions.

```
void names(void)
```

Prints all names of variables, system- and user-defined external variables, and system- and user-defined functions, known to the interpreter.

```
void whatis(ident vname)
```

Prints what the interpreter knows about the name **vname**, which can belong to a variable, a system- or user-defined external variable, or a system- or user-defined function. The information printed includes the variable's type, attribute, and description, if any.

```
ARRAY *aprint(ident arr)
```

Prints the elements of the array **arr**. The printing format depends on the type of the array's elements.

```
void destroy(ident vname)
```

Destroys the interpreter variable **vname**. All the memory allocated for the variable is freed and its entry is removed from the interpreter table of declared variables.

4.4 Interface between compiled C functions and interpreter data space

The following functions allow compiled C modules to use data from the interpreter data space.

`void *vt_dclvar(char *vname, void *dptr, char *tname, char *info)`

Declares a variable `vname` of type `tname` to the interpreter. If `dptr != NULL`, the declared variable is initialized by data pointed to by `dptr`. A short description of the variable in the argument `info` is stored with it. Returns the address of the variable's data in the interpreter data space.

`void *vt_assign(char *vname, void *dptr, char *tname)`

Assigns value to the interpreter variable `vname` of the type `tname`. Returns the address of the variable's data in the interpreter data space. If the variable has not been found or has a different type, prints an error message and returns `NULL`.

`word *vt_attrib(char *vname, word attrib)`

Adds attribute `attrib` to the attributes (if any) of the interpreter variable `vname`. Returns the address of the variable's attribute in the interpreter data space. If the variable has not been found or its attribute can not be modified, prints an error message and returns `NULL`.

`ARRAY *vt_dclarr(char *vname, char *etype, size_t nelem, char *info)`

Declares to the interpreter an array `vname` consisting of `nelem` elements of the type `etype`. A short description of the array in the argument `info` is stored with it. Returns the address of the array's `ARRAY` structure in the interpreter data space, or `NULL` if there is not enough memory.

`IARRAY *vt_dcliarr(char *vname, index nelem, ILIST nlists, char *info)`

Declares to the interpreter an index array `vname` with `nelem` indexes and `nlists` list entries. A short description of the index array in the argument `info` is stored with it. Returns the address of the index array's `IARRAY` structure in the interpreter data space, or `NULL` if there is not enough memory.

`void *vt_demand(char *vname, char *vtype, char *msg)`

Looks for a variable `vname` of the type `vtype` in the interpreter data space. If the variable is not found or has different type, prints the argument `msg` (can be used to identify calling function, etc.) and requests that the user define it. Returns the address of the variable in interpreter data space, or `NULL` if not found.

`ARRAY *vt_arrdemand(char *vname, char *etype, char *msg)`

Looks for an array `vname` with elements of the type `etype` in the interpreter data space. If the array is not found or has elements of a different type, prints the argument `msg` (can be used to identify calling function, etc.) and requests that the user define it. Returns the address of the array in interpreter data

space, or `NULL` if not found.

4.5 Problem-specific objects

Problem-specific objects are grid objects, grid vectors, and space functions. In the current implementation, syntactically, part of their declaration is performed via function calls.

```
void go_dcl(ident vname, void (*init)(void), void (*refine)(void),
          void (*unref)(void))
```

Redeclares the variable `vname` to be a grid object and specifies the object's initialization (`init`), refinement (`refine`), and unrefinement (`unref`) functions (also called initialization, refinement, and unrefinement rules associated with the object). Executes the initialization function. The variable can be of any type and must be declared beforehand. The variable `vname` will be called the data of the identically named grid object created by the call to `go_dcl`. The grid objects are kept in a system list and rules associated with the objects are automatically executed when the corresponding grid event occurs. For information on the interface between grid management functions and initialization, refinement, and unrefinement rules, see Section 5.3.2.

```
void go_remove(ident vname)
```

Removes `vname` from the list of grid objects. The variable `vname` is not destroyed.

`void gv_dcl(ident name, word gvtype, ARRAY *isolev, void *palette)`

Redeclares variable **vname** to be a grid vector. The grid vector type is specified by the argument **gvtype** (value **NODEVEC** specifies a node vector, value **CELLVEC** specifies a cell vector). Grid vectors are kept in a system list and treated specially by the save/restore facility and by the interactive graphics module. Arguments **isolev** and **palette** specify the array of isolevels and the palette to be used by graphics functions.[1]

`void gv_sectattr(ident vname, char *colname, float ymin, float ymax)`

Marks the grid vector **vname** for cross-section plot and sets its cross-section attributes. The arguments **colname** and **ymin**, **ymax** specify the color and the numerical range of the function values to be used for cross-section plots.

`void gv_remove(ident vname)`

Removes **vname** from the list of grid vectors.

`void sf_dcl(ident fname)`

Declares function **fname** to be a space function. Space functions are functions written according to special rules which can be called in the scan mode from the interactive graphics module (Section 7.8.2).

---

[1]See the description of graphics functions in Section 7.6.

4.6 Functions for handling index lists

4.6.1 Creating and changing lists

`int il_create(IARRAY *iaptr, index nelem, ILIST nlists)`

Creates an index array `vname` with `nelem` indexes and `nlists` lists based on an existing `IARRAY` structure pointed to by `iaptr`. If not enough memory returns 0, otherwise returns 1.

`void il_clear(IARRAY *iaptr)`

Makes empty all the lists of the index array pointed to by `iaptr`.

`void il_destroy(IARRAY *iaptr)`

Destroys the index array based on the structure pointed to by `iaptr`.

`void il_copy(IARRAY *iaptr1, IARRAY *iaptr2)`

Copies the index array pointed to by `iarr1` into the index array pointed to by `iarr2`.

`void il_prlist(IARRAY *iaptr, ARRAY ilarr)`

Prints index lists specified in the array of index list identifiers `ilarr`. Only non-empty lists are printed.

`void il_push(IARRAY *iaptr, index ind, ILIST ils)`

Inserts the index `ind` at the beginning of the list `ils` of the index array pointed to by `iaptr`. If `ind` already belongs to a list, prints an error message.

`void il_put(IARRAY *iaptr, index ind, ILIST ils)`

Inserts the index `ind` at the end of the list `ils` of the index array pointed to by `iaptr`. If `ind` already belongs to a list, prints an error message.

`void il_insnext(IARRAY *iaptr, index ind, index igiv)`

Inserts the index `ind` after the index `igiv` in the list `ils` of the index array pointed to by `iaptr`. If `ind` already belongs to a list, prints an error message.

`void il_delete(IARRAY *iaptr, index ind)`

Deletes the index `ind` from its list in the index array pointed to by `iaptr`. If index does not belong to a list, prints an error message.

`void il_move(IARRAY *iaptr, index ind, ILIST ito)`

Deletes the index `ind` from its list and inserts it at the the end of the list `ito`.

4.6.2 Walking through index lists

Below is a set of functions for walking through lists. The current index array must be set before using these access functions.

`void il_setiarr(IARRAY *iaptr)`

Sets the current index array to be the array pointed to by `iaptr`.

`index il_first(ILIST ils)`

Returns the first index in the list `ils`, or 0 if the list is empty.

`index il_last(ILIST ils)`

Returns the last index in the list `ils`, or 0 if the list is empty.

`index il_next(index ind)`

Returns the next index after `ind` in the list to which `ind` belongs, or 0 if the end of the list has been reached.

4.7 Save/restore facility

At present, the save/restore facility uses a binary format for save files. While the binary format has minimal storage requirements, it prevents the use of this facility for the transfer of data between different computers. The save/restore facility must be improved to provide a portable way for storing data.

`void sopen(ident fname)`

Opens the save file named `<fname>.dat`. If a file with this name exists, asks for confirmation to overwrite it.

`void sclose(void)`

Closes the currently opened save file.

`void sgrid(void)`

Saves the grid on the currently opened save file.

`void save(ident vname)`

Saves the variable's `vname` data together with its name and type in the currently

opened save file.

`void restore(ident fname)`

Restores the values of the variables saved in the save file `<fname>.dat`. In order to be restored, a variable with the identical name and type must be declared beforehand. For the arrays the number of elements must be sufficient to accommodate the saved data.

Chapter 5

The adaptive grid library

The adaptive grid library user-accessible macros, external variables, and functions are defined in the include file `grid.h`.

5.1 Specifying directions, orientations, and positions

Directions, orientations, and positions (Section 2.1) are specified by symbolic integer constants. When data pertinent to relative locations of grid components are kept in an array, these constants are used as array indexes. Names of constants are composed of substrings which have the following meaning: `XM`, `YM` stand for x^-, y^-; `XP`, `YP` stand for x^+, y^+; `N` stands for *number of*; `NO` stands for *none*; and `DIM`, `DIR`, and `ORI` stand for *dimensions, directions, and orientations.*

`NDIM`

Number of dimensions of computational domain space.

`XDIM, YDIM`

Non-negative integers, less than `NDIM`, specifying dimensions of space (axis of

coordinates).

NDIR, NORI

Number of directions and orientations in the computational domain space.

XMDIR, XPDIR, YMDIR, YPDIR

Non-negative integers, less than NDIR, specifying positive and negative directions of the axis of coordinates.

XMYMORI, XPYMORI, XMYPORI, XPYPORI

Non-negative integers, less than NORI, specifying orientations (of corners in a rectangle or cells in a supercell).

NODIR, NOORI

None of the directions, none of the orientations.

NDIR1

Number of directions in one-dimensional space.

MDIR1, PDIR1

Non-negative integers, less than NDIR1, specifying plus direction and minus direction in one-dimensional space.

NCONFIG

Number of possible geometrical configurations at a node. Valid node configuration numbers (Section 2.1) are positive integers less than NCONFIG.

5.2 Grid-specific data types

The functions of the adaptive grid library use the following defined data types.

`gn_index`

An unsigned integer type used for grid node indexes.

`sc_index`

An unsigned integer type used for supercell indexes.

`SUBCELL`

A structure used as a cell descriptor. It has two members:

`sc_index scind`

The index of the supercell containing the cell. Zero index value signifies a null descriptor.

`int pos`

The position of the cell in the supercell.

5.3 External variables

External variables are used to keep information about the grid's current status and to transfer information between grid management functions and compiled user modules.

5.3.1 Grid current status information

All the following external variables are read-only accessible from the interpreter. They are modified by grid management functions and should not be changed in compiled user modules.

`int maxlevel`

Maximum grid level defined during compile time.

`int finlevel`

The finest grid level of the current grid.

`iarray sciarr`

An array of index lists of supercells according to their grid level. Additional internal index lists in `sciarr` are used by the function `gr_adapt` to keep track of the supercells slated for deletion.

`ilist levelsc[maxlevel]`

The element `levelsc[lev]` specifies the list of level `lev` supercells in the index array `sciarr`.

`index maxnodes, maxscells`

The maximum numbers of grid nodes and supercells.

`index ngnodes, nscells`

The current numbers of grid nodes and supercells.

`rect domain`

The computational domain.

5.3.2 Refine/unrefine function interface

The set of external variables described here is used for communication between refine/unrefine functions in the adaptive grid library and refine/unrefine rules associated with grid objects. Every time a cell is refined or unrefined the information specifying grid components, which have been newly created or are about to be deleted, is placed into these variables. It is for use by refine/unrefine rules associated with grid objects. These variables should not be changed by refine/unrefine rules. Compiled user modules other than refine/unrefine rules can use them freely.

Because a refine/unrefine rule can be used for more than one grid object, the pointers to the data and the type record of the currently processed grid object are made available.

`SUBCELL ge_parcell`

The cell to be refined or unrefined.

`gn_index ge_pnode[NORI]`

The nodes of the cell to be refined or unrefined.

`sc_index ge_scind`

The newly created supercell or the supercell to be deleted.

`int ge_level`

The grid level of **ge_scind**.

gn_index ge_xnode

The x-node associated with **ge_scind**.

gn_index ge_tnode[NDIR]

The t-nodes and s-nodes associated with **ge_scind**.

gn_index ge_cnode[NORI]

The c-nodes associated with **ge_scind**.

int ge_tnew[NDIR]

The newly created or about to be deleted t-nodes associated with **ge_scind** are marked by 1.

int ge_cnew[NORI]

The newly created or about to be deleted c-nodes associated with **ge_scind** are marked by 1.

void *ge_objtype, *ge_objdata

Pointers to the type record and the data of the currently processed grid object.

5.3.3 Refine/unrefine decision interface

These external variables are used for communication between refine/unrefine decision functions and the function **gr_adapt**. Possible values are YES(1) or NO(0).

`int ge_refine[NORI]`

The refinement decisions for the cells of the current supercell.

`int ge_unref`

The unrefinement decision for the current supercell.

5.4 Grid management functions

The following functions are used for grid creation, refinement, and unrefinement. The functions can be called both from the interpreter and from a compiled C module.

```
void gr_dclgrid(gn_index ndslots, sc_index scslots,
          void (*usrinit)(void),void (*usrend)(void))
```

Declares a grid space with the maximum number of node slots `ndslots`, and the maximum number of supercell slots `scslots`. A grid must be "assigned" to the grid space by a grid creation function or by restoring a saved grid. The function `usrinit` will be called after each such "assignment". It may be used, for example, to preassemble matrix coefficients for the given root grid size. The function `usrend` is to be called when the grid space is destroyed and can be used to free space allocated by `usrinit`.

`void gr_destroy(void)`

Frees the grid space. Calls the function specified as an argument `usrend` to `gr_dclgrid`.

`void gr_rgcreate(RECT *dptr, char *xname, char *yname, int nx, int ny)`

Creates an **nx** by **ny** rectangular array of root supercells covering the rectangular domain specified by **dptr**. Strings **xname**, **yname** are used to name the corresponding axes of coordinates. This procedure "assigns" a value to the grid space, which must be allocated beforehand by calling **gr_dclgrid**.

`int gr_refine(sc_index scind, int pos)`

Refines a cell located at position **pos** of supercell **scind** using the one cell consistent refinement algorithm (Section 2.4). Supercells are created from lower level up so that consistency is never violated during the intermediate steps. Nodes associated with created supercells, which did not exist before, are created as well. After the creation of each new supercell, **gr_refine** places the information about grid components that have been created into external variables described in 5.3.2 and executes the refinement rules for all the grid objects. The function returns the number of created supercells. If **gr_refine** fails, it returns 0. Note that in the latter case some supercells and associated nodes may have been created.

`int gr_unref(sc_index scind)`

Deletes supercell **scind** using a one supercell consistent unrefinement algorithm (Section 2.4). Removes associated patch of fine grid which is not associated with other supercells. If the supercell cannot be unrefined without violating grid consistency the function quietly refuses to so. The user is advised to start un-

refinement from the finest level up, so that supercells would not be unnecessary prevented from unrefinement by their finer neighbours. Just before the deletion of each supercell, **gr_unref** places the information about grid components to be deleted into external variables described in 5.3.2 and executes the unrefinement rules for all the grid objects. If the supercell is deleted the function returns 1, otherwise it returns 0.

```
void gr_adapt(void (*rucfun)(sc_index))
```

Goes through the grid cells refining and unrefining them according to the user-supplied refinement criteria function **rucfun**. Function **rucfun** accepts a supercell index as its argument and uses external variables **ge_unref** and **ge_refine** to pass refine/unrefine decisions for the supercell to **gr_adapt**. Grid consistency is maintained automatically, independently of decisions made by **rucfun**. The function uses the consistent global refinement/unrefinement algorithm described in Chapter 2.4.

5.5 Grid traversal

The grid traversal functions use both the grid tree structure and the internal lists of supercells by their grid levels maintained by the grid management functions. Grid traversal using node-to-node connections is also possible and can be somewhat faster. It has not been yet implemented.

To traverse a grid, one of the traversal setting functions must be called first. The subsequent calls to a corresponding traversal function will return the nodes or

supercells of the grid subset (which can include the whole grid) defined by the last called traversal setting function. Each grid component in the subset is visited only once.

Besides using functions described in this section, grid traversal can be done using index lists (see Section 4.6).

5.5.1 Grid traversal using the tree structure

The following functions use the supercell tree for grid traversal. The supercells are visited in depth-first order. The descendants of a supercell are visited in $x - y$ order. The root supercells are also visited in $x - y$ order.

Note that the traversal functions and the traversal setting functions described in this section should be used together.

`void ga_setall(void)`

Initializes grid traversal using the tree structure. The entire grid will be traversed by the subsequent calls to a grid traversal function.

`void ga_setrest(RECT *r,int maxlev,sc_index rscind)`

Initializes grid subset traversal. The grid subset is defined by a set of restrictions. The supercells or nodes that will be returned by subsequent calls to a traversal function are those which at the same time intersect with the rectangle pointed to by `r`, are on grid levels lower than or equal to `maxlev`, and are descendants of supercell `rscind`. If `r == NULL` the rectangle includes the entire the computational domain. If `rscind == 0` the descendants of all the root

supercells will be included in the traversal.

`sc_index ga_firstup(void)`

Returns the index of the next supercell of the subset defined by the last call to a corresponding grid traversal setting function. At the end returns 0.

`gn_index ga_nodefirstup(void)`

Returns the index of the next node of the subset defined by the last call to a corresponding grid traversal setting function. At the end returns 0.

`gn_index ga_termnd(void)`

Returns the index of the next terminal node of the subset defined by the last call to a corresponding grid traversal setting function. At the end returns 0.

5.5.2 Grid traversal using internal supercell lists

The following functions use the internal lists of supercells according to their grid levels, which grid management functions maintain in the index array `sciarr`. The order of the supercells on the same grid level is undefined.

Note that the traversal functions and traversal setting functions described in this section should be used together.

`void ga_setlevelup(void)`

Initializes walk through supercells in level-up order. All the supercells of the grid will be returned by the subsequent calls to a grid traversal function.

`sc_index ga_levelup(void)`

Returns the next supercell in level-up order. At the end returns 0.

5.6 Search for neighbours

5.6.1 In-line functions to find connected nodes and supercells

Following in-line functions (C macros) directly use information contained in the grid node and supercell structures. If the required node does not exist, index 0 is returned.

`NBRN_IND(gnind,dir)`

Finds the index of the neighbouring node located in direction `dir` from the given node `gnind`.

`XNODE_IND(scind)`

Finds the associated x-node index given a supercell index `scind`.

`SCELL_IND(xnind)`

Finds an associated supercell index given an x-node index `xnind`.

`COARSE_IND(cnind)`

Finds the grid node which is co-located with a c-node `cnind` and is one grid level below it.

`FINE_IND(gnind)`

Finds the grid node which is co-located with a node **gnind** and is one grid level above it.

5.6.2 Finding neighbouring nodes

`unsigned gr_ndconf(gn_index gnind)`

Finds the configuration number (Section 2.1) of the node **gnind**.

`void gr_cellnd(gn_index gnind, int ori, gn_index lnode[NORI])`

Finds the nodes of the cell located at orientation `ori` from the node **gnind** and associated with it. The nodes' indexes are placed into the array **lnode** according to their orientation in the cell. If the cell does not exist, the existing neighbouring nodes will be placed in **lnode** with appropriate indexes, and the rest of the array elements will be set to 0.

`void gr_celltnd(gn_index gnind, int ori, gn_index lnode[NORI])`

Finds the terminal nodes co-located with the nodes of the cell located at orientation `ori` from the node **gnind** and associated with it. The nodes' indexes are placed into the array **lnode** according to their orientation in the cell. If the cell does not exist, the terminal nodes co-located with the existing neighbouring nodes will be placed in **lnode** with appropriate indexes, and the rest of the array elements will be set to 0.

`void gr_scellnd(void)`

Finds the nodes of a supercell and its parental subcell if it exists. The external variables from Section 5.3.2 are used for input (`ge_scind`) and output (`ge_xnode`, `ge_tnode`, `ge_cnode`, and `ge_pnode`).

`void gr_ndcell(gn_index gnind,SUBCELL ndcell[NORI])`

Finds the cells surrounding the given node `gnind`. The cells' descriptors are placed in the array `ndcell`. The cells are either on the same grid level or, if there is no cell on the same level, on the level below.

5.6.3 Location functions

Using the grid tree structure, the location of a point relative to the grid can be found very efficiently. In fact, for a grid covering the computational domain with uniform resolution, the number of operations which are needed to find the relative location is of the order of the logarithm of the number of nodes.

`POINT gr_ptofnd(gn_index gnind)`

Returns the coordinates of a node `gnind`.

`SUBCELL gr_clofpt(POINT pt, int reqlev, int *cellev)`

Finds the cell at the highest existing grid level less than or equal to `reqlev` which contains the given point `pt`. Returns the cell descriptor of the found cell. The actual cell level is written into `*cellev`. If the point is outside of the computational domain, returns null cell descriptor.

`gn_index gr_ndofpt(POINT pt, int reqlev, int *nodelev)`

Finds the node at the highest existing grid level less than or equal to `reqlev` that is the closest to the given point pt. Returns the node index. The actual node level is written into `*nodelev`. If the point is outside of the computational domain, returns 0.

`RECT gr_lrect(sc_index scind,int pos)`

Returns proper rectangle containing a cell specified by the supercell index `scind` and the cell position `pos`.

5.7 Saving/restoring the grid

Because of the grid complexity, special functions are used to save and restore the grid. Grid save/restore functions use standard interpreter functions to save/restore the arrays and variables which constitute the grid data.

`void gr_save(FILE *fp)`

Saves the grid data in a file. Only the part of the grid arrays up to the largest index currently used is saved.

`int gr_restore(FILE *fp)`

Restores the grid data from a file. Grid space must be declared beforehand. If the grid space is too small to hold the saved grid, the function fails. Returns 0 if successful, returns error code otherwise.

