One-Dimensional Numerical Model for Evaporation and Oxidation of Hydrocarbon Fuels

by

Ivan B. Oliveira

Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Mechanical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

Author .....
Department of Mechanical Engineering
May 7, 1999

Certified by .....
Simone Hochgreb
Associate Professor
Thesis Supervisor

Accepted by .....
Ain A. Sonin
Chairman, Department Committee on Graduate Students

One-Dimensional Numerical Model for Evaporation and Oxidation of Hydrocarbon Fuels

by

Ivan B. Oliveira

Submitted to the Department of Mechanical Engineering
on May 7, 1999, in partial fulfillment of the
requirements for the degree of
Master of Science in Mechanical Engineering

Abstract

In this work, a detailed chemistry, one-dimensional, reactive-diffusive model is implemented to study the basic aspects of evaporation and oxidation of a thin liquid fuel layer exposed to an incoming premixed flame. In particular, the model is applied to predict the total evaporation and ensuing oxidation of a liquid layer under repeated cycles. Methanol was used as a baseline fuel. Simplifications in the flow, geometry, and operating conditions are made to restrict the problem to its fundamental mechanisms. The solution method solves the appropriate governing equations in the liquid and gas phases, observing mass and species conservation with phase-equilibrium at the interface. The resulting eigenvalue problem is solved for pure liquid layers, but the extension of multi-component liquids is possible. Results show that increasing pressures lead to relatively lean regions near the interface due to the inverse dependence of phase-equilibrium concentrations on pressure. As a premixed flame arrives at the interface, large temperature gradients evaporate fuel from the layer as the remaining oxygen diffuses back into core gases. A short-lived diffusion flame results, which greatly enhances the rate of evaporation, serving as both a source of energy and a sink of fuel. Similar results are observed for pressure histories that resemble those of operating spark-ignition engines. Decreasing liquid layer thicknesses, increasing wall temperatures, and decreasing heats of vaporization are all observed to enhance the rate of evaporation mainly due to their impact on the heat transfer characteristics of the problem. Since the liquid layer surface is restricted to temperatures below or equal to the liquid boiling point, however, boundary layer temperatures for all cases are very similar, and thus total survival rate of evaporated fuel, repeatedly found to be roughly 2.9% for methanol, is quite insensitive to these parameters.

Thesis Supervisor: Simone Hochgreb
Title: Associate Professor

to my family

Acknowledgments

I would like to begin by thanking my advisor, Simone Hochgreb, for providing me with the opportunity to work on a very interesting project. Her guidance and encouragement have often made this work less difficult and more rewarding. I would also like to acknowledge the Sloan Automotive Lab, MIT, the EPA, and the Engine & Fuels Consortium, all of which made this project possible.

I am very grateful for the help I've receive at the Lab, from faculty and students. The help of Kuo-Chun Wu was key in beginning the project on the right note and his original work laid the foundation from which much of the code was built. Chris O' Brian deserves special mention for his patience and availability. His advice and input were most helpful, both in my understanding of the problem and in the implementation of ideas.

I would also like to thank those who were not officially involved in the project but who helped make these years enjoyable. First is Deanna, for her presence. I also thank my friends for providing an often-welcome distraction; from UVA to the Brazilians at MIT.

Foremost, I would like to thank my family, to whom I respectfully dedicate this thesis. Their encouragement and support have played a larger role in all my work than any other factor. In particular, I thank my mother for her love and encouragement, my father for his support and guidance, and both for the many sacrifices they have made. In my mind they truly define the ideals of parenting, and I will always admire their wisdom and experience. I thank my sisters, for their camaraderie, affection, and good humor. Growing up with them has been a priceless experience. Laikinha is dearly thanked, and will always be remembered with affection for her loyalty.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Hydrocarbon Emissions

Much of the research addressing hydrocarbon emissions from combustion devices [1] has relied heavily on experimental work. This is mainly due to inherent difficulties in performing calculations, which arise from the variety of physical processes, complex chemical mechanisms, stiff governing equations, and complex geometries involved. As a result, however, limitations in measuring detailed chemical compositions and heat transfer characteristics in the small crevices or liquid-layers from which hydrocarbons may emerge have created a need to explore alternatives to this approach. This work aims at constructing and implementing a numerical model to allow for the detailed understanding of the fundamental processes governing evaporation and oxidation of liquid fuel layers that are often found in combustion devices.