Chapter 6

The standard tool library

Standard tool library consists of functions which are of general applicability, but do not belong to the system nucleus. Only some of the library tools may be needed for a particular application. There may be several standard tool libraries, implementing, for example, different classes of discretization methods. The present library is oriented towards the bilinear finite element approximation.

6.1 Grid node coordinates

In the grid data, coordinates are kept only for the centers of the supercells. If node coordinates are needed, they have to be computed by finding an associated supercell and its size. This requires a certain amount of work, which can be significant if the coordinates are needed often. One example is the computation of the x-momentum of a node vector in the numerical model of flame propagation in shear flow (8.2.2). The solution is to declare a grid object having a floating point node vector as its data, and have the coordinates of the nodes computed by the initialize function and updated by the refine function associated with the grid object. And, of course, if we want to convert the grid into a non-uniform non-rectangular grid, we can take a more general

approach and have the node coordinates computed by a user program.

This is a good example of how a skeleton grid structure, which stores minimal information, can be augmented by using a grid object mechanism. Different problems place different demands on the grid data structure. Having a set of optional grid objects to meet these demands allows the amount of required memory to be reduced.

Following are the initialize and refine functions needed to define a grid object for a node coordinate vector. There is no need for the unrefine function.

`void initXcoord(void)`

Initializes the function which computes the coordinates for all the existing nodes of the grid.

`void refXcoord(void)`

Refines the function which computes the coordinates of the nodes of the newly created supercell.

6.2 Node and supercell lists

Another example of the use of the grid object mechanism is maintaining lists of nodes and supercells according to their terminality, level, and node type. In this module we use a slightly different programming style than in the previous section. The necessary interpreter variables are declared to the interpreter from within a compiled C module. Other compiled C modules can access them through external variables declared in the module as well as by using interpreter interface functions. The refine, unrefine, and

initialize functions for the grid objects of the module are defined inside the module and are not visible to the user.

To create lists of nodes and/or supercells, one or both of the following functions must be called from the interpreter or from a compiled C module.

`void sd_defgn(void)`

Declares a grid object with an index array for grid node indexes as its data, and declares arrays of index list numbers. The `i`-th element of an array of index list numbers is the number of the list of nodes of corresponding type on the `i`-th level of the grid. The following is a list of declared interpreter variables and pointers to them declared as external C variables.

```
vt_dcliarr("gnodesia",maxnodes,3*maxlevel)
external IARRAY *gnodesia;
```

The index array for grid node indexes.

```
ilist termnd[maxlevel]
external ILIST *termnd;
```

The array of numbers of terminal r-node lists.

```
ilist nontnd[maxlevel]
external ILIST *nontnd;
```

The array of numbers of non-terminal node lists.

```
ilist tnodes[maxlevel]
external ILIST *tnodes;
```

The array of numbers of internal t-node lists.

```
void sd_defsc(void)
```

Declares a grid object with an index array for supercell indexes as its data, and declares arrays of index list numbers. The i-th element of an array of index list numbers is the number of the list of supercells of corresponding type on the i-th level of the grid. The following is a list of declared interpreter variables and pointers to them declared as external C variables.

```
vt_dcliarr("ctreeia",maxscells,3*maxlevel)
external IARRAY *ctreeia;
```

The index array for supercell indexes.

```
ilist termsc[maxlevel]
external ILIST *termsc;
```

The array of numbers of terminal supercell lists.

```
ilist nontsc[maxlevel]
external ILIST *nontsc;
```

The array of numbers of non-terminal supercell lists.

```
ilist semtsc[maxlevel]
external ILIST *semtsc;
```

The array of numbers of semi-terminal supercell lists.

6.3 Standard refinement and unrefinement rules

These refinement and unrefinement rules can be used directly in a grid object declaration or as a part of a user refinement or unrefinement rule. The grid object data must be a floating point node vector.

```
void reflin(void)
```

Linearly interpolates a floating point node vector from the parent cell to the new supercell.

```
void unreflin(void)
```

Linearly injects a floating point node vector from a supercell about to be removed to the parent cell. Because remaining s-nodes will become t-nodes, assigns new values to them by linearly interpolating values from the neighbouring c-nodes.

6.4 Bilinear approximation utilities

In the bilinear finite element method, an approximation to the solution is sought in the form of a function which is bilinear over each grid cell. Such a piecewise bilinear function is specified by a floating point node vector of coefficients of the expansion (as in equations 8.12, 8.13). In order for this expansion to be continuous, the values of the vector at the internal t-nodes must be linearly interpolated from the values at the adjacent c-nodes. We will say that the function expressed by such an expansion is *based* on the node vector used in the expansion. Clearly, a node vector and a function

which is based on it have identical values on the grid nodes.

6.4.1 General purpose algebraic functions

`float vabs(ARRAY vec)`

Computes and returns the absolute norm of the function based on the node vector vec on terminal nodes. The norm is computed over the computational domain.

`float vl2(ARRAY vec)`

Computes and returns the L_2 norm of the function based on the node vector vec on terminal nodes. The norm is computed over the computational domain.

`void vbounds(ARRAY *vec, RECT *r, float *vminptr, float *vmaxptr)`

Computes minimum and maximum of the function based on the node vector vec on terminal nodes. The minimum and maximum are computed over the rectangle pointed to by r. If `r = NULL`, the rectangle is assumed to include the computational domain. The minimum is returned via the argument `vminptr`; the maximum is returned via the argument `vmaxptr`.

`float vint(ARRAY vec)`

Computes and returns the integral of the function based on the node vector vec on terminal nodes. The integral is computed over the computational domain.

`float sd_gvbilinxy(float *varr, POINT pt)`

Computes the value of the function based on the node vector with elements

pointed to by `varr` at the given point `pt`.

6.4.2 Grid vector utilities

These utilities help to compute or modify node vectors so that functions based on them are continuous.

```
void st_ndvset(char *ndvname, float (*fptr)(float,float))
```

Assigns the values of the function of two space variables pointed to by `fptr` to the node vector `ndvname`. Values on the internal t-nodes are linearly interpolated from the neighbouring c-nodes.

```
void sd_lintnd(float gv[])
```

Assigns to the internal t-nodes of the node vector `gv` the values obtained by linearly interpolating the values from the neighbouring c-nodes.

6.5 Preassembling stiffness matrices

The implementation of the finite element method on recursive grids described in Chapter 3 can be greatly simplified and made more efficient using the set of tools for stiffness matrix definition and preassembly. Because the grid can have only a limited number of local configurations (Section 2.1), the preassembled rows take relatively little memory. In addition, the preassembly and the allocation of memory can be done selectively for the operators actually used in the problem.

6.5.1 External variables

The following external variables are used for communication between bilinear finite element functions and a compiled user module calling these functions. These variables are not interpreter accessible.

`gn_index fe_ctrnd`

The central node of the discretization.

`int fe_level`

The grid level of the central node.

`int fe_maxlevel`

The maximum grid level to consider.

`unsigned fe_conf`

The configuration number of the central node.

`gn_index fe_packnd[13]`

An array of grid node indexes of the discretization star packed in a defined order.

`int fe_ndconfig[NCONFIG]`

The number of grid nodes in the discretization star. The configuration number is used as the array's index.

`float fe_elmrow[NORI]`

Used by the functions which compute element stiffness matrix coefficients to

place the results of their computations.

The following arrays of pointers are to be used to access preassembled packed rows of stiffness matrices for common differential operators. The actual memory for the coefficients of a particular operator is allocated by the preassembly function **fe_assemble**. Each array declaration is followed by the symbol of the corresponding operator.

`float *fe_dx2[MAXLEVEL][NCONFIG]` — $\frac{\partial^2}{\partial x^2}$

`float *fe_dy2[MAXLEVEL][NCONFIG]` — $\frac{\partial^2}{\partial y^2}$

`float *fe_d2[MAXLEVEL][NCONFIG]` — $\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$

`float *fe_dx1p[MAXLEVEL][NCONFIG]` — $\frac{\partial}{\partial x}$ with an upstream correction for the positive x-component of velocity

`float *fe_dx1n[MAXLEVEL][NCONFIG]` — $\frac{\partial}{\partial x}$ with an upstream correction for the negative x-component of velocity

`float *fe_dy1p[MAXLEVEL][NCONFIG]` — $\frac{\partial}{\partial y}$ with an upstream correction for the positive y-component of velocity

`float *fe_dy1n[MAXLEVEL][NCONFIG]` — $\frac{\partial}{\partial y}$ with an upstream correction for the negative y-component of velocity

`float *fe_bilin[MAXLEVEL][NCONFIG]`— the identity operator (free term)

6.5.2 Functions

`void fe_init(void)`

Initializes preassembling module. The function must be called before any calls to **fe_assemble** are made.

```
void fe_assemble(void (*elmrow)(int level,int ori),
        float *acoeff[MAXLEVEL][NCONFIG], int vdir, float tau)
```

Allocates memory for the coefficients if it has not been allocated before. Preassembles the coefficients of the stiffness matrix for the linear operator specified by the argument **elmrow**. The array of pointers **acoeff** is initialized to access the coefficients. The element **acoeff[lev][nconf]** points to the array of packed coefficients of the row of the stiffness matrix computed for a grid node on the level **lev** and with the configuration number **nconf**. The number of coefficients in the row is stored in **fe_ndconfig[nconf]**. The order of the coefficients in the array of packed row coefficients corresponds to the order of grid nodes in **fe_packnd**.

The linear operator is specified by the pointer **elmrow**, which points to a function that computes the stiffness matrix for an element. A set of such functions for common linear differential operators is included in the module.

If the upstream correction is required, the argument `vdir` must be the direction code of the velocity component corresponding to the differential operator, and the argument `tau` is equal to the desired upstream coefficient. If `vdir == NODIR`, the upstream correction is not performed and `tau` is ignored.

`void fe_free(float *acoeff[MAXLEVEL][NCONFIG])`

Frees memory allocated for preassembled coefficients.

`void fe_adjtnd(void)`

Finds the discretization star for a terminal grid node. The central node of the discretization and its grid level are specified in `fe_ctrnd` and `fe_level`. The configuration number of `fe_ctrnd` is returned in `fe_conf`. The computed indexes of the nodes of the discretization star are placed in the array `fe_packnd` in a defined order that corresponds to the order of preassembled coefficients used by `fe_assemble`. The number of nodes in the array is stored in `fe_ndconfig[nconf]`. For a boundary node the existing part of the discretization star is found.

`void fe_mirradjnd(int mirrdir)`

Mirrors the part of the discretization star which is located on one side of the axis passing through the central node of the discretization. The axis is perpendicular to the direction `mirrdir` and the part of the discretization star to be mirrored is located in the direction opposite to `mirrdir` from the axis. The discretization star must first be computed by the function `fe_adjtnd`. The function `fe_mirradjnd` is to be used for mirror boundary conditions.

The following is a list of functions which compute the coefficients of the element stiffness matrices for common linear differential operators. The input parameters are the grid level `lev` and the orientation of the element relative to the central node `ori`.

The functions place the computed coefficients into the array **fe_elmrow**. These functions are used as an argument to the function **fe_assemble**. Each function declaration is followed by the symbol of the corresponding operator.

`void fe_dx2mrow(int lev, int ori)` — $\frac{\partial^2}{\partial x^2}$

`void fe_dy2mrow(int lev, int ori)` — $\frac{\partial^2}{\partial y^2}$

`void fe_d2mrow(int lev, int ori)` — $\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$

`void fe_dx1mrow(int lev, int ori)` — $\frac{\partial}{\partial x}$

`void fe_dy1mrow(int lev, int ori)` — $\frac{\partial}{\partial y}$

`void fe_bilinmrow(int lev, int ori)` — the identity operator (free term)

Chapter 7

Visualization tools

Graphics is the most implementation-dependent part of the package. At present, the system is implemented on an IBM PC-AT under MS-DOS using the Miscrosoft C6.00 graphics library, on an RS-6000 under AIX using the X Window GL library, and partially on a CRAY under CTSS using NCARGKS.

When a graphics function is called a new graphics window is created. In the X-window implementation on RS-6000 the window placement is controlled by the window manager. After exiting, the graphics window is closed and control returns to the interpreter. In the MS-DOS implementation on IBM PC-AT, the interpreter input/output screen is erased and the VGA high-resolution graphics mode is selected. After exiting, the screen is returned to the default text mode.

7.1 Color spaces

An individual color is defined by three floating point numbers (red, green, blue) and is referred to by its integer index in a color space. We define and use two color spaces — the named color space and the indexed color space. Depending on the implementation the spaces may overlap (this is the case in the IBM PC-AT implementation). The

indexed color space is used to display ranges of numerical values — as in grid vector display. The named color space is used in all the other cases — menus, labels, graphs.

The mapping of the color space indexes into the space of floating point triplets which define colors is called the color space palette. Several predefined palettes are provided for the indexed color space and new palettes can be defined interactively.[1] A palette is identified by a pointer to the palette's data array. Note that if the color spaces overlap, the palette change for the indexed color space may affect the named color space.

The following symbolic constants are defined in the header file `graph.h`.

`NINDCOLORS`

The number of colors in the indexed color space.

`NNAMEDCOLORS`

The number of colors in the named color space. The following symbolic constants specify colors:[2]

`BLACK`	`BLUE`	`GREEN`	`CYAN`
`RED`	`MAGENTA`	`BROWN`	`WHITE`
`GRAY`	`LIGHTBLUE`	`LIGHTGREEN`	`LIGHTCYAN`
`LIGHTRED`	`LIGHTMAGENTA`	`LIGHTYELLOW`	`BRIGHTWHITE`

---

[1] The mechanism for the interactive definition of new palettes is not yet fully implemented.

[2] Color names used in the interactive graphics module are abbreviated versions of the color names used for the symbolic constants.

7.2 Coordinate systems

Three coordinate systems are used by graphics functions.

Screen coordinates. The screen coordinates of a pixel are expressed relative to the lower-left corner of the screen. The coordinate count starts from zero.

World coordinates. The mapping of the current world window rectangle onto the graphics window defines the world coordinates.

Text coordinates. The coordinates of a text character are specified by row and column numbers, which start from 1 at the upper-left corner of a pop-up text window.

7.3 Grid function display

We use two alternative graphical representations for functions of two variables — isointerval colorfills and isolevel contours. In both cases an absolute isolevel array, which is a monotonically increasing array of floating point numbers, is used to assign colors to the intervals of the function range or to the chosen function values.

Under the isointerval colorfill representation, a color is assigned to every interval between the two neighbouring elements of the absolute isolevel array, referred to hereafter as an *isointerval*. Then for every isointerval, the area of the function domain over which the function value falls into the isointerval is painted by the corresponding color.

In the isolevel contour representation a color is assigned to every element of the absolute isolevel array. The contours formed by the points in the function domain

for which the function value is equal to an element of the absolute isolevel array are each drawn using the color assigned to the corresponding element.

An absolute isolevel array is computed using a relative isolevel array specified by the user. A relative isolevel array must belong to one of two array types — positive or indefinite. Positive array elements increase monotonically from 0 to 1; indefinite array elements increase monotonically from −1 to 1. The elements of an absolute isolevel array are computed as

$$a[i] = v_{min} + (v_{max} - v_{min}) * r[i], \quad \text{for } i = 0, \dots N \tag{7.1}$$

for a positive relative isolevel array and

$$a[i] = \begin{cases} -\min(v_{min}, 0) * r[i], & r[i] < 0 \\ \max(v_{max}, 0) * r[i], & r[i] \geq 0 \end{cases} \quad \text{for } i = 0, \dots N \tag{7.2}$$

for an indefinite relative isolevel array, where a is an absolute isolevel array, r is a relative isolevel array (both arrays start from 0 and have N elements), and v_{min} and v_{max} are the minimum and the maximum of the function to be displayed.

The following color index is used to represent the i-th isointerval $[a[i], a[i+1])$ (isointerval colorfill representation) or the i-th isolevel $a[i]$ (isolevel contour representation)

$$color = i \bmod \texttt{NINDCOLOP3} \tag{7.3}$$

where `NINDCOLORS` is the number of colors in the indexed color space.

A grid vector representation can be reflected along any side of the computational domain to obtain presentation quality pictures. This feature is needed, for example, when the phenomenon of interest is axisymmetric and only half of the physical domain is taken as the computational domain. We call this device *mirrors*.

7.4 Cross-sections

A cross-section is the restriction of a function of two variables over the straight line interval connecting two points (the section interval), with the distance from the starting point as the independent variable.

Cross-sections are displayed in the section window, which may be identical to the graphics window, depending on the implementation. The section interval is mapped into the horizontal axis of the section window. The interval of the function range (y scale), defined by the user, is mapped into the vertical axis of the section window. A user-defined color is used to draw the graph.

7.5 External variables

The following external variables are used by the visualization functions. The predefined palettes are intended for the indexed color space and, therefore, contain `NINDCOLORS` color definitions.

`rect wwin`

The current world window rectangle. This rectangle is mapped onto the graphics

window to define the world coordinates.

point secpt1, point secpt2

The starting and ending points of the current cross-section.

void *defpalette

A pointer to the default palette array.

void *rbpalette

A pointer to the red-blue palette array. As the index increases, colors gradually change from blue to red.

void *indefpal

A pointer to the palette designed to display an indefinite set of ranges. As the index increases, colors gradually change from blue to blue-green (for the negative ranges), and from red-green (yellow) to red (for the positive ranges), with white between the two groups (the range around zero).

float isopos[NINDCOLORS+1]

A predefined relative isolevel array for definite functions. As the index increases, the elements grow from 0 to 1 in uniform steps.

float isoindef[NINDCOLORS+1]

A predefined relative isolevel array for indefinite functions. As the index increases, the elements grow from -1 to 0 in uniform steps, and from 0 to 1 also in uniform steps. The numbers of negative and positive elements are the same

or differ by one.

7.6 Visualization functions

The visualization functions can be used in the stand-alone mode or through the interactive graphics module. In the stand-alone mode visualization functions are called from the interpreter as any other function would be.

```
void dvec(float vec[], float relisolev[], void *palette)
```

Displays the piecewise bilinear function based on the node vector `vec` using the current representation type. The argument `relisolev` specifies the relative isolevel array to be used. The indexed color space palette is defined by the palette pointer argument `palette`. The current world window `wwin` is used to map the computational plane into the graphics window. The current mirror (if any) is used to reflect the image.

```
void dgrid(int minlev, int maxlev)
```

Plots the supercells of the grid between the levels `minlev` and `maxlev`. The outside contours of the supercells are green, and the internal dividing lines are grey. The current world window `wwin` is used to map the computational plane into the graphics window.

```
void dsect(void)
```

Plots the functions based on the grid vectors which have been marked for the cross-section. The current section interval defined by points `secpt1`, `secpt2` is

used. The cross-section graph color and the y-scale for each function are defined by the current cross-section attributes (Section 4.5) of the corresponding grid vector.

7.7 Postscript printing

Graphics functions have the capability to create PostScript (Adobe Systems Incorporated, 1985,1990) descriptions of the picture they produce in parallel with drawing it on the screen. If the PostScript output is enabled, the PostScript descriptions are written to a disk file with the name specified by the user. The extension `.ps` is added to the name automatically. At present, a PostScript file can be opened and the PostScript output enabled only from the interactive graphics module (for details see 7.8.3).

The resulting PostScript files can be printed on a color or black and white PostScript printer. A PostScript code for printer color capability recognition is included in every file. When printing on a black and white printer, the rendering of colors is automatically done as following. The indexed color space colors are replaced by gray scale levels increasing uniformly with the color index from 0 (black) to 1 (white). The named color space colors are replaced by black except for `GRAY`, `WHITE`, and `BRIGHTWHITE` which are replaced by the gray scale levels 0.33, 0.67, and 1.0, respectively. The color lines on the cross-section graphs are replaced by dashed lines.

Note that PostScript descriptions are not screen dumps. Therefore a hardcopy can have much higher resolution than the corresponding picture on the screen. In

addition, hardcopy pictures can be manipulated and/or combined in various ways by editing PostScript files before printing.

The following external variables allow the PostScript output to be tuned. The lengths on the page are expressed in points (1/72 in). The page coordinate system normally starts at the lower-left corner.

`int copies`

The number of copies to print. The default is 1.

`rect pagevp`

The page viewport rectangle. The graphics window is mapped onto the rectangle. The default rectangle's lower-left corner is at point (0,0), and the upper-right corner is at point (648,468). Logically, the page viewport is located on the page viewport plane, which can be shifted and rotated relative to the page.

`float xshift, float yshift`

The page coordinates of the origin of the page viewport plane. The default coordinates are (540,72).

`float rotation`

The degree of rotation of the page viewport plane relative to the page. The plane is rotated around its origin. The default rotation is 90 degrees.

`float linewidth`

The current linewidth. The default linewidth is 1.

7.8 The interactive graphics module

The interactive graphics module is invoked as the function `graph()` without arguments. The graphics window is opened and the input from the keyboard and the pointing device (a three button mouse) is used to control the operations of the interactive graphics module. On exiting from the interactive graphics module, the graphics window disappears and control returns to the interpreter.

7.8.1 Input processing

A three-button mouse is the preferred tool to control the interactive graphics module. The left button (`BUTTON1`) is used to select a point and to open a pop-up window or a menu, the right button (`BUTTON2`) is used to accept a menu choice or to close a pop-up window, and the middle button (`BUTTON3`) is used to escape and to cancel.

The input events are first put into the input queue from which they are read by the program in first-in-first-out order. The input queue works in two slightly different modes — text mode and graphics mode. In both modes certain keyboard keys are mapped onto mouse buttons: `F1` — the left button, `ESCAPE` — the middle button, and `ENTER` — the right button. The cursor control keys are mapped onto mouse movements — in the graphics mode one keystroke moves the cursor by one pixel in the corresponding direction; in the text mode the cursor is moved by one character position.

Several input processing modes are used throughout the interactive graphics module.

Choice menus. Items are selected by moving the mouse and accepted by BUTTON2. BUTTON3 is used to escape a menu without making a choice.

Input forms. Input forms are pop-up windows that contain annotation fields and input fields. The keyboard is used in the usual way to fill input fields. The cursor is moved between the input fields and inside the input fields by using the cursor keys. Typing can be corrected by DELETE and BACKSPACE keys. An input form is accepted by pressing BUTTON2 and escaped from by pressing BUTTON3.

Output windows. Output windows are used to display information. They are closed by pressing BUTTON2 or BUTTON3.

Scan mode. In the scan mode the cursor controlled by the mouse moves over the graphics window. The scan mode is ended either by pressing BUTTON1, in which case an action based on the current cursor position is taken, or by pressing BUTTON3, in which case the scan mode is exited. If F2 is pressed in the scan mode, the cursor location window is created at the current cursor position. After the cursor location window has been created, the world coordinates of the moving cursor are continuously displayed in it. The cursor location window is closed by pressing F2 the second time.

The cursor location in the scan mode defines several values which are available to the interactive graphics module after BUTTON1 is pressed.

The current point is the point in the current world window corresponding to the current cursor location.

The current grid level is set to be equal to the grid level of the terminal node closest to the current point. It can later be changed in the grid level submode.

The current node is the node on the current grid level closest to the current point.

The current cell is the cell on the highest existing grid level less than or equal to the current grid level that contains the current point. Note that the current node may be one grid level above the current cell.

Grid level submode. This submode is normally used in combination with output windows or input forms. When `PAGEUP` is pressed, the current grid level is increased by 1, up to the maximal grid level at the current point; when `PAGEDOWN` is pressed, the current grid level is decreased by 1, down to level 0. The current node and the current cell are affected by this grid level change.

7.8.2 Space functions

As we have mentioned in Section 4.5, the system maintains the list of space functions that have to be written according to certain rules and can be called from the interactive graphics module. In this section we will describe how to write these functions.

A space function module must include the header files `window.h` and `intgraph.h`, which contain the necessary definitions of external variables and function prototypes.

A space function has access to the information about the current cursor location in space and relative to the grid. This information is passed through the external

variables listed below.

external float xworld, yworld;

The world coordinates of the current point in the scan mode.

extern int xynodelev;

The current grid level.

extern gn_index xynode;

The current node (the node closest to the current point on the current grid level).

extern SUBCELL xysubcell;

The current cell (the cell on the highest existing level less than or equal to the current grid level which contains the current point).

The rubber window mechanism is used for the space function screen output. Under this mechanism text strings, together with their desired positions in the window, are first output into the rubber window list. After a space function has been executed, a pop-up window is created which is large enough to print all the text accumulated in the list at the specified positions. The function **wd_autoprint** outputs text into the pop-up rubber window list. It is the only output operator which should be used for the space function screen output. The rubber window list initialization and the pop-up window creation is done behind the scenes by the interactive graphics module.

int wd_autoprintf(int row, int col,char *fmt, ...)

Outputs text in the rubber window starting from row row and column col. The

function formats the arguments, which follow the format string **fmt** exactly as the C function `printf` does. The number of the output characters is returned.

7.8.3 Main menu

When the function **graph()** is called, the graphics window is created and the main menu appears. The menu offers to select one of the items listed below.

`Quit`

Quit the interactive graphics module.

`Drawing menu`

Call the drawing menu (7.8.4).

`Scan vectors`

Go into the scan mode. When BUTTON1 is pressed a pop-up input form window appears. In its top subwindow the current grid level, the current node, and the current cell are printed. In the bottom subwindow the list of grid vector names is printed. Each vector name is followed by the value of the vector at the current node for the node vectors, or at the current cell for the cell vectors. A value can be changed in the interpreter data space by overtyping the corresponding field in the window. Initially, the current grid level is set to be equal to the grid level of the terminal node closest to the current point. The grid level submode (Section 7.8.1) is used with the window and the grid level can be changed by pressing PAGEUP, PAGEDOWN keys. Pressing BUTTON2 closes the window and

returns to the scan mode; pressing BUTTON3 returns to the main menu.

`Setting menu`

Call the setting menu (7.8.5).

`Grid menu`

Call the grid menu (7.8.6).

`Scan function`

A choice menu appears with the names of declared space functions. Upon the choice of a space function the system goes into scan mode. When BUTTON1 is pressed, the function is executed and its output is printed in a pop-up window. Initially, the current grid level is set to be equal to the grid level of the terminal node closest to the current point. The grid level submode (Section 7.8.1) is used with the window and the grid level can be changed by pressing PAGEUP, PAGEDOWN keys. Pressing BUTTON2 closes the window and returns to the scan mode; pressing BUTTON3 returns to the main menu.

`Mark`

A choice menu appears with the names of colors. After a color is chosen, the scan mode is entered. When BUTTON1 is kept pressed the cursor will leave a trace of the chosen color. Returns to the main menu by pressing BUTTON2 or BUTTON3.

`PostScript`

If the PostScript output file has been closed, the file name input form appears.

After the desired file name is typed, it can be accepted by pressing BUTTON2. The extension .ps is added to the file name automatically. If a file with the identical name already exists, the system asks for confirmation to overwrite it. Otherwise the file is opened and the PostScript output is enabled. All the subsequent drawing functions will write a PostScript description of their output into the file. Note that choice menus will not be written into the file and that clearing the screen writes the eject page command to the PostScript file. To escape the input form without opening the file, press BUTTON3.

If the PostScript output file has been opened, a choice menu appears. On the top of the menu there is a PostScript file information subwindow, in which the file name and the output status are displayed. The menu choices include returning to the main menu, closing the file, or enabling or disabling the PostScript output.

7.8.4 Drawing menu

`Quit`

Return to the main menu.

`Clear screen`

Clear the screen.

`Draw vector`

A choice menu appears with the names of the grid vectors. The function dvec is called to display the chosen vector using the current representation and the

current world window.

`Draw grid`

The function **dgrid(0,maxlevel)** is called to draw the grid.

`Redraw`

Redraw the screen.

`Isolevels`

Depending on the current grid vector representation type, print the isolevels or the isointervals, and the corresponding colors.

`Stamp`

Call a user-defined function that prints the information identifying the current solution.

`Cross-section`

Call the cross-section menu (7.8.7).

`Set zoom`

Go into the scan mode. Two consecutive BUTTON1 clicks define corners of the zoom rectangle. The rectangle is accepted by pressing BUTTON2, in which case it is made the current world window rectangle and the screen is automatically redrawn. The scan mode can be escaped without redefining the rectangle by pressing BUTTON3. If BUTTON2 has been pressed first, without defining a new zoom, the current world window is made equal to the computational domain.

7.8.5 Settings menu

`Quit`

Return to the main menu.

`Set palette`

A choice menu appears with the names of predefined palettes. The indexed color space is switched to the chosen palette.

`Representation`

A choice menu allows the user to choose between colorfill and isocontour representations of grid vectors. The chosen representation will be used for the subsequent drawing.

`Set mirror`

A choice menu allows the user to set a mirror at any of the four sides of the computational domain or to return to the no mirror condition. If a mirror is set, the subsequent grid vector displays will be reflected around the corresponding side.

7.8.6 Grid menu

The grid menu facilitates the examining of the grid data structures and has been extensively used to debug the adaptive grid library.

`Quit`

Return to the main menu.

`Print node`

Go into the scan mode. When `BUTTON1` is pressed the internal data for the current node are displayed in a pop-up window. The grid level submode (Section 7.8.1) is used with the window. Return to the scan mode by pressing `BUTTON2`; return to the menu by pressing `BUTTON3`.

`Print scell`

Go into the scan mode. When `BUTTON1` is pressed the internal data for the supercell to which the current cell belongs are displayed in a pop-up window. The grid level submode (Section 7.8.1) is used with the window. Return to the scan mode by pressing `BUTTON2`; return to the menu by pressing `BUTTON3`.

`Add scell`

Go into the scan mode. When `BUTTON1` is pressed, a new supercell covering the current cell is created using the consistent refinement algorithm.

`Delete scell`

Go into the scan mode. When `BUTTON1` is pressed, the supercell to which the current cell belongs is deleted if it is possible without violating consistency.

7.8.7 Cross-section menu

`Quit`

Return to the drawing menu.

`Clear screen`

Clear the screen.

`Quick plot`

Go into the scan mode. The two consecutive `BUTTON1` clicks define the starting and the ending points of a cross-section. The cross-section is accepted by pressing `BUTTON2`. The screen is erased and the cross-section of the currently displayed vector is drawn. Pressing `BUTTON2` redraws the current grid vector and returns the system to the scan mode to define a new cross-section. The scan mode can be escaped by pressing `BUTTON3`.

This function permits a quick examination of the currently displayed grid vector.

`Choose section`

Go into the scan mode. The two consecutive `BUTTON1` clicks define the starting and the ending points of a cross-section. The cross-section is accepted by pressing `BUTTON2`.

`Section list`

A submenu appears which allows the user to change grid vector cross-section attributes interactively (Section 4.5).

`Plot`

Call the function `dsect` (7.6) to plot the cross-sections of all the marked grid vectors.

Legend

Print the coordinates of the starting and ending points of the current cross-section, followed by the list of the grid vectors marked for cross-section. On each line of the list of marked vectors there are three items. The first item is a line interval drawn in the color used for the cross-section of the grid vector whose name is the second item in the line. The third item is the y-scale attribute for the grid vector.

Stamp

Call a user-defined function which prints the information identifying the current solution.

Chapter 8

Model problems

We apply the numerical methods and software described above to two models which are representative of problems encountered in combustion science.[1] Both problems have local high activity areas of complex shape. The first problem is a model of a Bunsen burner. The second is a model of a premixed flame with decoupled fluid dynamics.

8.1 Bunsen burner

It is well known that the structure of the Bunsen flame is strongly sensitive to the deficient reactant's Lewis number (Le). If the thermal diffusivity of the premixture sufficiently exceeds the molecular diffusivity ($Le > 1$), the flame assumes a continuous luminous conical shape. However, in the opposite case, when molecular diffusivity is high enough ($Le < 1$), the flame appears "open" near its tip (Lewis & von Elbe, 1961, Law et al., 1982, Mizomoto & Yoshida, 1987).

The analytical description based on high activation energy asymptotics succeeded in capturing many basic features of the phenomenon (Sivashinsky, 1975). At

[1] The problems were formulated by Professor G. Sivashinsky.

($Le < 1$) the temperature, reaction rate, and flame speed were found to suffer a dramatic drop as the reaction zone approaches the tip. In order to maintain the flame at the tip, its speed should be equal to that of the oncoming gas flow. In light of the observed tendencies it was plausible to expect that near the tip the flame simply goes out. Further analytical studies of the problem seem to support this assertion (Buckmaster, 1979, Frankel & Sivashinsky, 1984).

For all that, the previous theories being asymptotic in nature were, strictly speaking, not valid at the extremity of the tip. To obtain a comprehensive picture it is therefore very instructive to undertake a direct numerical simulation of the phenomenon. Apart from that, it is important to shed light on a striking numerical result recently obtained for the physically similar but geometrically simpler case of the inward propagating spherical flame (Fernandez et al., 1989). It was found that at $Le < 1$, despite a considerable drop in temperature and in reaction rate, there was no fuel residue of the unburned reactant. Furthermore, the flame speed exhibited a strange nonmonotonicity at the final stage of combustion.

The present numerical simulations of Bunsen flames fully confirm these findings. Despite the strong tendency towards extinction, the flame survives along the whole front. Moreover, there is no leakage of the deficient reactant through the reaction zone.

8.1.1 Mathematical model

We consider a conventional adiabatic, constant density reaction-diffusion model, which in suitably nondimensionalized formulation reads (Sivashinsky, 1975),

$$\frac{\partial T}{\partial t} + \frac{1}{\sin\alpha}\frac{\partial T}{\partial z} = \frac{\partial^2 T}{\partial r^2} + \frac{\nu}{r}\frac{\partial T}{\partial r} + \frac{\partial^2 T}{\partial z^2} + (1-\sigma)\Omega \tag{8.1}$$

$$\frac{\partial C}{\partial t} + \frac{1}{\sin\alpha}\frac{\partial C}{\partial z} = \frac{1}{Le}\left(\frac{\partial^2 C}{\partial r^2} + \frac{\nu}{r}\frac{\partial C}{\partial r} + \frac{\partial^2 C}{\partial z^2}\right) - \Omega \tag{8.2}$$

$$\Omega = \frac{(1-\sigma)^2 N^2}{2Le} C e^{N\left(1-\frac{1}{T}\right)} \tag{8.3}$$

Here T is the nondimensional temperature, referred to T_b — the adiabatic temperature of the burned gas. C is the nondimensional concentration of the deficient reactant, referred to C_0 — the concentration of the reactant in the fresh mixture. r, z, t are the nondimensional spatio-temporal coordinates referred to D_{th}/U_b and D_{th}/U_b^2, respectively. D_{th} is the thermal diffusivity of the mixture. U_b is the normal speed of a planar flame. $\sigma = T_0/T_b$, where T_0 is the temperature of the fresh mixture. N is the nondimensional activation energy. α is the angle of inclination of the flame interface to the symmetry axis at large r (Figure 8.1). Ω is the nondimensional chemical reaction rate, normalized to insure that the nondimensional speed of a planar flame is nearly unity at large N. For the axisymmetric burner, $\nu = 1$. We shall also discuss the slot burner case, for which $\nu = 0$.

The numerical problem will be considered in the domain $0 < r < r_1$, $z_0 < z < z_2$. For large r, the temperature and the concentration fields are nearly identical to $T^{(0)}$, $C^{(0)}$ pertinent to an inclined planar flame. Hence, for large r_1, the natural boundary

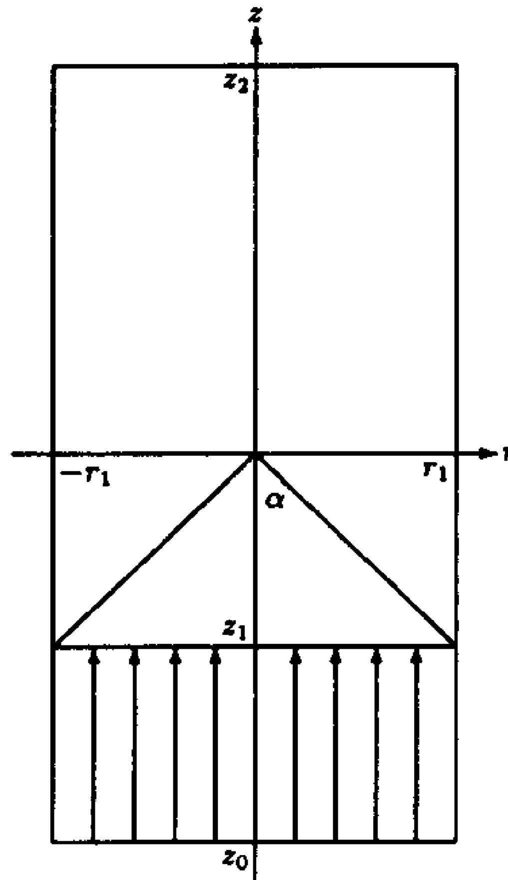


Figure 8.1: Bunsen burner configuration.

conditions for the system (8.1–8.3) at $r = r_1$ are

$$T(r_1, z, t) = T^{(0)}(z \sin\alpha + r_1 \cos\alpha) \tag{8.4}$$

$$C(r_1, z, t) = C^{(0)}(z \sin\alpha + r_1 \cos\alpha) \tag{8.5}$$

In this study $T^{(0)}$ and $C^{(0)}$ are approximated by the profiles provided by the high activation energy theory (Sivashinsky, 1975).

Hence,

$$T^{(0)}(r_1,z) = \begin{cases} 1, & z_1 \le z \le z_2 \\ \sigma + (1-\sigma)e^{(z\sin\alpha + r_1\cos\alpha)}, & z_0 \le z \le z_1 \end{cases} \tag{8.6}$$

$$C^{(0)}(r_1,z) = \begin{cases} 0, & z_1 \le z \le z_2 \\ 1 - e^{Le(z\sin\alpha + r_1\cos\alpha)}, & z_0 \le z \le z_1 \end{cases} \tag{8.7}$$

The parameters z_0, z_1, z_2 are chosen as (Figure 8.1)

$$z_0 = -2r_1\tan^{-1}\alpha, \quad z_1 = r_1\tan^{-1}\alpha, \quad z_2 = 2r_1\tan^{-1}\alpha \tag{8.8}$$

The remaining boundary conditions are taken as

$$\frac{\partial T}{\partial r} = \frac{\partial C}{\partial r} = 0 \qquad \text{at } r = 0 \tag{8.9}$$

$$\frac{\partial T}{\partial z} = \frac{\partial C}{\partial z} = 0 \qquad \text{at } z = z_2 \tag{8.10}$$

$$T = \sigma, C = 1 \qquad \text{at } z = z_0 \tag{8.11}$$

The problem (8.1–8.3) (8.4–8.5) (8.9–8.11) was solved for the following set of parameters: $r_1 = 10$; $\sigma = 0.2$; $N = 10$; $\alpha = \pi/4, \pi/6, \pi/12$; $Le = 1, 1.5, 0.5$; $\nu = 1, 0$.

8.1.2 Discretization

The equations (8.1–8.2) are first discretized in time by Euler's method (Conte & de Boor, 1980). The resulting equations are then discretized in space using the finite element method (Strang & Fix, 1973, Zienkiewicz, 1977) with the basis functions and

the upstream corrected weight functions described in Section 3.

The unknown functions T and C are sought as linear combinations of the piecewise bilinear discontinuous basis functions ϕ_i of space variables r and z described in Section 3

$$T = \sum T_i \phi_i, \tag{8.12}$$

$$C = \sum C_i \phi_i \tag{8.13}$$

where T_i and C_i are the node based parameters to be computed.

Applying the weighted residual method combined with a mass-lumped approximation and Euler's method for the time derivative, we obtain

$$T_i^{n+1} = T_i^n + h_n^t \sum_{k \in \mathcal{K}_i} \left(\left(d_{ik}^r + d_{ik}^z + \frac{\nu}{r} g_{ik}^r - \frac{g_{ik}^z}{\sin\alpha} + s_{ik}^T \right) T_k + s_{ik}^C C_k \right) \tag{8.14}$$

$$C_i^{n+1} = C_i^n + h_n^t \sum_{k \in \mathcal{K}_i} \left(\left(\frac{1}{Le} \left(d_{ik}^r + d_{ik}^z + \frac{\nu}{r} g_{ik}^r \right) - \frac{g_{ik}^z}{\sin\alpha} + s_{ik}^T \right) T_k + s_{ik}^C C_k \right) \tag{8.15}$$

where the coefficients d_{ik}^r, d_{ik}^z representing the second derivatives and the coefficients g_{ik}^r, g_{ik}^z representing the first derivatives are calculated by integrating simple functions

$$d_{ik}^r = \int_{Q_i \cap S_k} \frac{\partial \psi_i^u}{\partial r} \frac{\partial \phi_k}{\partial r}, \quad d_{ik}^z = \int_{Q_i \cap S_k} \frac{\partial \psi_i^u}{\partial z} \frac{\partial \phi_k}{\partial z} \tag{8.16}$$

$$g_{ik}^r = \int_{Q_i \cap S_k} \psi_i^u \frac{\partial \phi_k}{\partial r}, \quad g_{ik}^z = \int_{Q_i \cap S_k} \psi_i^u \frac{\partial \phi_k}{\partial z} \tag{8.17}$$

The coefficients s_{ik}^T, s_{ik}^C representing the chemical reaction term are computed after the term is linearized inside each computational cell (we omit the formulas).

8.1.3 Time step

A stable time step is chosen according to the formula

$$h^t = \frac{1}{2}\,\frac{\min(Le,1)}{(h^r_{min})^{-2}+(h^z_{min})^{-2}} \tag{8.18}$$

where h^r_{min} and h^z_{min} are the sizes of the finest cell of the grid.

8.1.4 Grid refinement criteria

Omitting tedious details, the grid refinement criteria for the cell j is taken to be

$$w_j = \max\left(\frac{\partial^2 T}{\partial r^2}(h^r_j)^2, \frac{\partial^2 T}{\partial z^2}(h^z_j)^2, \frac{\partial^2 C}{\partial r^2}(h^r_j)^2, \frac{\partial^2 C}{\partial z^2}(h^z_j)^2\right) \tag{8.19}$$

where the derivatives are the averages of the numerical derivatives of the current solution computed at the cell's nodes. The cell is refined if w_j is greater than the given refinement tolerance. Some of the cell neighbours may have to be refined as well, in order to preserve consistency. If w_j is less than the given unrefinement tolerance for all the four subcells of a previously refined cell, the refinement is reversed, unless the unrefinement operation would violate grid consistency. An example of an automatically generated grid is shown in Figure 8.2.

8.1.5 Numerical experiments

As one would expect, we found the Bunsen flame structure to be highly sensitive to the Lewis number. Figures 8.3–8.14 demonstrate this by showing the spatial distribution

for temperature, concentration, and reaction rate for $Le = 1$, $Le = 1.5$ and $Le = 0.5$ for the axisymmetric geometry at $\alpha = \pi/6$. For larger ($\alpha = \pi/4$) and smaller ($\alpha = \pi/12$) angles no qualitative changes were observed.