Recent legislation has successively restricted acceptable levels of hydrocarbon emissions from combustion devices, a trend which is likely to continue in the future. It is therefore imperative that detail models be constructed to accurately describe the various sources of hydrocarbon emissions and the often subtle impact of operating conditions that cannot be obtained from experimental data. Sources of hydrocarbon emissions that have been identified [1] include oil layers, cylinder crevices, and liquid fuel layer in the combustion chamber. This work is concerned with addressing aspects of the latter of these processes.

A serious source of hydrocarbon emissions is "cold start" operation, characterized by flooding of the engine with liquid fuel onto cold walls during engine warm-up. Although rough calculations predict that the total amount of surviving fuel for such a situation is substantial, they lack in accuracy and cannot predict the effect of various operating conditions. A typical approach in modeling such processes has been to use simplified chemistry or simplified evaporation models. The simplifications required for such an approach, which often include steady-state assumptions, hamper the accuracy of the results since the time scales of heat transfer modes in the process are actually of the order of changing conditions in combustion devices. With the constant improvements in computational power, it has become increasingly possible to combine various mechanisms in the process, which should allow for detailed calculations and consequently better accuracy.

Such calculations can quickly become prohibitive if all aspects of engine operation are attempted, however. In this work, simplified geometric and flow assumptions have been made in order to focus on the heat transfer and chemical time scales of the process. A single baseline fuel has been used (methanol) in order to determine the sensitivity to the different parameters involved. Although the results presented address key aspects of the process, the model can be used for a variety of other fuels and operating conditions. Since the large majority of hydrocarbons that survive from a liquid layer are in the form of the actual fuel species, only total fuel emission has been addressed in this work, with an emphasis on the impact of the various parameters on rates of evaporation. It is possible, however, to address survival rates of any arbitrary species (provided reliable chemical-kinetic mechanisms are available) since the model solves for detailed chemistry.

1.2 Summary of Past Work

Previous detailed numerical work has been done in modeling oxidation of unburned combustion gases from crevices [2, 3] in conditions similar to those found in combustion engines. Although successful in explaining many of the controlling oxidation

mechanisms for such configurations, these models describe only dry wall phenomena, which are thermally quite different from liquid layer evaporation and oxidation. Therefore, the effect of operating conditions and mixture properties, as well as emissions characteristics, are very different in both cases, which demands the construction a new model.

Work regarding the solution of gas-phase conservation equations has been extensively explored in the combustion field. This project initially adopted the code available from Wu [3], which solved the equations for one-dimensional, impermeable wall oxidation. The present version of the code, however, shows no resemblance to the original since limitations required a complete restructuring in order to allow for flexibility and expansion. The original code relied on the NAG libraries [4] for numerical integration. A proprietary collection of software, the libraries do now allow access to the source code, which prevented modifications that were required for the adaptation of the remeshing procedure and other optimization issues. LSODES [5], a freely distributed software, substituted the NAG libraries for time integration and was found to give superior results. This modification required the implementation of the method of lines and the resulting restructuring of the code.

Modeling aimed at describing evaporation and oxidation of liquids has also been extensively explored in the combustion field (although to a lesser extent than gas-phase-only modeling). The majority of this modeling has relied on several assumptions to simplify the solution of the problem. Detailed calculations have been made in some instances [6], however, and observations have been made for constant pressure combustion of liquid droplets. Since the geometry and conditions are different from those of interest here, the present model was constructed.

1.3 Model Overview

The problem in question consists of a thin liquid layer on a relatively cool, impermeable wall. The assumed geometry is one-dimensional (cartesian or spherical). Under circumstances of interest here, a premixed flame originating from large x initially

moves toward the liquid layer and eventually provides the heat required for evaporation. General description of the geometry and a schematic representation of the problem are provided in chapter 2.

The dominating physical process is the requirement that at the liquid-gas interface phase-equilibrium be maintained. This condition has been generally imposed in past calculations [7] due to the much smaller time scales associated with reaching phase-equilibrium at the interface with respect to other possible time scales of the problem. In addition, conservation of mass, species, and energy is applied to the entire domain (liquid, interface, and gas). The system results in a set of partial differential equations with an eigenvalue which represents some measure of mass flux through the interface (rate of evaporation or condensation). Chapters 3 presents the equations that describe the problem and a derivation of the required boundary conditions.