At $Le = 1$ (Figures 8.3–8.6) the width of the reaction zone and its mean temperature practically do not change along the flame front.

At $Le = 1.5$ (Figures 8.7–8.10), one observes the region of elevated temperature near the tip and a marked shrinking of the reaction zone.

The most interesting situation emerges for $Le = 0.5$. Here the temperature drops near the tip, forming an elongated region of reduced temperature. The reaction zone width expands towards the tip, while its intensity suffers a dramatic decrease. As a result, the flame is visually perceived as being extinguished at the tip (Figure 8.13).[2] This impression however is misleading. Despite the sharp reduction in its intensity, the reaction manages to consume all the fuel brought to the reaction zone. This is readily seen in Figure 8.12 and more persuasively in Figure 8.14 depicting the $T - C - \Omega$ profiles at $r = 0$ and at $r = 0.75r_1$. Note that at $r = 0$ the temperature at the reaction zone drops to 75% of its maximum value, while the reaction rate loses more than 95% of its maximal intensity.

The speed of the finite-width curved flame is a somewhat ambiguous concept. In principle, the normal speed of any isoconcentration interface may be identified as a flame speed. However, regardless of the definition, Figure 8.12 clearly indicates that at $Le = 0.5$ the flame speed varies nonmonotonously along the front from a gradual

[2]Locally this picture is quite akin to that recently obtained by Denet and Handelwang (1989,1990) in their simulation of cellular flames spontaneously occurring in low Lewis number premixtures.

decrease[3] to a steep rise near the tip.

In the slot burner case ($\nu = 0$), the Lewis number conditioned trends discussed above, while definitely being preserved, become noticeably weakened. Figures 8.15–8.18 depict the pertinent $T - C - \Omega$ distributions and the profiles corresponding to $\nu = 0$, $Le = 0.5$, and $\alpha = \pi/6$.

8.1.6 Discussion

The numerical simulations conducted in this study unveil a novel feature of the low Lewis number Bunsen flames — their ability to withstand high curvature at the tip without being extinguished. Physically this result seems to be quite understandable. Unburned fuel cannot survive in the high temperature environment maintained by the undepleted parts of the flame adjacent to the tip. It is curious that such a simple argument against extinction has not been raised earlier. The flame opening observed in experiments thus may well be merely an optical illusion conditioned by the considerable drop in luminosity at the tip. It would be of great interest to check experimentally whether there is a penetration of the unburned fuel through the flame. As far as we know such a study has never been undertaken.

From a somewhat different perspective, the results obtained pose a serious challenge to the so-called stretch induced mechanism of flame extinction. This mechanism, while successfully describing the extinction of planar counterflow flames (Sivashinsky,

[3]The minimal velocity may well even become negative, while the flame tip assumes the curious diffusion-dominated bulbous shape found by Buckmaster and Crowley (1983) in their study of strongly elongated Bunsen flames.

1976, Sato & Tsuji, 1978, Buckmaster, 1979, Law et al., 1981), seems insufficient to ensure the extinction of curved flames. On the other hand, experiments (Sokolik, 1967, Chomiak & Jarosinsky, 1982, Abdel-Gayed & Bradley, 1985,1989) and some theoretical estimates (Berestycki & Sivashinsky, 1991) on turbulent flames demonstrate that highly curved flames indeed may be quenched provided the level of turbulence is high enough. It seems plausible to suggest that the quenching here is partially linked to the effectively finite scale (Taylor microscale) of the underlying flow field. In such a situation the low-temperature portions of the flame may easily compete with those of high temperature, thereby promoting total cooling of the system. In the case of the effectively infinite Bunsen cone discussed in this paper, such a development is clearly unfeasible.

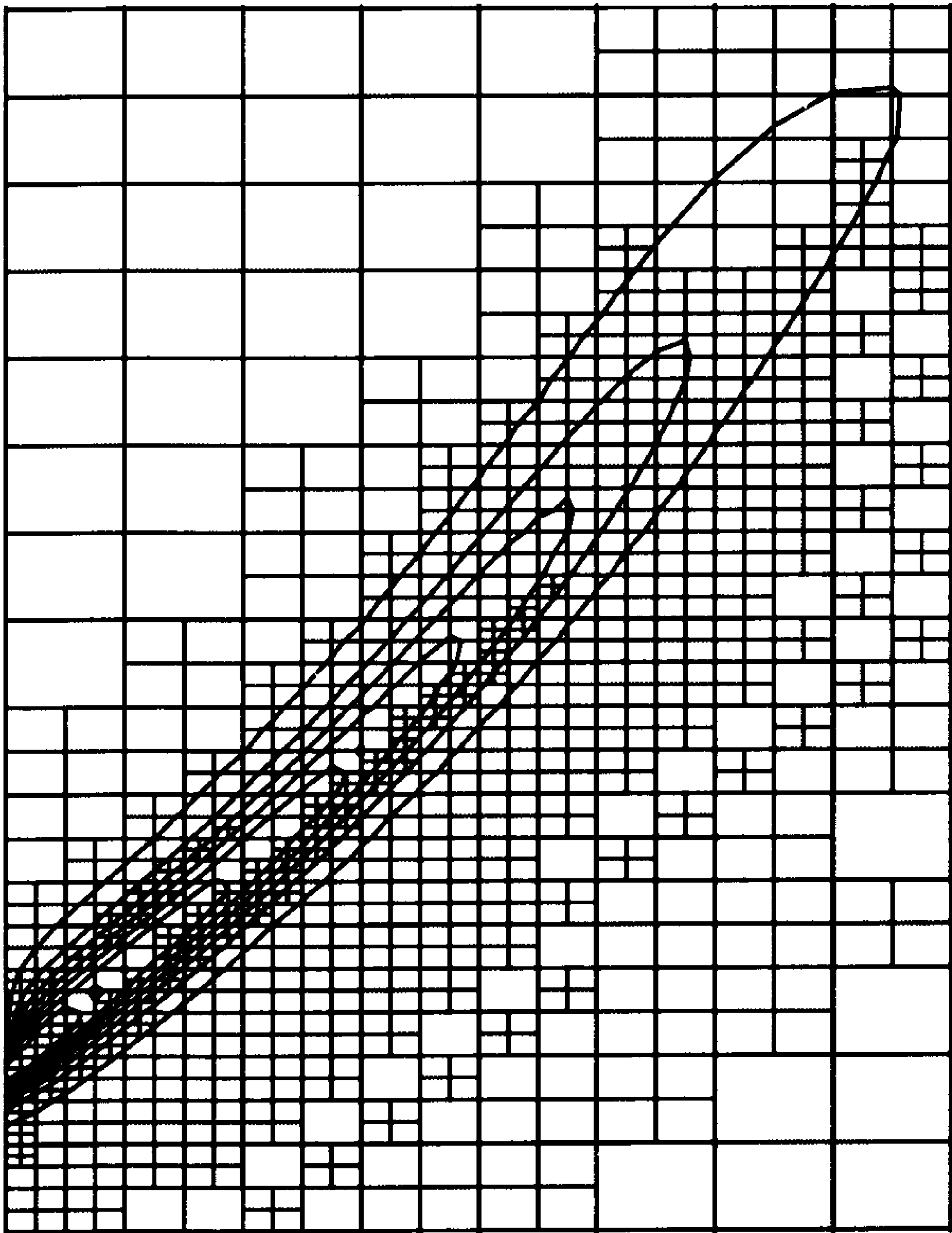


Figure 8.2: Zoom of automatically generated grid for the axisymmetric Bunsen burner case ($\nu = 1$, $\alpha = \pi/4$, $Le = 0.5$, $-10 \leq r \leq 10$, $-20 \leq z \leq 20$) superimposed over the iso-contours of the reaction rate distribution. The zoom dimensions are $2 \leq r \leq 10$, $-12 \leq z \leq 2$.

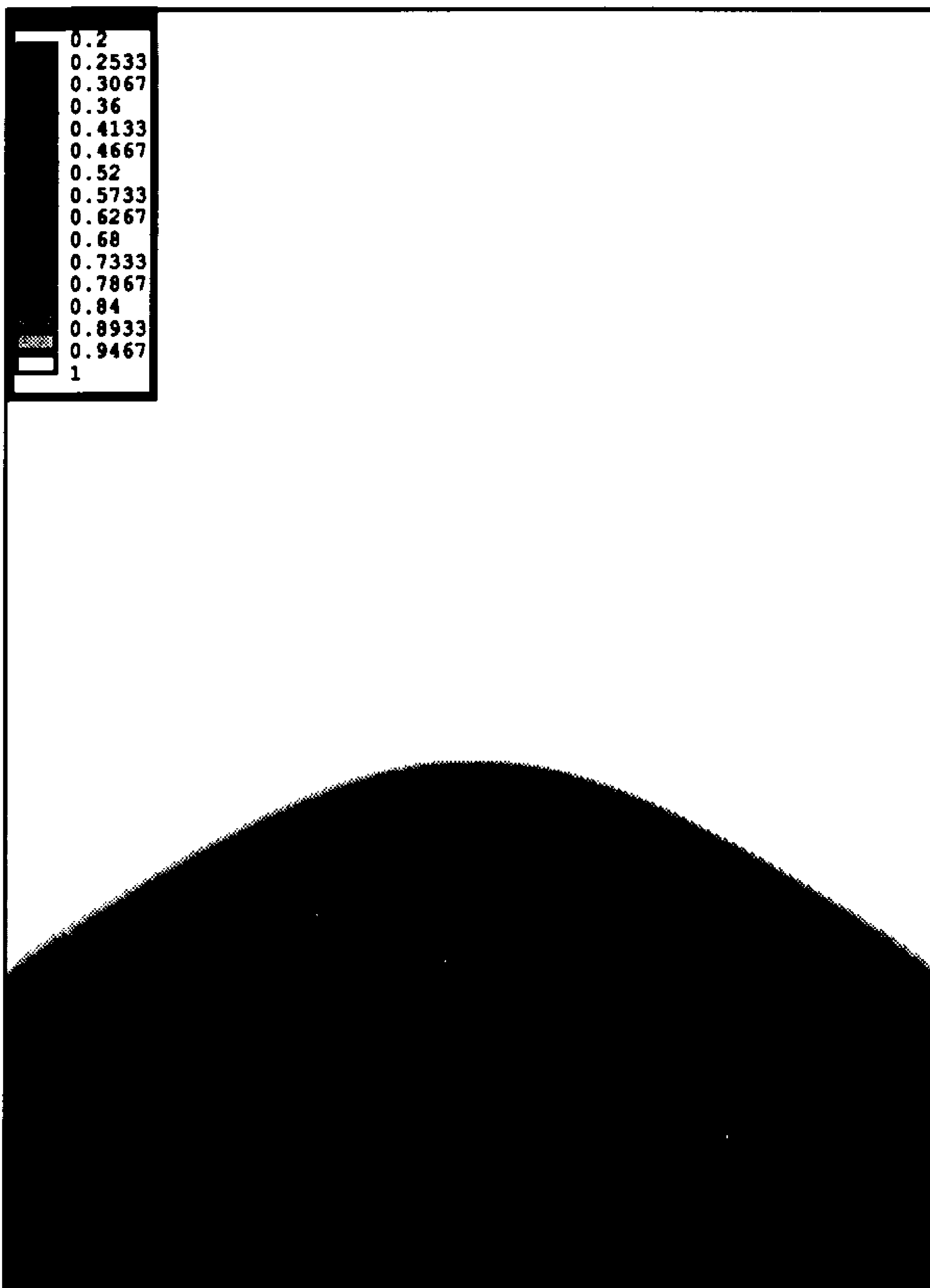


Figure 8.3: Temperature distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 1$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$).

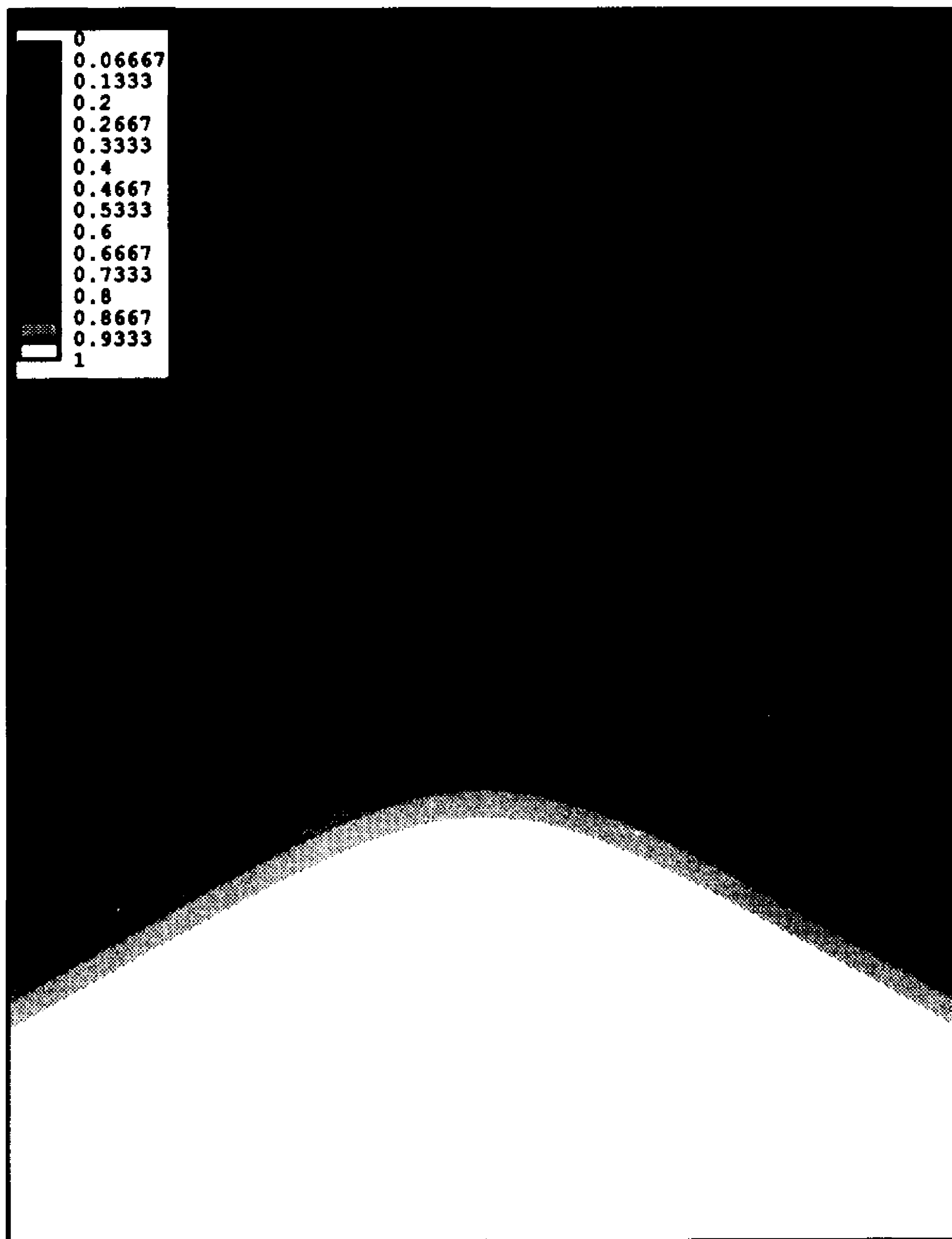


Figure 8.4: Concentration distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 1$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$).

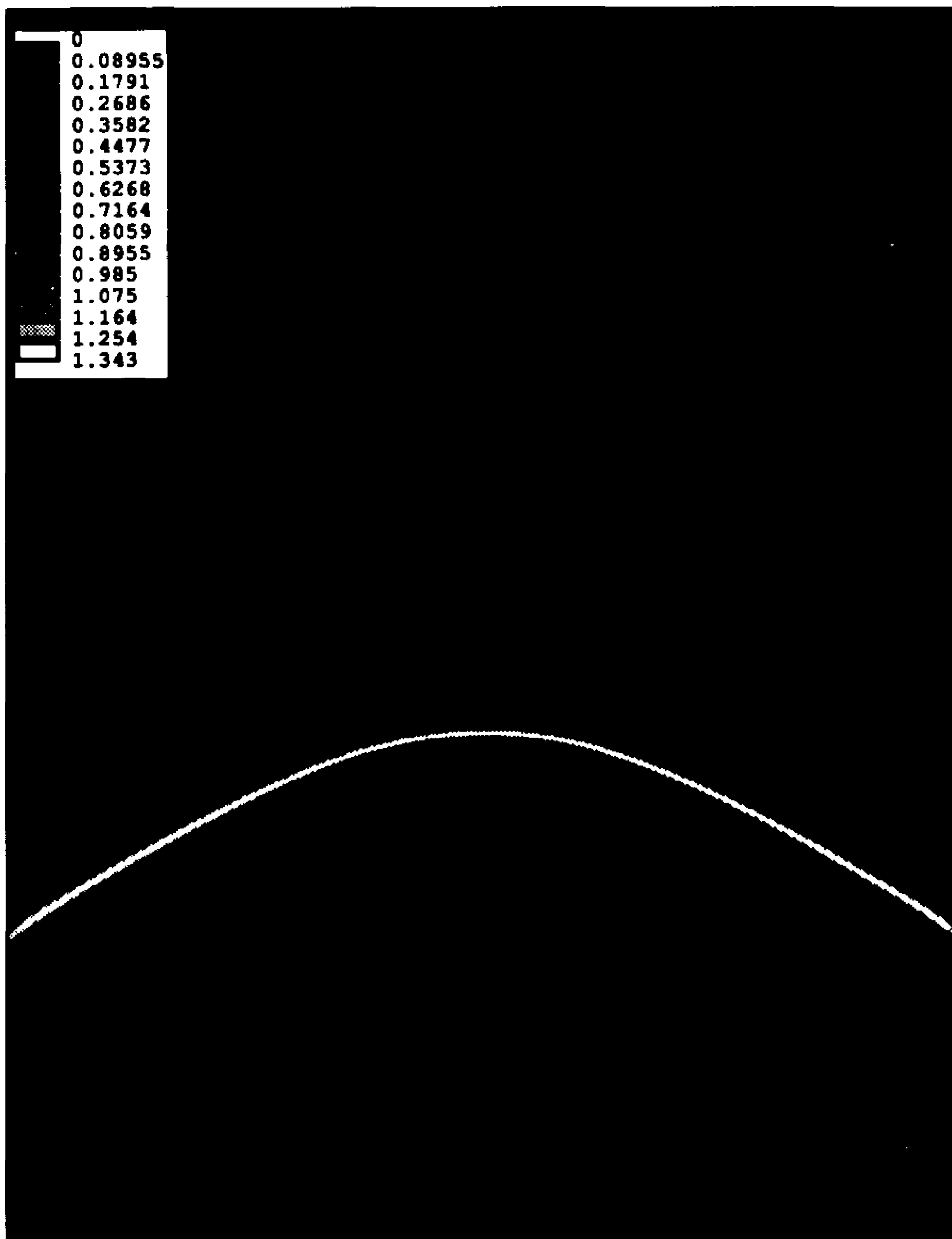
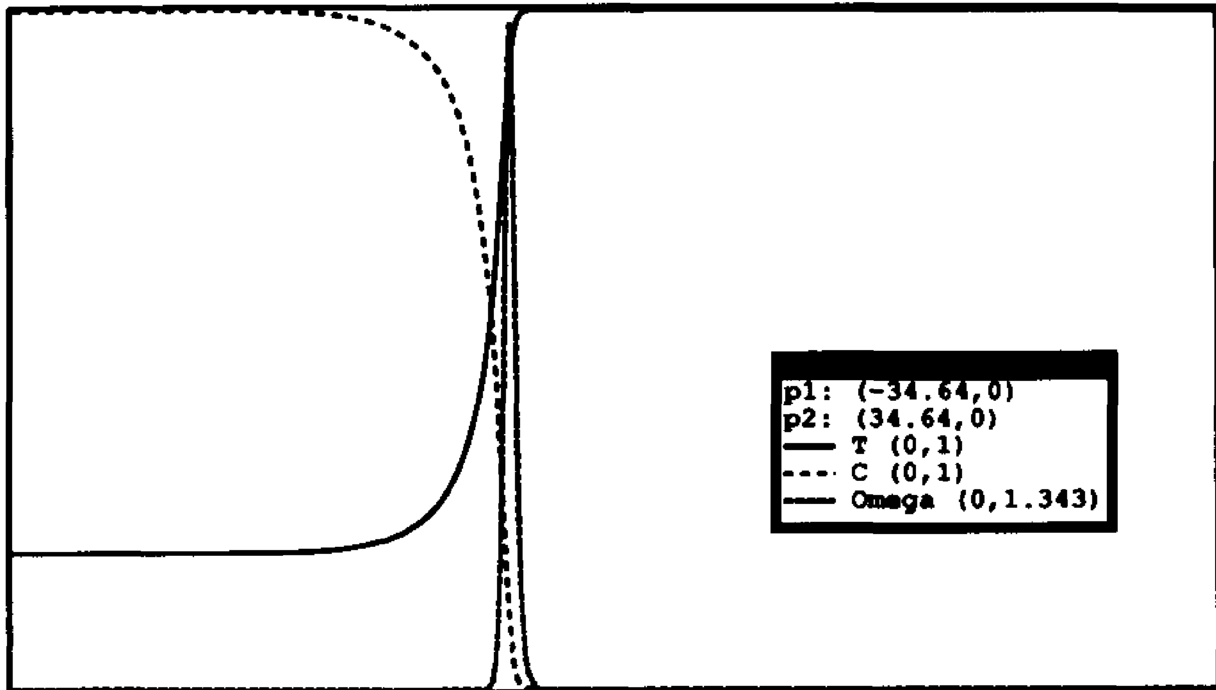


Figure 8.5: Reaction rate distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 1$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$).



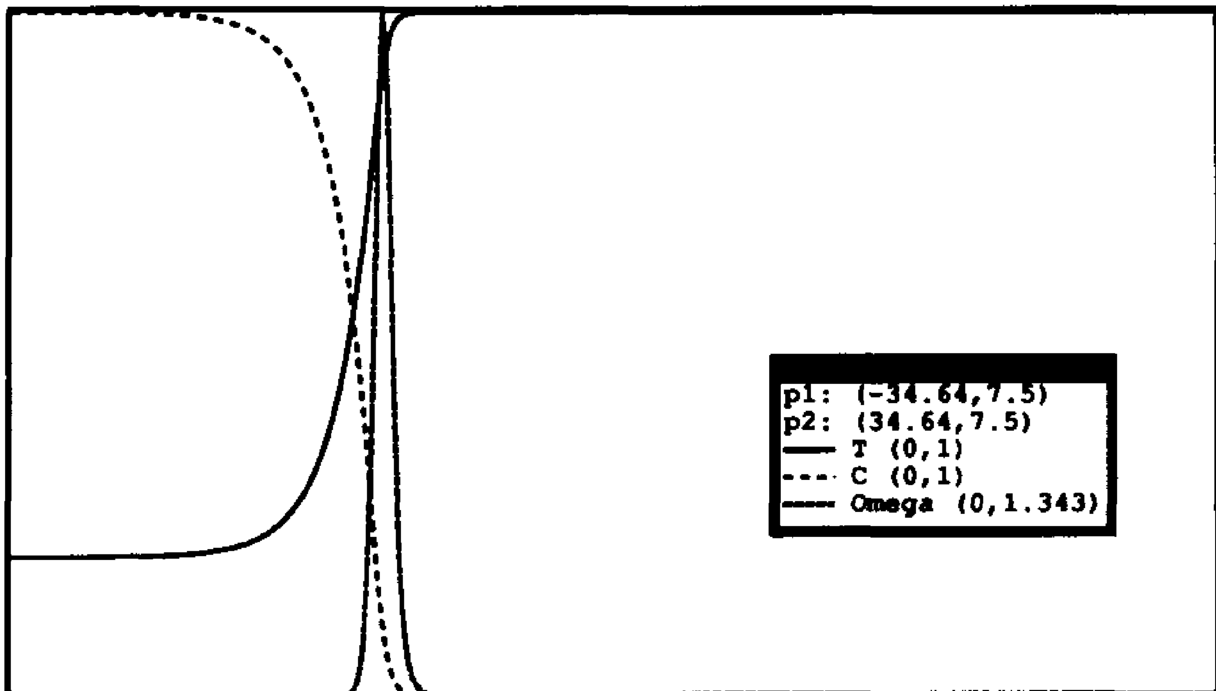


Figure 8.6: Temperature (T), concentration (C), and reaction rate (Ω) profiles for the axisymmetric Bunsen burner at $r = 0$ (a) and $r = 7.5$ (b) ($\nu = 1$, $\alpha = \pi/6$, $Le = 1$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). Note that T, C and Ω are scaled differently — while T and C vary within the interval $(0, 1)$, Ω varies within $(0, 1.343)$.

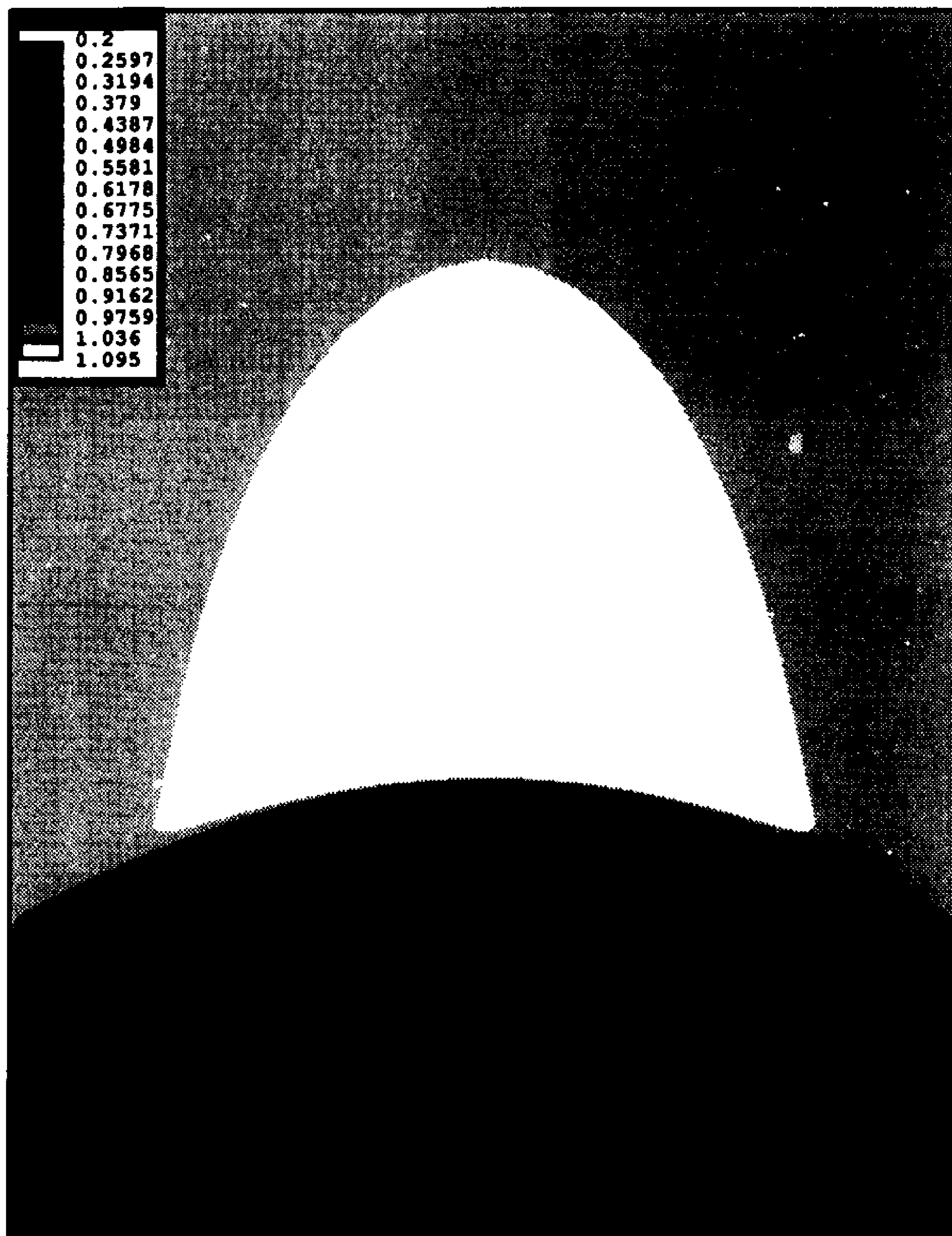


Figure 8.7: Temperature distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 1.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$).

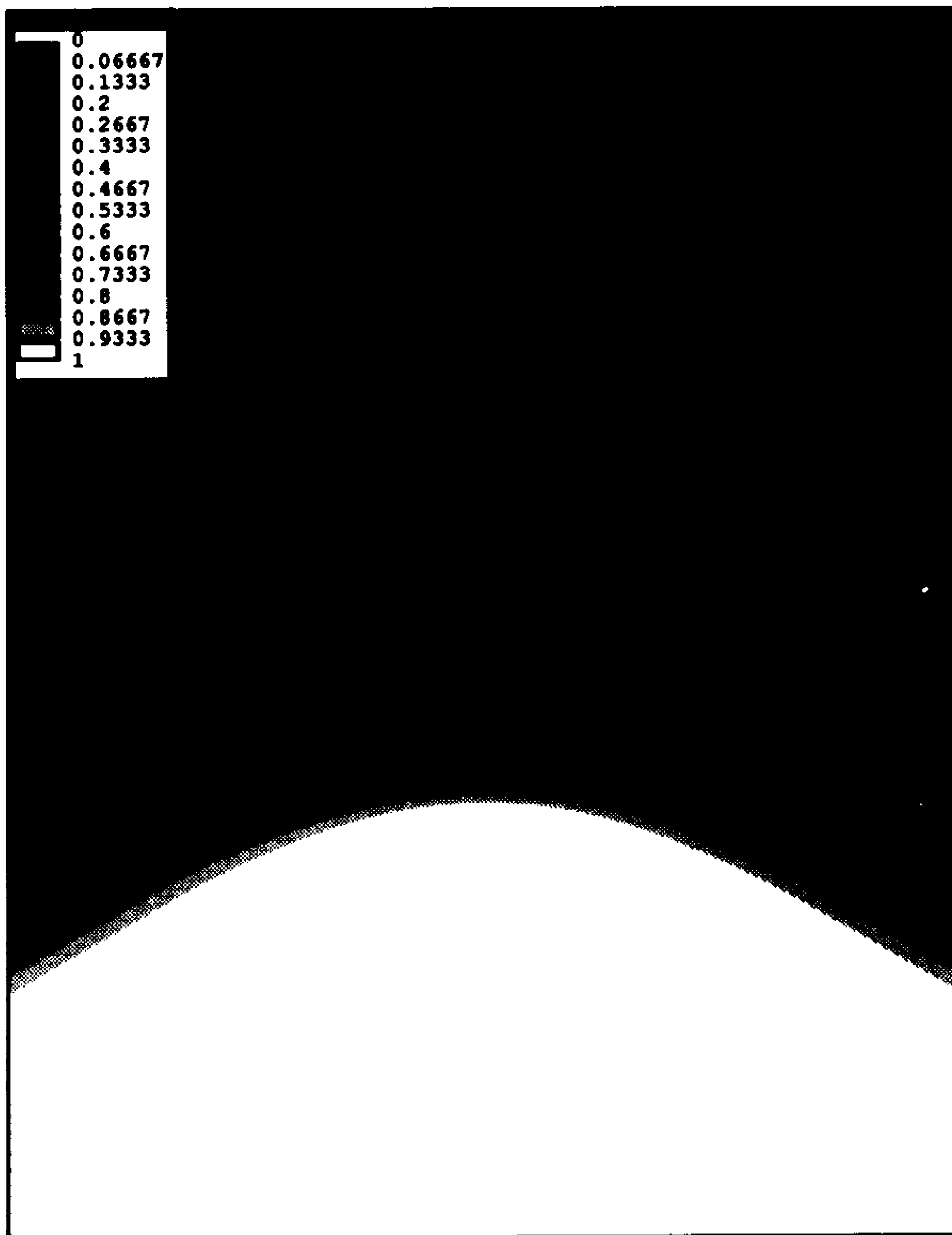


Figure 8.8: Concentration distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 1.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$).

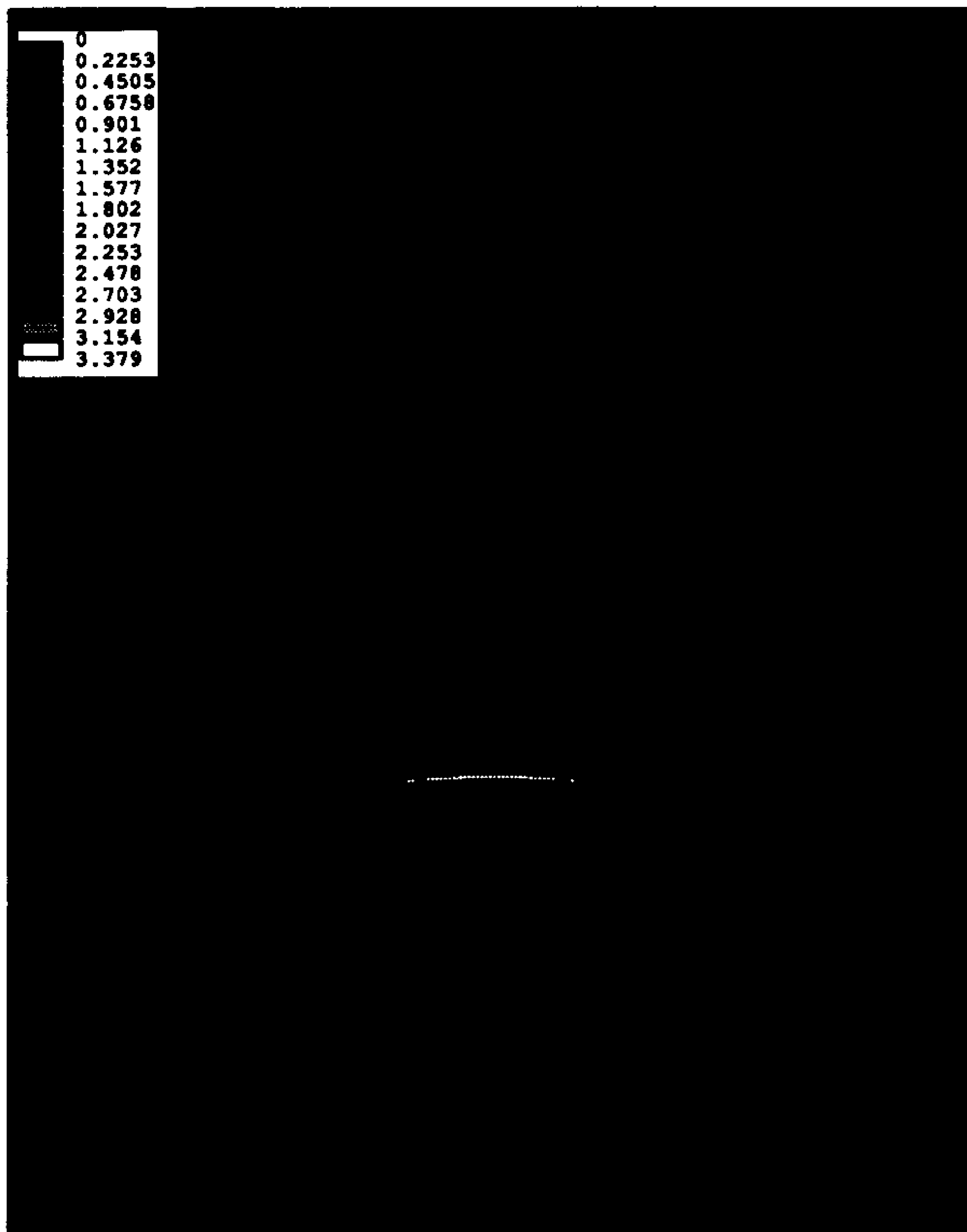
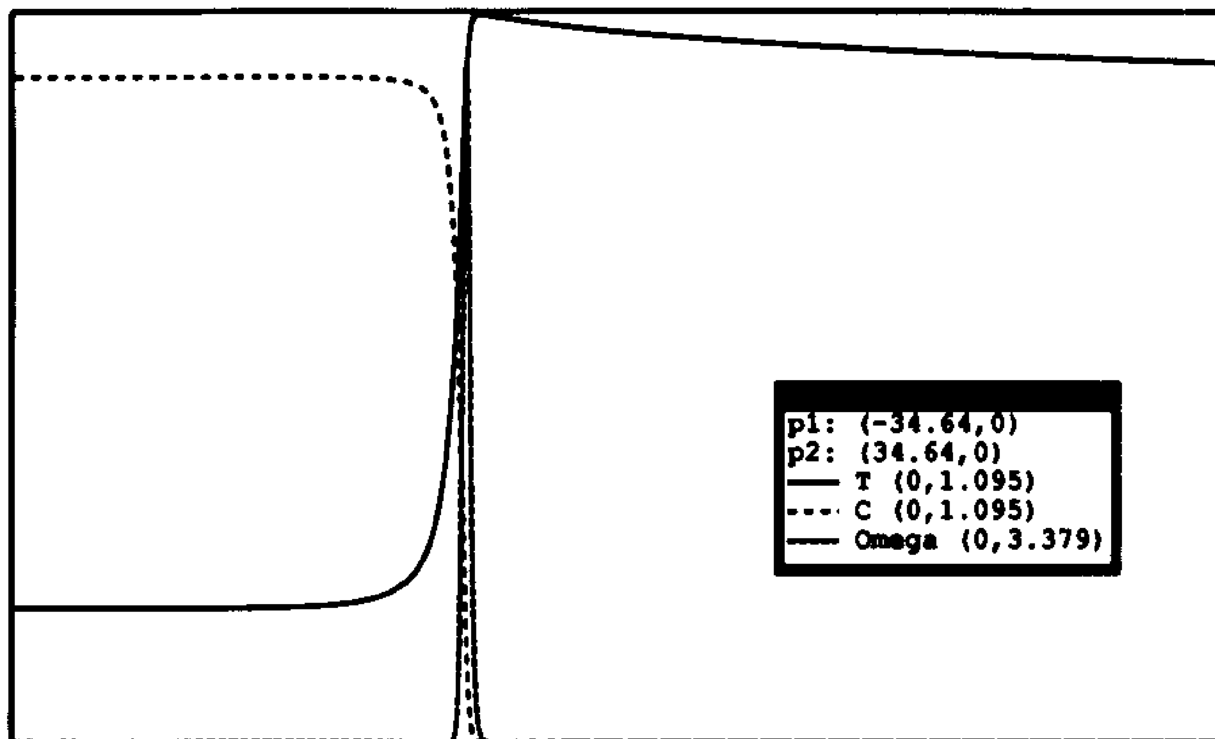


Figure 8.9: Reaction rate distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 1.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$).



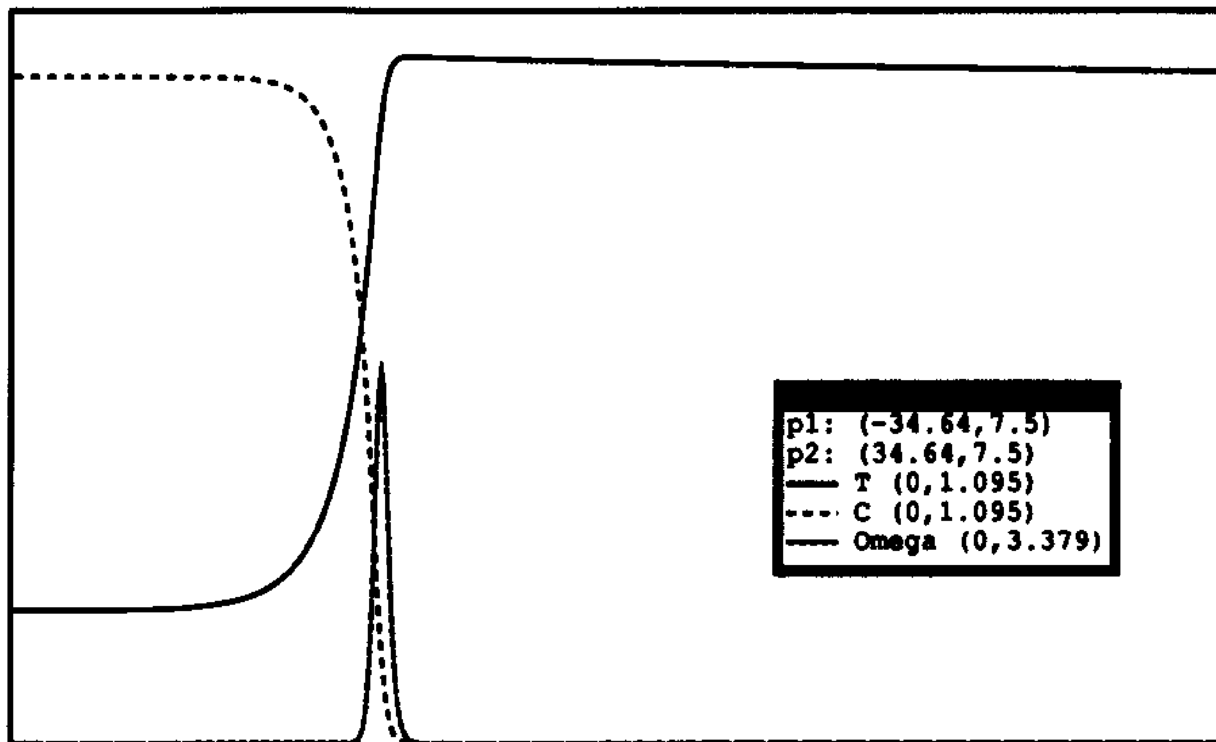


Figure 8.10: Temperature (T), concentration (C), and reaction rate (Ω) profiles for the axisymmetric Bunsen burner at $r = 0$ (a) and $r = 7.5$ (b) ($\nu = 1$, $\alpha = \pi/6$, $Le = 1.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). T and C vary within the interval $(0, 1.095)$, while Ω varies within $(0, 3.379)$.