An adaptive meshing procedure [8] has been implemented and the method of lines is used to reduce the partial differential equations into a set of ordinary differential equations. Chapter 4 addresses the solution method and some of the issues related to the numerical time integration of the problem. The software package LSODES [5] is used for this purpose with some slight modifications in order to incorporate an effective strategy for adaptive meshing.

1.4 Objectives

The objectives of this work are as follows:

- To develop and implement a successful and efficient computer model for the detailed calculation of evaporation and oxidation of liquid fuel layers or droplets. Include conservation at the liquid-gas interface and full chemical kinetic mechanisms.
- To verify the model with analytical solutions and existing experimental data.
- To use the model to describe controlling fundamental mechanisms involved in the evaporation and oxidation rates of fuel liquid layers for a baseline fuel.

- To observe the different effects of constant pressure versus engine operating conditions and derive qualitative and quantitative conclusions from the mechanisms that aid or hamper evaporation and total fuel oxidation.

Chapters 5 and 6 preset the verification and results mentioned above. It should be noted that the present version of the code was structured so that additional expansion to allow for evaporation and condensation of different species (multi-component liquids) is possible. This is the natural continuation of this work and would improved accuracy under circumstances where such a process plays an important role. It should also be noted that the list above is not extensive regarding possible outputs from the code, which allow for the analysis of emissions of other species and heat transfer processes at different locations of the domain.

Chapter 2

Problem Definition

2.1 Geometric Configuration

In order to determine the effect of oxidation on the evaporation of a fuel liquid layer, a simple geometric configuration has been adopted. A one-dimensional flame approaches a cold, impermeable wall that is covered with a fuel liquid layer of a given initial thickness. The geometry can be either cartesian, cylindrical, or spherical (in which case a wall may not exist). The flat geometry is of particular interest in this work since it closely resembles the surface of an engine wall in the context of the quench distance of an approaching flame.

2.1.1 Problem Schematic

Figure (2-1) is a schematic representation of typical initial conditions regarding the thermal environment of the problem. A flame is shown approaching from the left into a cold liquid layer. In this example, the wall temperature, T_w, is higher than the fuel boiling point temperature at the prevalent pressure, T_b (which is likely for fuels with low T_b such as methanol). The liquid surface temperature does not increase above T_b, resulting in a heat flux q_i'' due to gas temperature gradients near the surface. Evaporation at the surface requires heat transfer from the liquid layer of $q_o'' = \dot{m}_e L_t$, where $\dot{m}_e$ and L_t are the evaporation mass flux and heat of vaporization, respectively.

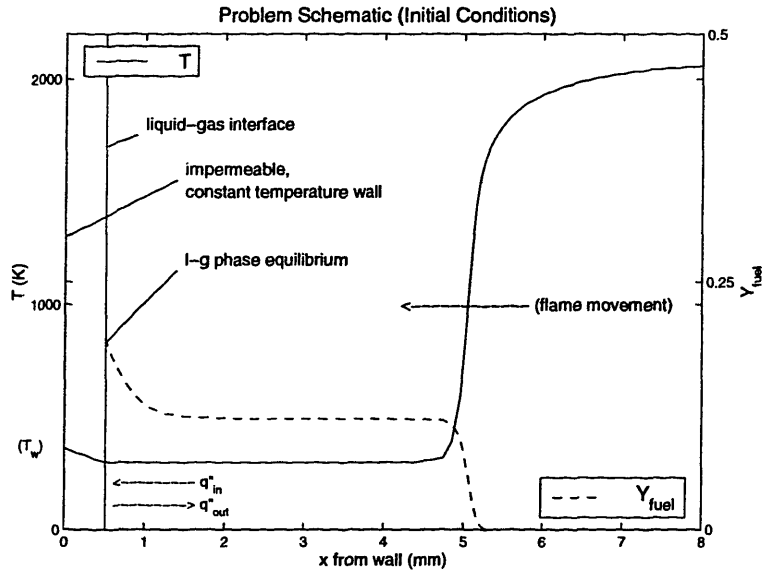


Figure 2-1: Schematic of the thermal and fuel species environments of the problem

Figure (2-1) also shows the initial fuel species configuration for this problem. At the liquid-gas interface, the species can be assumed to be at the liquid-gas phase equilibrium concentration, which is a function of interface temperature and pressure. If, for example, the interface concentration is higher than the adjacent gas fuel concentration, evaporation must occur (positive $\dot{m}_e$) to compensate for diffusion and maintain phase equilibrium. If the opposite is true, condensation must occur to maintain phase equilibrium (negative $\dot{m}_e$).