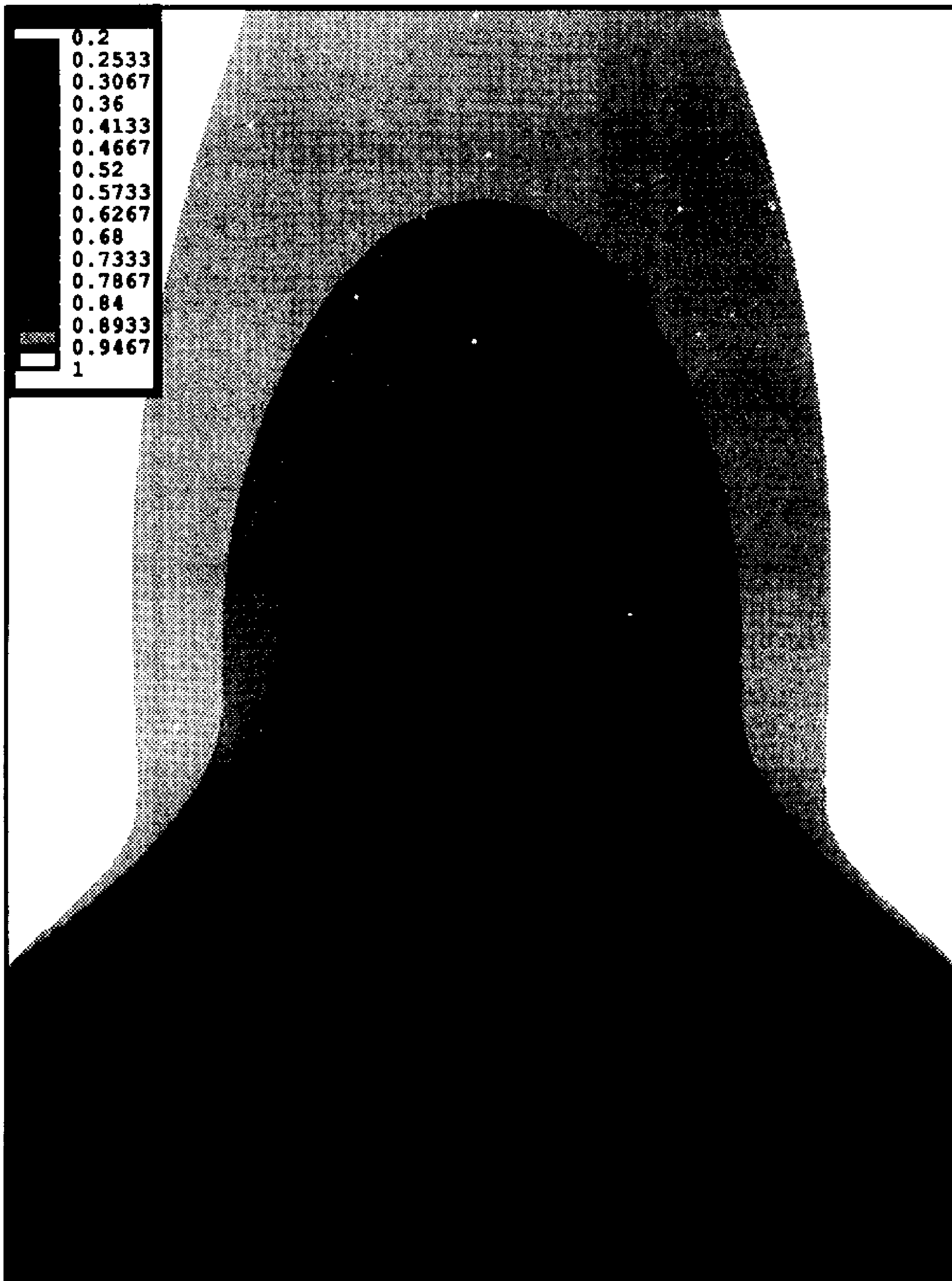


Figure 8.11: Temperature distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$).

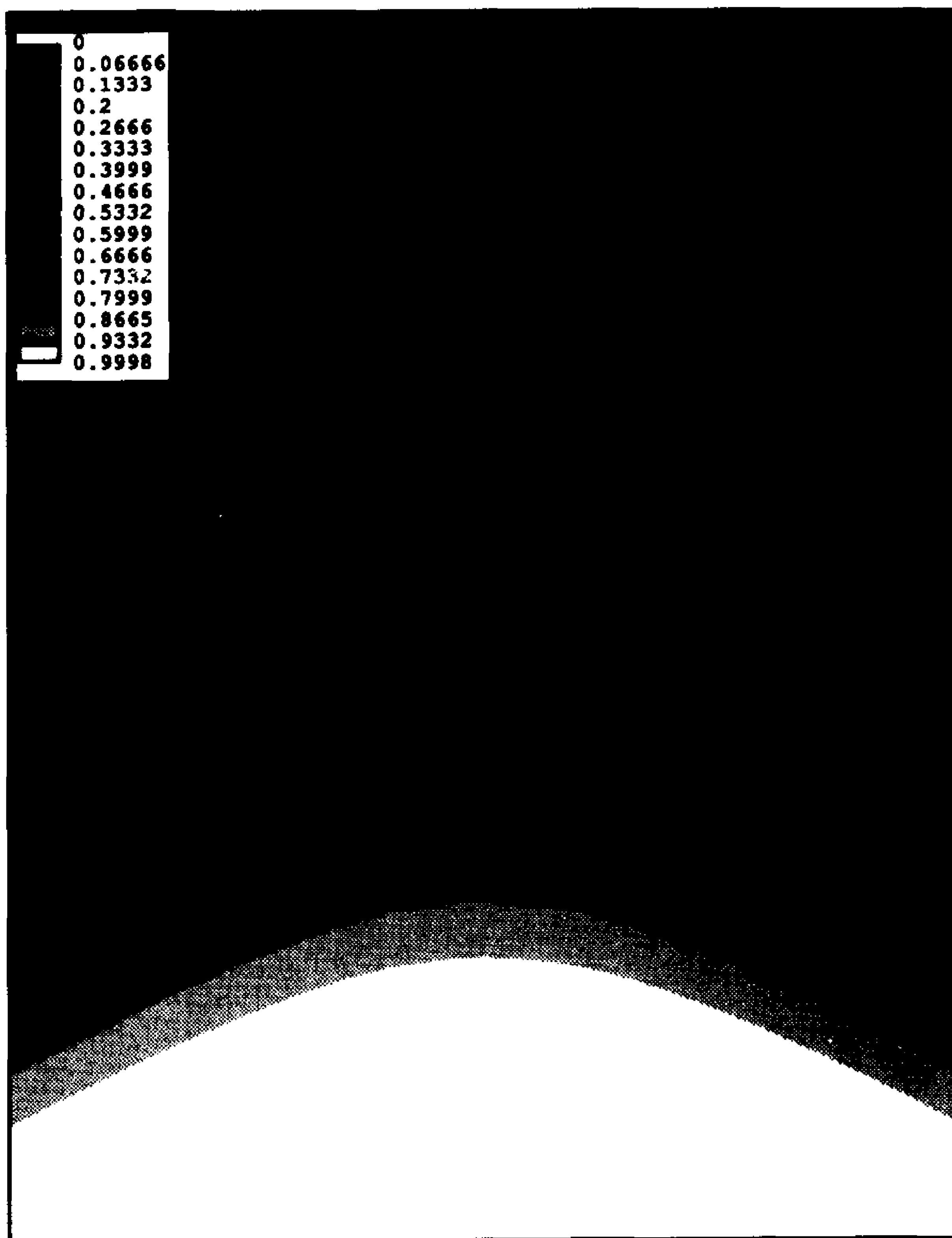


Figure 8.12: Concentration distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $L\epsilon = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$).

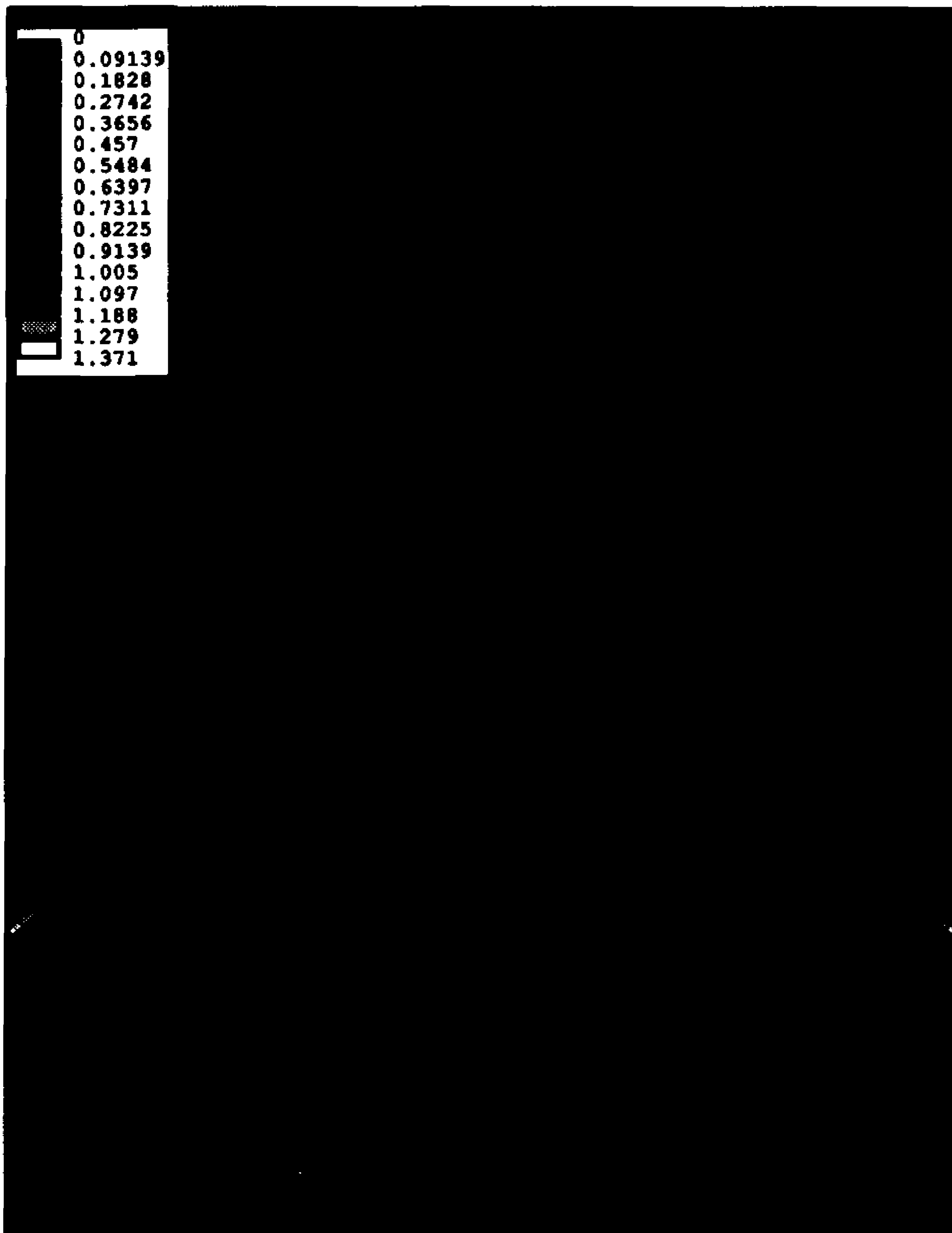
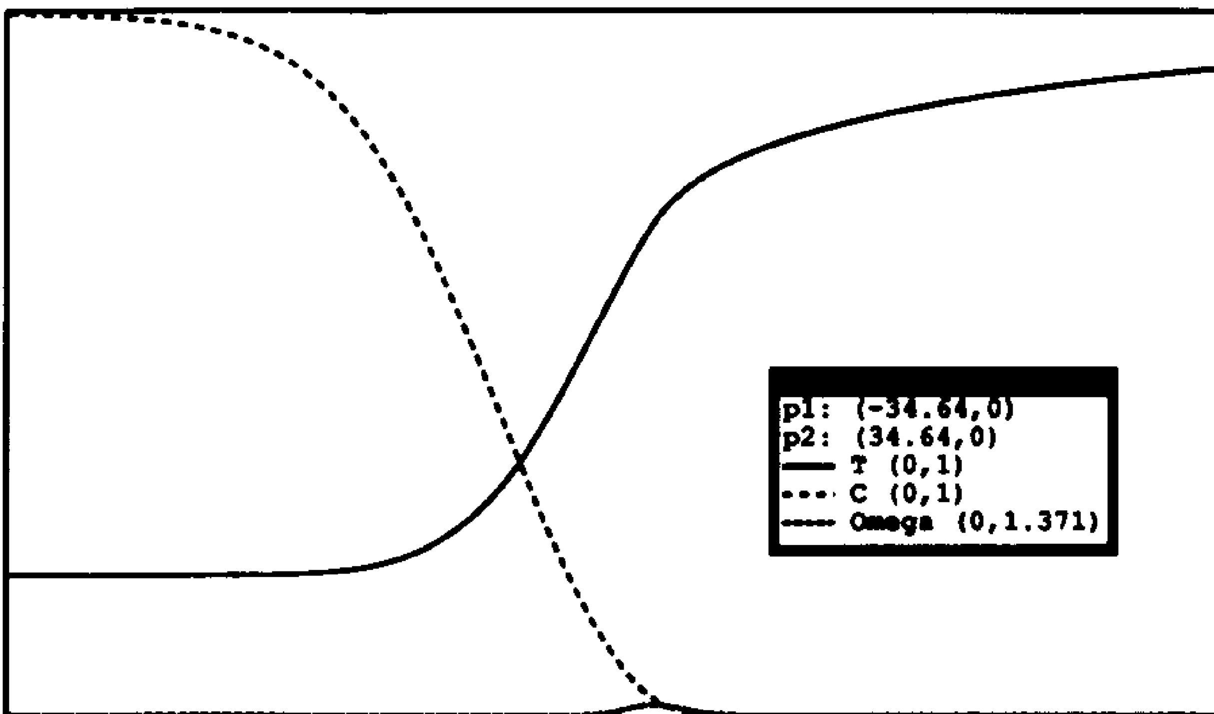


Figure 8.13: Reaction rate distribution for the axisymmetric Bunsen burner ($\nu = 1$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$).



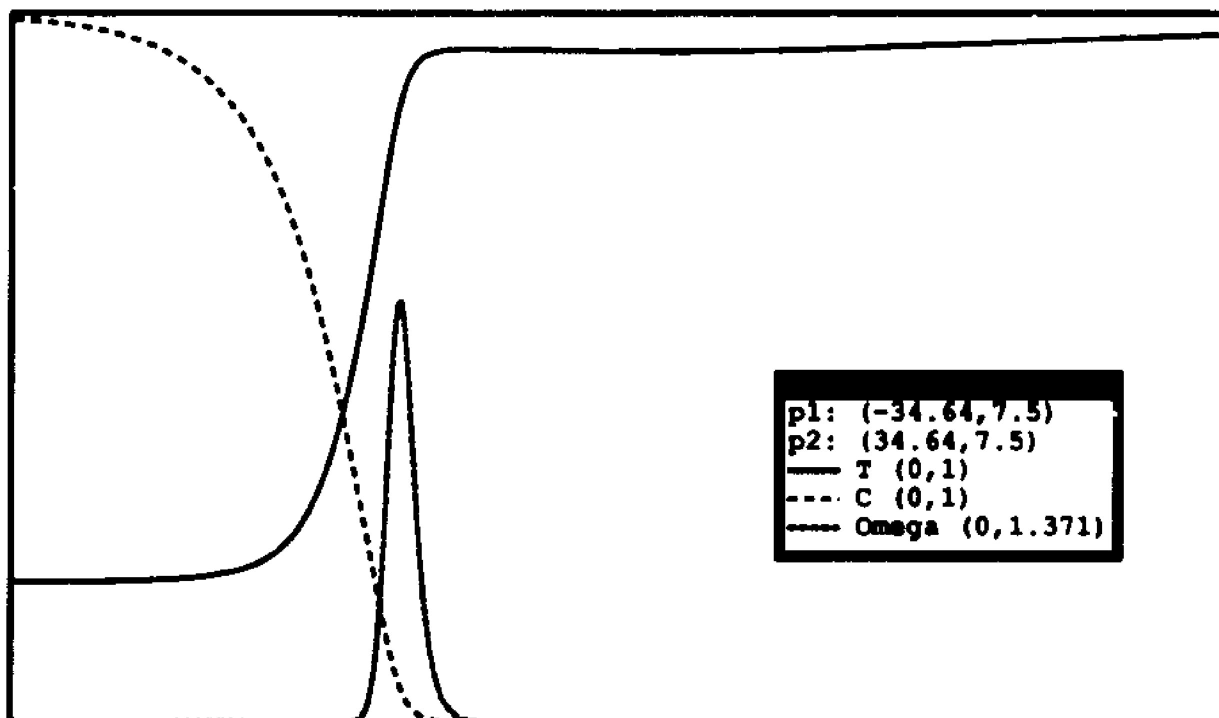


Figure 8.14: Temperature (T), concentration (C), and reaction rate (Ω) profiles for the axisymmetric Bunsen burner at $r = 0$ (a) and $r = 7.5$ (b) ($\nu = 1$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). T and C vary within the interval $(0, 1)$, while Ω varies within $(0, 1.371)$.

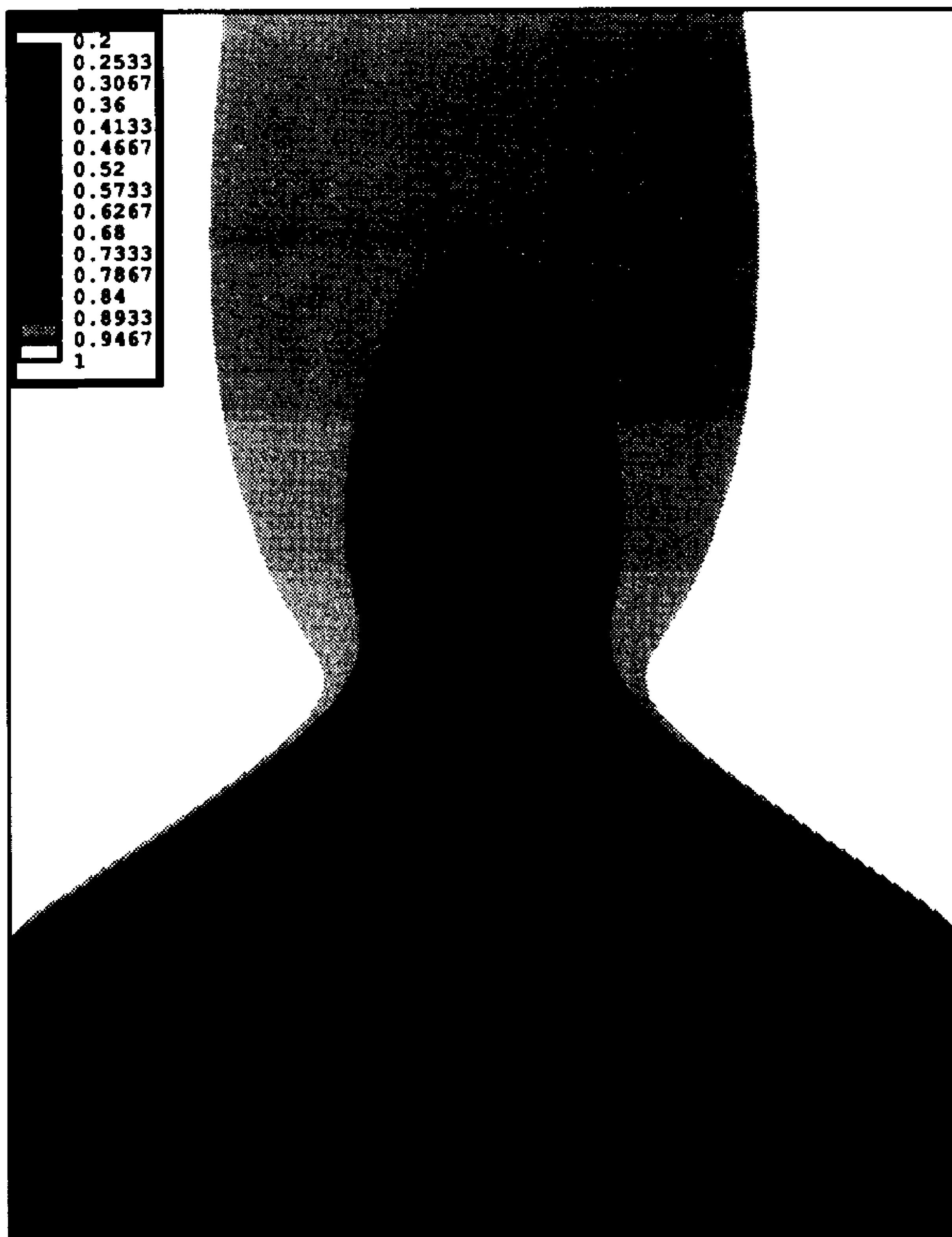


Figure 8.15: Temperature distribution for the slot Bunsen burner ($\nu = 0$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$).