The coupling between species and heat transfer at the liquid-gas interface arises due to the strong dependence of gas phase-equilibrium concentrations on the local temperature, and consequently, the dependence of the local temperature on the energy transfer from (or to) the gas phase. This coupling gives rise to an eigenvalue, which here is represented by the mass flux through the interface (see chapters 3 and 4).

As shown in the Figure (2-1), most of the problems considered in this work have a left-most boundary consisting of an impermeable, smooth wall covered by an incompressible liquid layer. The wall-liquid interface is maintained at a given temperature for the entire calculation cycle. The right-most boundary consists of a uniform concentration of gases at infinity where all gradients in temperature and species profiles approach zero. These imposed boundary conditions may change depending on the

type of problem in question. The version of the code used for the present calculations allows for easy implementation of other boundary conditions representing liquid droplets, fixed heat fluxes at the wall, or any other Neumann or Dirichlet conditions that represent a particular problem. Details on the implementation of these boundary terms are presented in chapters 3 and 4.

The behavior of the liquid layer or droplet under a variety of conditions over a period of time is of interest in this work. Initial conditions vary with the type of problem, but typically consist of the specification of species and temperature profiles in the liquid layer and in the gas phase at a particular point in time. Often, this will consist of a flame front travelling toward the wall, starting from a particular location. The pressure history for all calculations (whether constant or varying in time) is also specified.

2.1.2 Approximations

In order to maintain a tractable degree of complexity, certain simplification are made in the model.

One-Dimensional Geometry

The one-dimensional nature of the problem implies that the thickness of liquid layers in question, x_{lo}, is much smaller than its surface area. This assumption is made on the basis that the layers considered here are of small x_{lo} (typically less than 500 μm). The approximation also implies that the flow field in the gas phase is neglected in the problem. Since the cases studied here are very close to the wall, viscous forces may be said to dominate the process, eliminating strong convective flow in the problem domain.

Gas-Phase Laminar Transport

Although a one-dimensional turbulence model can easily be incorporated into the gas-phase conservation equations [3, p. 999], this project is concerned with determining

the fundamental evaporation characteristics of a liquid layer exposed to a flame. Thus, the additional complexity of turbulent intensity was avoided.

Constant Temperature Wall

The wall was considered to remain at a constant temperature throughout the calculations. This is an idealization of the situation that may be encountered in combustion devices, which may exhibit increasing wall temperatures with operation time. By specifying appropriate conditions, however, the model can easily take arbitrary wall temperature changes into account. This was avoided in the present work in order to again eliminate extraneous complexities.

Other Simplifications

The wall is considered to be perfectly smooth. Therefore, it is impossible for nucleation sites to exist and boiling in the liquid phase will not occur. Furthermore, motion induced by density gradients in the liquid layer are assumed to have a negligible effect on heat transfer. This assumption is made on the basis that, for very thin liquid layers, viscous damping prevents significant convection compared to conduction time scales.

2.2 Pressure Dependence

In addition to being sensitive functions of temperature, phase-equilibrium concentrations at the interface are also functions of pressure. This occurs due to the fact that the vapor pressure for a given temperature is independent of pressure, but as the pressure changes, the partial pressure of the evaporating species changes accordingly. For the same vapor pressure, increases in the total pressure decrease the species partial pressure and thus reduce the driving potential for evaporation. In order to determine the magnitude of this dependence in comparison to other mechanisms during evaporation and oxidation, calculations were performed at different constant pressures.