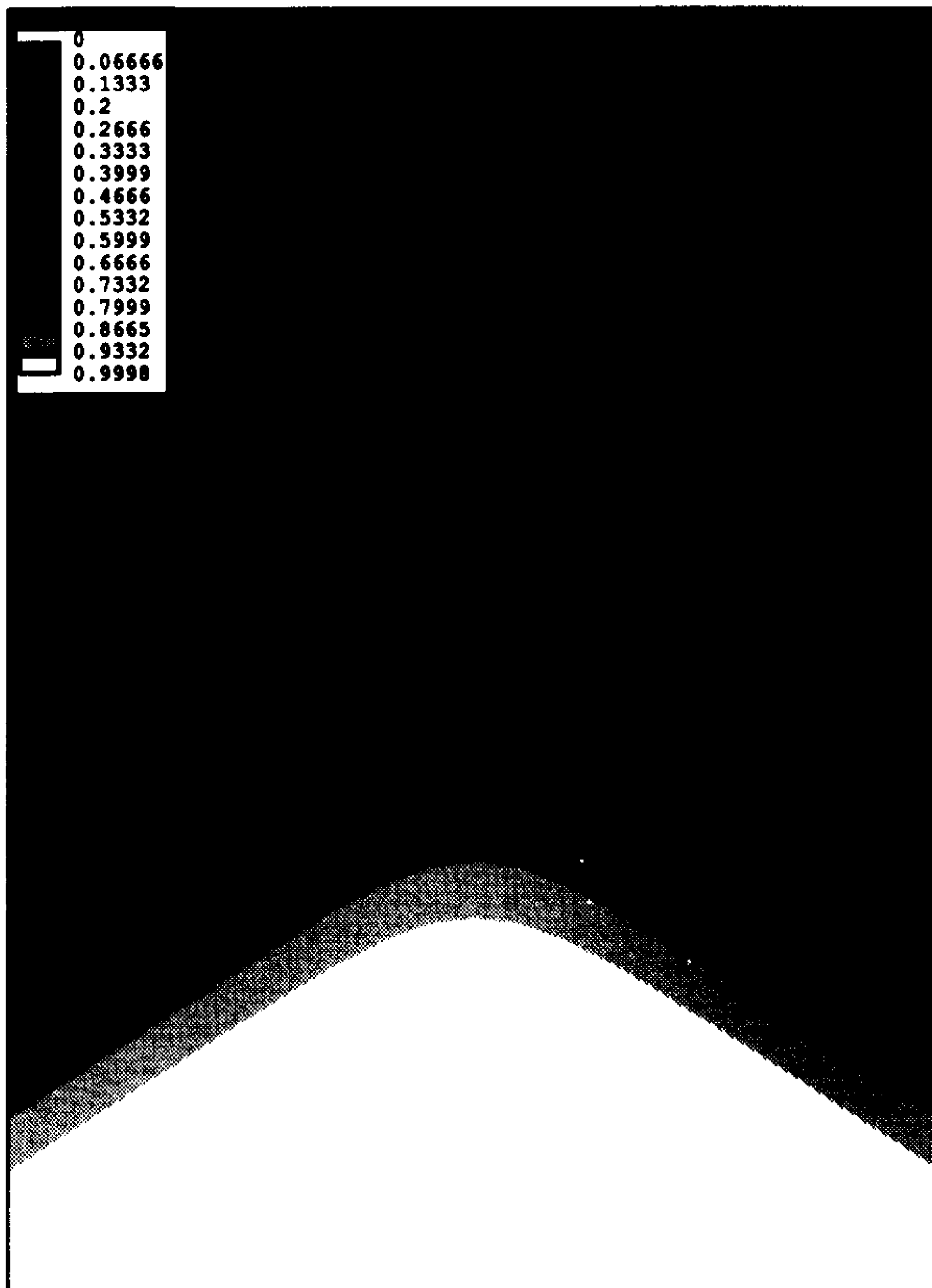


Figure 8.16: Concentration distribution for the slot Bunsen burner ($\nu = 0$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$).

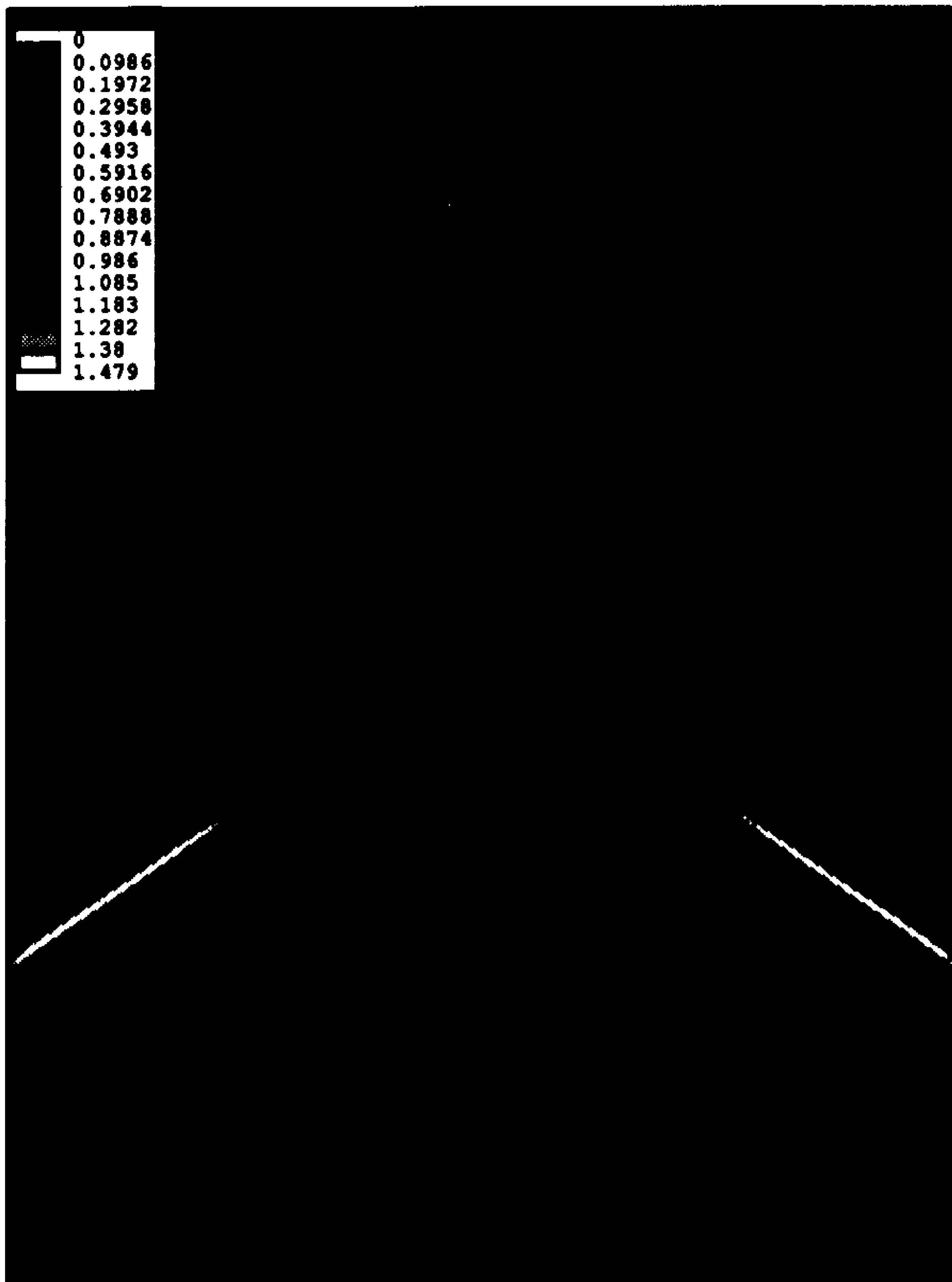
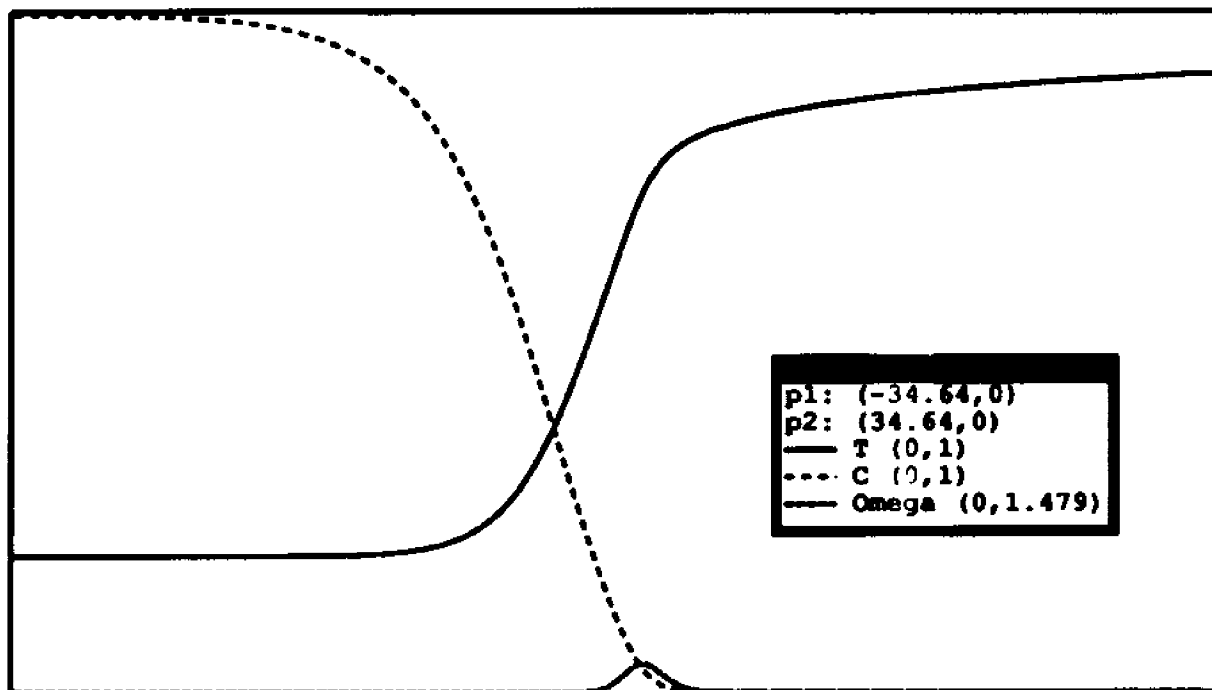


Figure 8.17: Reaction rate distribution for the slot Bunsen burner ($\nu = 0$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$).



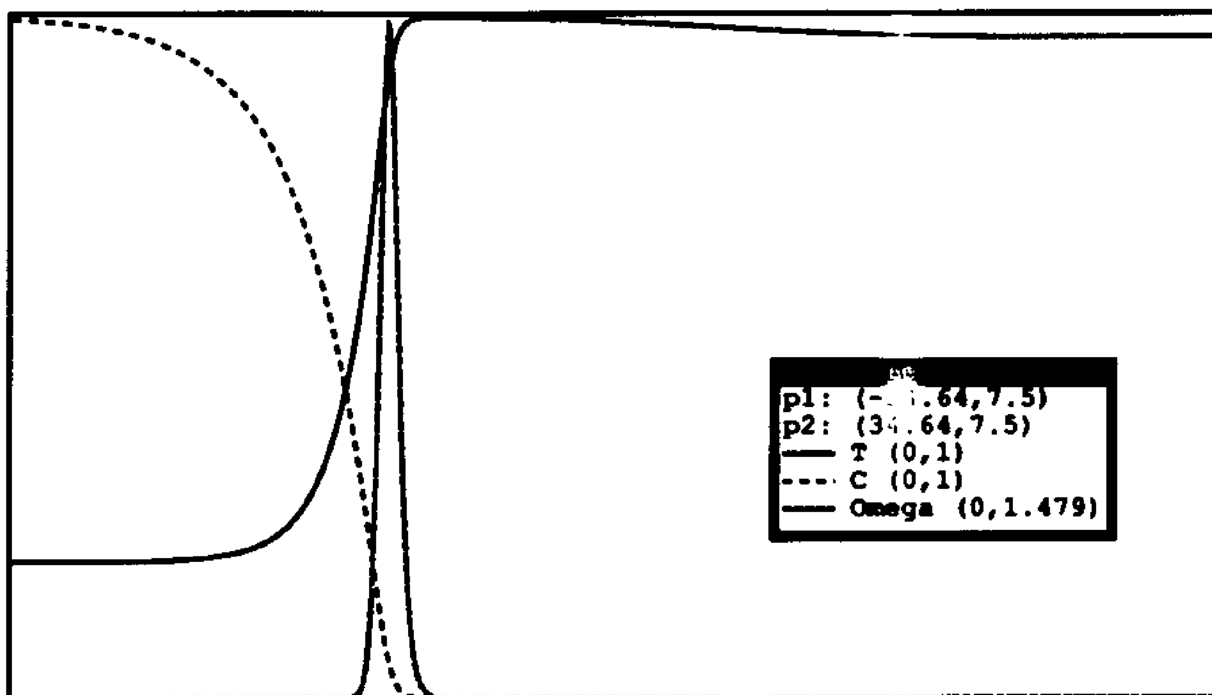


Figure 8.18: Temperature (T), concentration (C), and reaction rate (Ω) profiles for the slot Bunsen burner at $r = 0$ (a) and $r = 7.5$ (b) ($\nu = 0$, $\alpha = \pi/6$, $Le = 0.5$, $-10 \leq r \leq 10$, $-34.6 \leq z \leq 34.6$). T and C vary within the interval $(0, 1)$, while Ω varies within $(0, 1.479)$.

8.2 Flame propagation in shear flow

This problem deals with the premixed flame propagating through a unidirectional periodic flow field and is intended to simulate certain basic features of turbulent combustion. It is shown that the chemical reaction rate may undergo considerable variation along the front, causing the flame to split up into separate flamelets. As a result, depending on the thermal-diffusive properties of the mixture, the flame assumes a periodic configuration resembling that of a usual (or inverted) cellular flame.

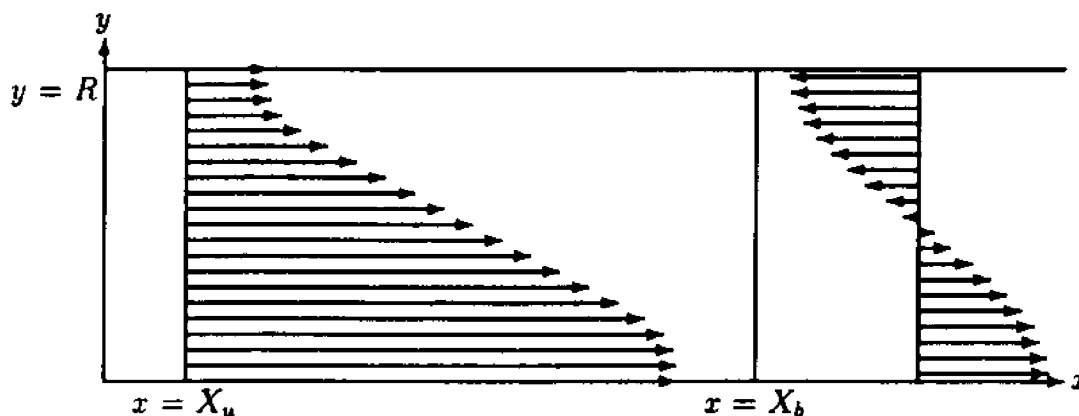


Figure 8.19: Flame propagation in shear flow configuration.

8.2.1 Problem formulation

We consider the propagation of a premixed flame in a two-dimensional infinite channel with a given field of velocities (see Figure 8.19). A single one-step chemical reaction between fuel and oxidizer is assumed. To simplify the computations we will consider a model with constant density.

Let us introduce the notations: x, y — the cartesian coordinates on a 2-dimensional plane; R — the width of the tunnel; X_u — the x coordinate of the moving

boundary of the computational domain, unburned side; X_b — the x coordinate of the moving boundary of the computational domain, burned side ($X_b > X_u$); T — normalized temperature of the gas mixture ($0 \leq T$); C — normalized concentration of a reactant (the other reactant being in excess) ($0 \leq C \leq 1$); $v_x(x,y,t)$, $v_y(x,y,t)$ — the components of a given velocity field of the gas mixture; Ω — the reaction term; β — the reduced activation energy; Le — the Lewis number; γ — the heat release parameter.

The model is described by the following equations in normalized variables

$$\frac{\partial T}{\partial t} + v_x\frac{\partial T}{\partial x} + v_y\frac{\partial T}{\partial y} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \Omega \quad (8.20)$$

$$\frac{\partial C}{\partial t} + v_x\frac{\partial C}{\partial x} + v_y\frac{\partial C}{\partial y} = \frac{1}{Le}(\frac{\partial^2 C}{\partial x^2} + \frac{\partial^2 C}{\partial y^2}) - \Omega \quad (8.21)$$

in the portion of the tunnel $(X_u, X_b) \times (0, R)$ moving with the flame front.

The reaction term Ω is given by the formula

$$\Omega = \frac{\beta^2}{2Le} C \exp^{\frac{\beta(T-1)}{1+\gamma(T-1)}} \quad (8.22)$$

The boundary conditions are

$$\begin{aligned} \tfrac{\partial T}{\partial y} &= 0, & \tfrac{\partial C}{\partial y} &= 0 & \text{at } y &= 0, R, \\ T &= 0, & C &= 1 & \text{at } x &= X_u, \\ \tfrac{\partial T}{\partial x} &= 0, & \tfrac{\partial C}{\partial x} &= 0 & \text{at } x &= X_b. \end{aligned} \quad (8.23)$$

One of the interesting cases is when the velocity field is specified by the formula

$$v_x(x,y,t) = A\cos\left(\frac{2\pi y}{R}\right), v_y(x,y,t) = 0. \tag{8.24}$$

The special property of this problem is that the flame front is moving and, at the same time, changing its shape. We have to move the computational domain in order to keep the flame front inside it. The average flame front velocity, not being known in advance, has to be computed numerically.

In case $A = 0$, an asymptotic solution exists

$$v = 1 - \frac{1}{\beta}((Le - 1) + 3(1 - \sigma) - 1.344) \tag{8.25}$$

where $\sigma = 0.2$ when $\beta = 10$.

8.2.2 Flame front velocity

The discretization of the equations (8.20–8.21) is almost identical to the case of the Bunsen burner. The only substantial new feature in this model is the computation of the flame front velocity. It has to be done for two reasons: it is one of the main parameters of the flame we want to compute, and it is needed to have the computational domain move so that the flame front is always inside it.

Because of symmetry considerations we know that the flame front moves only in the x direction. At intermediate stages of the computation, the flame front not only moves in one direction, but changes its shape as well. Therefore, some kind of average

velocity must be defined. The flame front velocity correction is computed through the x-momentum of the temperature change by the following formula

$$\Delta v = \frac{\int x(T(x,y,t_0+\Delta t) - T(x,y,t_0))}{\Delta t \int T(x,y,t_0)} \tag{8.26}$$

where the integral is taken over the computational domain. In order for the averaging to be meaningful, the function values must be on one side of zero. In general, the formula is valid only when the function support stays inside the computational domain during the interval of time used to compute the velocity correction. Although the support of the temperature does not have this property, the resulting error is not essential, and the formula (8.26) has been successfully used in practice.

8.2.3 Numerical experiments

Numerical experiments were conducted with the goal of determining the dependence of the flame front velocity on the amplitude of the velocity field, the wavelength, and the Lewis number. These experiments will be described elsewhere. As an example, we shall present only the solution for the case of $Le = 1.5$, $A = 1$, and $R = 10$. The computational domain length was taken to be 32. The solution is illustrated in Figures 8.20–8.22. The first of these pictures shows the final adaptive grid superimposed over the contours of the temperature. Note, that on all the pictures the origin of the system of coordinates is at the lower-right corner and the x-axis is vertical.

Experiments were conducted for the above case in order to determine the influence of the grid resolution. Solutions were computed on the sequence of grids $\mathcal{G}_0$, $\mathcal{G}_1$,

and $\mathcal{G}_2$, in which every grid has two times finer resolution than the previous grid. The flame front velocities were 0.753, 0.726, and 0.712, respectively. Physically, this result is quite understandable, given that numerical diffusion decreases as grid resolution increases. The smaller the the overall diffusion, the smaller the flame front velocity. The resulting flame front velocity (0.71) is smaller than the flame front velocity numerically computed in absence of the imposed shear flow (0.90). The velocity computed by the asymptotic formula (8.25) for the latter case (0.84) is slightly lower than numerically computed velocity.