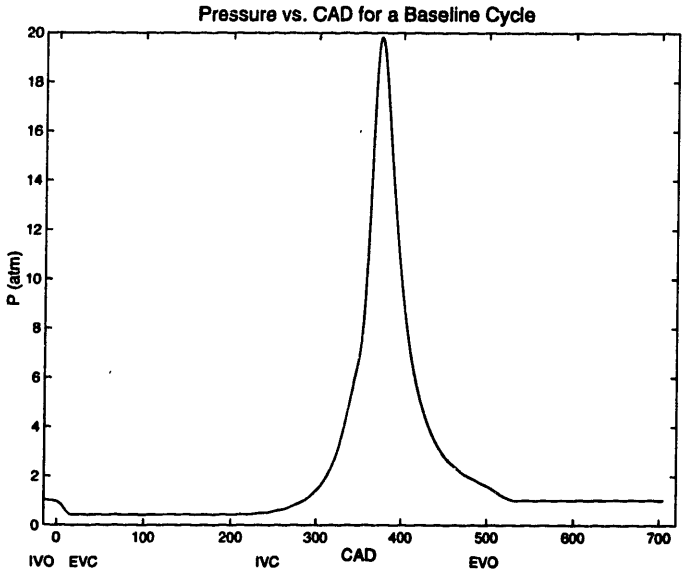


Figure 2-2: Pressure history during one cycle for baseline case

2.3 Engine Environment Calculations

This work is mainly concerned with evaporation processes that occur in environments that resemble those of internal combustion engines. To simulate such environments, the model accepts an input of varying pressure during the calculation. The pressure history can be obtained experimentally or from engine calculations. In the present model, it was calculated by the zero-dimensional engine simulation of Poulos and Heywood [9]. A typical profile of the pressure history for a given cycle is shown in Figure (2-2) (see section 6.3 for engine specifications and details).

For each cycle, evaporation is driven by an incoming flame that reaches the liquid layer and provides a large amount of heat. This process occurs roughly at the point in Figure (2-2) where the pressure reaches its maximum for the cycle. For identical cycles, the rate of change of liquid layer thickness is only a function of initial thickness (due to differences in heat transfer). Given a set of calculated values of the rate of change per cycle of liquid layer thickness, it is possible to calculate the total number of cycles (and hence the time) required for complete evaporation.

Operating conditions (such as wall temperature) and fuel properties (such as volatility) are likely to change the time required for evaporation since they impact the

heat transfer processes involved. The latter part of chapter 6 is devoted to calculating and quantifying the effect of wall temperatures and fuel properties in the evaporation of a liquid layer under repeated cycles.

Chapter 3

Model Formulation

3.1 Model Overview

This chapter covers the governing equations and methodology used in solving the current reactive, diffusive, and convective problem with evaporation at the liquid-gas interface (see Flowchart #1 in Appendix C). Conservation equations in both liquid and gas domains for mass, chemical species, and energy are coupled with species and energy conservation at the interface. Convection is such as to maintain liquid-gas phase equilibrium at the interface of species that are likely to condense or evaporate. Although all the problems considered in this work assumed a single species in the liquid phase, the natural extension of this project is to include the possibility of condensation and evaporation of other species. The code has been structured to allow for these additions and the material is presented in this chapter for this general case, noted where appropriate.

In order to improve the solution procedure, a coordinate transform was applied to the liquid and gas phase conservation equations. Such a transform can be applied to the gas-only, reactive-diffusive problem but is especially advantageous for the current problem since it contains information on the boundary movement and gas convective terms. The next chapter covers this topic in detail and presents the equations in their final form. This chapter focuses on the governing equations and physical processes.

3.2 Governing Equations

3.2.1 Gas Conservation Equations

Equations (3.1), (3.2), and (3.3) describe the gas-phase conservation equations (in physical coordinates) of mass, species, and energy, respectively. The one-dimensional coordinate variable r is used in describing the formulation, along with the parameter a which represents the geometric configuration of the problem: $a = 1$ for flat (cartesian) coordinates, $a = 2$ for cylindrical coordinates, and $a = 3$ for spherical coordinates. Subscripts g and l are used to describe variables in the gas and liquid phases, respectively. In addition to the conservation equations, the equation of state (Equation (3.4)) is used to relate ρ_g, P, and T_g.