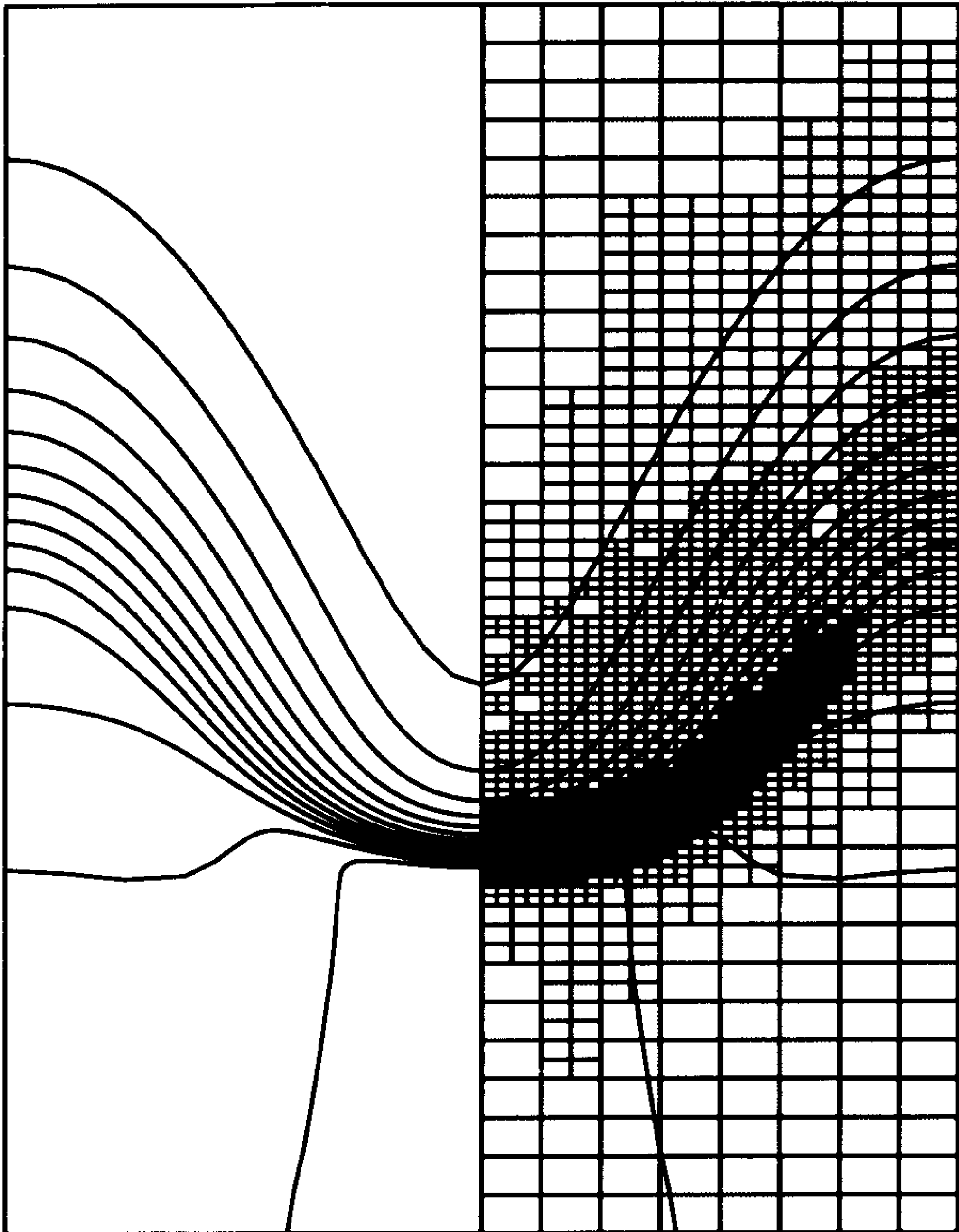


Figure 8.20: Contours of temperature distribution for the shear flow (Le = 1.5, $0 \leq x \leq 32$, $0 \leq y \leq 20$) with superimposed adaptive grid.

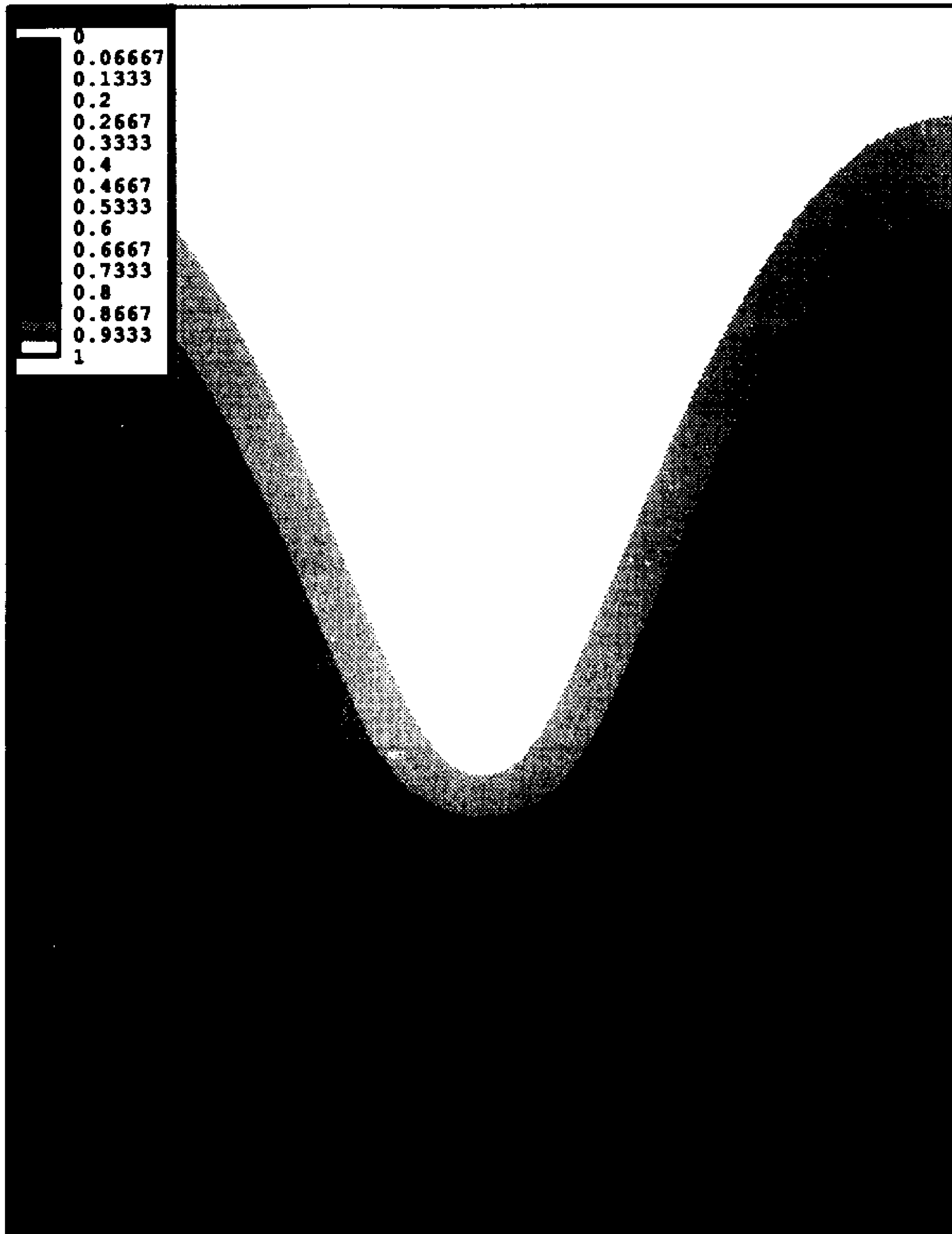


Figure 8.21: Concentration distribution for the shear flow (Le = 1.5, $0 \leq x \leq 32$, $0 \leq y \leq 20$).

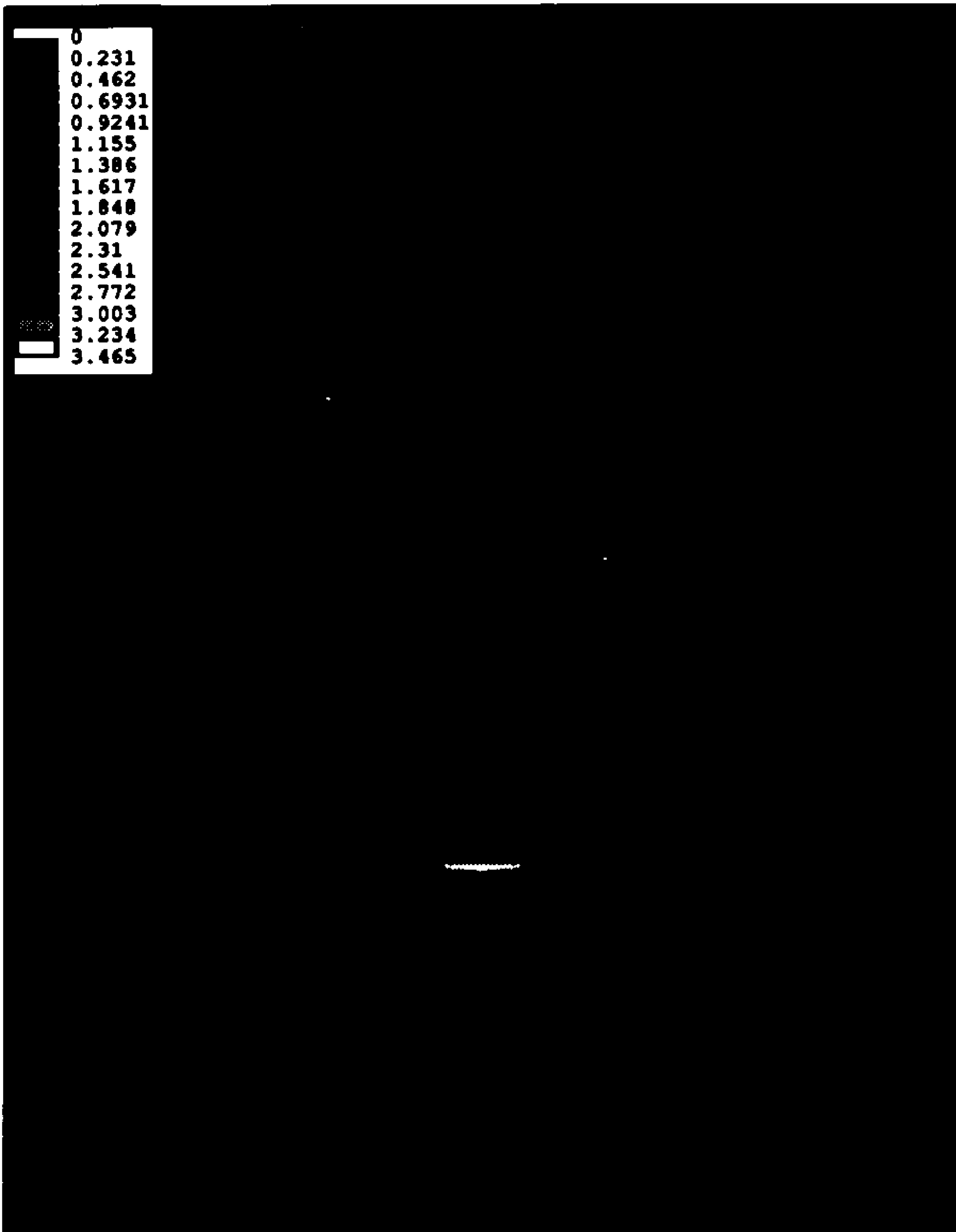


Figure 8.22: Reaction rate distribution for the shear flow (Le = 1.5, $0 \leq x \leq 32$, $0 \leq y \leq 20$).

Appendix A

Implementation example: Bunsen burner model

The implementation of a particular solver using the interactive programming environment described above consists of a set of compiled C modules and a declaration and initialization script in the interpreter language. The programming style we use is dictated by considerations of readability and adaptability. All the parameters and variables, that specify the given problem in a class of problems, that are likely to be changed, or that have to be interactively examined are declared in the interpreter.

We describe the implementation of the solution of the Bunsen burner model (Section 8.1) using finite elements spatial discretization and explicit time steps (Chapter 3). The size of this implementation is about 1400 lines (including comments) of compiled C modules plus about 60 lines of interpreter script. This is small compared to the size of the interactive programming environment (which consists of about 18000 lines of C code). Our guess estimate is that a stand-alone implementation with comparable functionality would have five times more code.

A.1 Implementation notes

Numerical modules have local (static or external) variables which require proper initialization. Some of them must be assigned the values of parameters defined in the interpreter data space, and some are pointers to be assigned addresses of data in the interpreter data space. Normally, every numerical function callable from the interpreter will re-initialize these variables to account for possible changes in the interpreter data space made interactively between function calls.

In order to treat boundary conditions, each grid node is assigned a node boundary type which is different for different boundary conditions and for special boundary nodes, such as corners. The codes of boundary types are kept in the vector of bytes `ntype`.

The time step is performed as following. First, the chemical reaction term is computed for the terminal grid nodes and placed in vector `Omega`. This saves a significant amount of work compared to computing values of `Omega` as needed during the subsequent computations. Second, changes in temperature and concentration are computed and placed in vectors `dT` and `dC`. Third, the vectors `T` and `C` are updated. Negative values of C, obtained during the update, are set to zero.

A.2 Compiled C modules

`void nm_init(void)`

Initializes local variables in numerical modules. Reads values of some inter-

preter variables and initialize pointers to others. Precomputes frequently used expressions.

`void nm_run(int nit)`

Performs `nit` iterations consisting of a fixed amount of explicit time steps (in our experiments 20 steps) followed by grid adaptation. During the iterations, the changing grid is displayed on the screen. A pop-up window with information about the solution process appears during the time step phase.

`void nm_tstep(int nstep)`

Performs `nstep` explicit time steps.

`void nm_err(void)`

Computes and prints the numerical discretization of the right hand side of the equations (8.1–8.2). The error size indicates convergence to a steady state.

`void nm_omega(void)`

Computes vector `Omega` (chemical reaction term) for a given temperature `T` and concentration `C`.

`index nm_adapt(void)`

Adapts the grid to the current solution. Returns the number of supercells after adaptation.

`void nm_rootinit(void)`

Performs initialization once for a new root grid, including preassembling the

stiffness matrix coefficients for the new grid.

`void nm_rootend(void)`

Frees memory allocated by nm_rootinit when the grid is destroyed.

`void initbnd(void)`

Initializes node boundary type vector **ntype**.

`void refbnd(void)`

Refinement rule for the node boundary type vector **ntype**.

`float tinit(float z,float r)`

`float cinit(float z,float r)`

Computes an initial estimate for temperature and concentration at point `(r,z)`.

`void refT(void)`

`void refC(void)`

Refinement rules for temperature vector `T` and for concentration vector `C`. A linear refinement rule is used for the internal nodes and for the boundary nodes with Neumann boundary condition; boundary condition right hand side is used for the boundary nodes with Dirichle boundary condition.

`void nm_d2Td2Cscan(void)`

A space function which computes and displays numerical second derivatives at the node pointed to by the cursor in scan mode. Used for debugging and examining the solution.

A.3 Interpreter script

The following script is used to declare interpreter variables, define the parameters, and perform initialization for the Bunsen burner numerical model. After executing the script, the interpreter enters the interactive mode. The solution can be obtained by calling the function `nm_init`. It can be examined using various available tools, and/or saved for later use.

```
############### init file for alpha = Pi/4 ##############
int NODEVEC = 8    # code for node vector
int CELLVEC = 16   # code for cell vector

############## geometric parameters ###########
float PI = 3.1415926535
float alpha = PI/4 # characteristic angle
float R1 = 10.0    # radius of the pipe
float Z1 = - R1/tan(alpha)
float Z0 = 2.0*Z1  # inlet boundary
float Z2 = -Z0     # outlet boundary

dclgrid(4100,1000,nm_rootinit,nm_rootend) # declare grid space
rgcreate(&{Z0,Z2,0.,R1},"z","r",20,5) # create root grid
wwin = domain      # world window

############## combustion parameters ##########
float N = 10.0    #  dimensionless activation energy
float Le = 0.5     #  Lewis number
float sigma = 0.2  #  thermal expansion
float A = (1.-sigma)*(1.-sigma)/(2.*Le) # reduced pre-exp factor
float vel = 1.0/sin(alpha)              # velocity of the mixture

############# declare grid vectors ############
byte ntype[maxnodes] # boundary type of nodes
go_dcl(ntype,initbnd,refbnd,NULL)  # declare as grid object
gv_dcl(ntype,NODEVEC,&isopos,defpalette) # declare as grid vector

float T[maxnodes] # temperature
go_dcl(T,NULL,refT,unreflin)
```

```
gv_dcl(T,NODEVEC,&isopos,defpalette)

float dT[maxnodes]      # T change
gv_dcl(dT,NODEVEC,&isoindef,indefpal)

float C[maxnodes] # concentration
go_dcl(C,NULL,refC,unreflin)
gv_dcl(C,NODEVEC,&isopos,defpalette)

float dC[maxnodes]      # C change
gv_dcl(dC,NODEVEC,&isoindef,indefpal)

byte rfind[4*maxscells] # refinement indicator
gv_dcl(rfind,CELLVEC,&isopos,defpalette)

float Omega[maxnodes] # reaction term
gv_dcl(Omega,NODEVEC,&isopos,defpalette)

############### ordering module ################
sd_defgn()             # index lists of nodes

################ numerical method ##############
float tstep            # time step
int nit = 0            # number of iterations performed
float phystime = 0     # physical time passed from the start
##### refinement/unrefinement parameters
int maxreflevel = 2 # restriction of refinement level
float rtol = 0.02      # if (refcrit > rtol) refine
float utol = 0.004     # if (refcrit < utol) unrefine

############ declare space functions ###########
sf_dcl(nm_d2Td2Cscan)  # compute second derivatives

######### compute initial approximation ########
nm_init()              # initialize numerical module
ndvset(T,tinit)        # initial guess for T
ndvset(C,cinit)        # initial guess for C
```

Bibliography

Abdel-Gayed, R. G., and Bradley, D. (1985). *Criteria for turbulent propagation limits of premixed flames,* Combust. Flame 62, 61–68.

Abdel-Gayed, R. G., and Bradley, D. (1989). *Combustion regimes and the straining of turbulent premixed flames,* Combust. Flame 76, 213–218.

Adobe Systems Incorporated. (1985). *Postscript Language Tutorial and Cookbook.* Addison-Wesley, Reading, Mass.

Adobe Systems Incorporated. (1990). *Postscript Language Reference Manual.* Addison-Wesley, Reading, Mass.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass.

Benkhaldoun, F., Leyland, P., and Larrouturou, B. (1988). *Dynamic mesh adaption for unsteady nonlinear phenomena — application to flame propagation,* Proceedings of the Second International Conference on Numerical Grid Generation in Computational Fluid Dynamics, Miami Beach.

Berestycki, H., and Sivashinsky, G. I. (1991). *Flame extinction by periodic flow field.* SIAM J. Appl. Math. 51, 344–350.

Berger, M. J., and Oliger, J. (1984). *Adaptive mesh refinement for hyperbolic partial differential equations,* J. Comput. Phys. 53, 484–512.

Buckmaster, J. D. (1979). *A mathematical description of open and closed flame tips,* Combust. Sci. Tech. 20, 33.

Buckmaster, J. D. (1979). *The quenching of a deflagration wave held in front of a bluff body,* Seventeenth Symposium (International) on Combustion, The Combustion Institute, Pittsburgh, Pa., 835–842.

Buckmaster, J. D., and Crowley, A. B. (1983). *The fluid mechanics of flame tips,* J. Fluid Mech. 131, 341–361.

Chomiak, J., and Jarosinsky, J. (1982). *Flame quenching by turbulence,* Combust. Flame 48, 241–249.

Conte, S. D., and de Boor, C. (1980). *Elementary Numerical Analysis,* McGraw-Hill, New York.

Dannenhofer, J. F., and Baron, J. R. (1986). *Robust grid adaptation for complex transonic flows,* AIAA 24th Aerospace Sciences Meeting Proceedings, Paper AIAA-86-0495, Reno, Nevada, January 6–9, 1986.

Denet, B. (1989). *Thése de l'Université de Provence,* Marseille, France.

Denet, B., and Handelwang, P. (1990). *A pseudo-spectral scheme for turbulent thermo-diffusive premixed flames,* Preprint.

Ewing R. E. (1986). *Adaptive mesh refinements in large-scale fluid flow simulation,* in Babuska, I. et al. (Eds.), Accuracy Estimates and Adaptive Refinements in Finite Element Computations, Wiley, New York, 299-314.

Fernandez, G., Larrouturou, B., and Sivashinsky, G. I. (1989). *Numerical Combustion,* A. Derrieux and B. Larrouturou (Eds). Lecture Notes in Physics, 351, 287–298.

Frankel, M. L., and Sivashinsky, G. I. (1984). *On quenching of curved flames,* Combust. Sci. Tech. 40, 257–268.

Hyman, J. M., and Larrouturou, B. (1986) *On the Use of Adaptive Moving Grid Methods in Combustion Problems.* Proceedings of the Second Workshop on Modelling of Chemical Reaction Systems, Heidelberg.

Kernighan, B. W., and Ritchie, D. M. (1988). *The C Programming Language.* Prentice Hall, Englewood Cliffs.

Law, C. K., Ishizuka, S., and Mizomoto, M. (1981). *Lean-limit extinction of propane/air mixtures in the stagnation point flow,* Eighteenth Symposium (International) on Combustion, Pittsburgh, Pa., 1791–1798.

Law, C. K., Ishizuka, S., and Cho, P. (1982). *On the opening of premixed Bunsen flame tips,* Combust. Sci. Tech. 28, 89–96.

Lewis, B., and von Elbe, G. (1961). *Combustion, Flames, and Explosion of Gases,* Academic Press, New York, 2nd Edition.

McCormick, S. F. (1989) *Multilevel Adaptive Methods for Partial Differential Equations.* SIAM, Philadelphia.

Mizomoto, M. and Yoshida, H. (1987). *Effects of Lewis number on the burning intensity of Bunsen flames,* Combust. Flame 70, 47–60.

Rheinboldt, W. C., and Mesztenyi, C. K. (1980). *On a data structure for adaptive finite element mesh refinements,* ACM Trans. on Math. Software 6, No. 2, 166–187.

Sato, J. and Tsuji, H. (1978). *Behavior and extinction of a premixed flame in a stagnation flow for general Lewis numbers,* Proc. of the Sixteenth Japanese Symposium on Combustion, 13–15.

Sato, J. and Tsuji, H. (1983). *Extinction of premixed flames in a stagnation flow considering general Lewis numbers,* Combust. Sci. Tech. 33, 193–205.

Sivashinsky, G. I. (1975). *Structure of Bunsen flames.* J. Chem. Phys. 62, 638–643.

Sivashinsky, G. I. (1976). *On a distorted flame front as a hydrodynamic discontinuity,* Acta Astronautica, 3, 889–918.

Sokolik, A. S., Karpov, W. P., and Semenov, E. S. (1967). *Turbulent combustion of gases,* Combust. Expl. Shock Waves 3, 36–45.

Strang, G. and Fix, G. J. (1973). *An Analysis of the Finite Element Method,* Prentice-Hall, Englewood Cliffs.

Zienkiewicz O. C. (1977). *The Finite Element Method,* McGraw-Hill, London.