Mass

$$\frac{\partial \rho_g}{\partial t} + \frac{1}{r^{a-1}} \frac{\partial}{\partial r} \left(r^{a-1} \rho_g v_g \right) = 0 \tag{3.1}$$

Species

$$\frac{\partial Y_{k,g}}{\partial t} = -v_g \frac{\partial Y_{k,g}}{\partial r} + \frac{1}{\rho_g r^{a-1}} \frac{\partial}{\partial r} \left(r^{a-1} \rho_g D_{k,g} \frac{\partial Y_{k,g}}{\partial r} \right) + \frac{\dot{\omega}_{k,g}}{\rho_g} \tag{3.2}$$

Energy

$$\frac{\partial T_g}{\partial t} = -v_g \frac{\partial T_g}{\partial r} + \frac{1}{\rho_g C_{p,g} r^{a-1}} \frac{\partial}{\partial r} \left(r^{a-1} \lambda_g \frac{\partial T_g}{\partial r} \right) - \frac{1}{\rho_g C_{p,g}} \sum_{k=1}^{K} \dot{\omega}_{k,g} h_{k,g} \tag{3.3}$$

Ideal Gas

$$\rho_g = \frac{P}{R T_g} \tag{3.4}$$

In the above equations, $Y_{k,g}$, $D_{k,g}$, and $h_{k,g}$ represent the mass fraction, diffusivity, and specific enthalpy of species k; $C_{p,g}$ and λ_g the total specific heat and thermal conductivity of the mixture; and v_g the gas velocity in the positive r direction. The equations allow for a chemical kinetic mechanism containing K species. The mass production rates, $\dot{\omega}_{k,g}$, and thermodynamic properties are calculated using the CHEMKIN libraries [10]; transport properties were obtained from the Transport

Package by Kee et. al. [11]. Although the above equations assume a uniform pressure due to the low Mach number of the flow, pressure (which is an input to the above set of equations) may vary over time. In addition, the Dufour and Soret effects are assumed negligible in the problem. Other effects can be incorporated in the above conservation equations and a more complex equation of state can be easily substituted for Equation (3.4).

3.2.2 Liquid Conservation Equations

The liquid phase is modeled as an incompressible fluid, where conservation of mass, species, and energy are observed.

Mass

Due to the incompressible fluid assumption, conservation of total mass in the liquid phase leads to the rate of change of the interface boundary coordinate r_s. For a moving boundary,

$$\frac{dr_s}{dt} = -v_s \frac{\rho_{g,s}}{\rho_{l,s}}, \tag{3.5}$$

where the subscript s is used for values at the interface and v_s is the gas velocity relative to the interface.

Species

In the absence of convection, the fundamental species conservation equation can be expressed as

$$\frac{\partial Y_{k,l}}{\partial t} = \frac{1}{r^{a-1}} \frac{\partial}{\partial r} \left(r^{a-1} D_l \frac{\partial Y_{k,l}}{\partial r} \right). \tag{3.6}$$

The transport mechanism in the liquid phase can be characterized by the liquid Peclet number. It can be defined as the ratio of the boundary regression rate and the diffusion velocity in the liquid phase:

$$Pe_l = r_s \frac{\left|\frac{dr_s}{dt}\right|}{D_l}, \tag{3.7}$$

where D_l is derived from liquid properties. Experiments and calculations have shown [6] that Pe_l derived from liquid properties alone should fall between 20 and 54 for liquid hydrocarbons. Previous droplet calculations with these high Peclet numbers show, however, that much lower rates of evaporation and condensation than experimentally observed would occur for certain species (such as water). This disparity suggests that motion in the liquid domain (most likely induced by outside drag or the deployment process) greatly enhances mass transport in the liquid phase. A compromise solution has often been applied where the liquid diffusivity has been replaced by an effective diffusivity, D_{eff}, forcing the calculated Pe_{eff} to agree with experimental Pe_l.

This approach is avoided here since the mechanisms that generate motion in a liquid layer and how this motion affects transport in the layer are mostly unknown and outside the scope of the present project. In addition, condensation and evaporation of species such as water for the small time scales have a negligible effect on the oxidation process. Rather than assuming a specific large value of D_{eff}, the liquid mass transport has been replaced by a lumped model, where the species concentration $Y_{k,l}$ is uniform throughout the liquid layer. As a result, the overall species conservation equation reduces to

$$\frac{\partial Y_{k,l}}{\partial t} = \frac{A_l}{V_l \rho_l}\left(\dot{m}_k - \rho_l Y_{k,l}\frac{dr_s}{dt}\right), \tag{3.8}$$

where V_l and A_l are the volume and area of the liquid domain corresponding to the problem geometry. The mass flux of species k leaving the liquid domain, $\dot{m}_k$, is obtained from the solution of the eigenvalue of the problem. This solution will be discussed in detail in section 3.4.

Energy

The following equation is solved for energy conservation in the liquid phase:

$$\rho_l C_{p,l} \frac{\partial T_l}{\partial t} = \frac{1}{r^{a-1}} \frac{\partial}{\partial r} \left(r^{a-1} \lambda_l \frac{\partial T_l}{\partial r} \right). \tag{3.9}$$

Although a similar argument concerning the Peclet number could be applied to the energy equation (substituting the thermal diffusivity, α_l, for D_l), it should be noted that the energy equation has a much stronger impact on the transient behavior of the process. This is due to the fact that conditions at the interface (particularly species phase equilibrium values) are very sensitive to temperature and relatively insensitive to the small changes in concentration that may occur due to species condensation.

This is especially true in the current problem where only single species fuels are considered, and where water condensation is a negligible effect regarding the change in fuel concentration in the liquid domain. Thermal gradients, however, can be large and quickly change the liquid surface temperatures, strongly affecting phase equilibrium and vaporization rates. Therefore Equation (3.9) is solved to capture the effect of thermal diffusion in the liquid layer. Since much is still not known about the transport processes involved in the liquid layer, thermal diffusivity is calculated from liquid properties alone. Note that it is possible to easily substitute the liquid thermal diffusivity for an effective value, α_{eff}.

3.3 Boundary and Interface Conditions

Since the model is one-dimensional, the term "left boundary" (and subscript o) is used to identify the boundary at $r = 0$ in the liquid domain either at a rigid wall ($a = 1$ or $a = 2$) or at the center of a spherical droplet ($a = 3$). The term "right boundary" (and subscript ∞) is used for the boundary at $r \to \infty$ in the gas domain. The liquid-gas interface must, of course, fall within the left and right boundaries.

3.3.1 Boundary Conditions

Since the flat geometry case is aimed at modeling a liquid layer adjacent to a rigid wall at constant temperature, the species and energy left boundary conditions for

$a = 1$ are

$$\left(\frac{\partial Y_{k,l}}{\partial r}\right)_o = 0, \tag{3.10}$$

and

$$(T_l)_o = T_w, \tag{3.11}$$

where T_w is the temperature of the rigid wall. Equation (3.11) can easily be modified for an imposed heat flux at the wall, q''_w, thus becoming a Neumann condition,

$$\left(\frac{\partial T_l}{\partial r}\right)_o = \frac{q''_w}{\lambda_l}.$$

For the $a = 3$ case, aimed at modeling a spherical droplet, the left boundary conditions are

$$\left(\frac{\partial Y_{k,l}}{\partial r}\right)_o = 0, \tag{3.12}$$

and

$$\left(\frac{\partial T_l}{\partial r}\right)_o = 0. \tag{3.13}$$

At the right boundary, it is assumed that all species and temperature gradients approach zero, and so, for all a,

$$\left(\frac{\partial Y_{k,g}}{\partial r}\right)_\infty = 0, \tag{3.14}$$

and

$$\left(\frac{\partial T_g}{\partial r}\right)_\infty = 0. \tag{3.15}$$

3.3.2 Liquid-Gas Interface Conditions

Whereas the left and right boundary terms are imposed on the problem, the liquid-gas interface boundary terms must conform to the physical and conservation laws governing the evaporation process.

The fundamental physical process controlling the problem at the interface is the requirement that species mass fractions in the gas domain be in phase equilibrium with those in the liquid domain. Conservation of mass, species, and energy must be satisfied at the boundary, but must also conform to this phase equilibrium requirement. The result is a set of $2(K+1)$ boundary conditions at the interface ($K+1$ species and energy flux conservation equations, K species equilibrium conditions and 1 temperature equality condition) plus an eigenvalue which satisfies the phase equilibrium constraint (a total of $2(K+1)+1$ unknowns). It will be shown that these unknowns define Neumann boundary conditions on each side of the interface for K specie and temperature.

The total mass flow, which controls changes in species concentrations at the interface, is the eigenvalue of the problem. For convenience, the eigenvalue will be represented by the gas velocity at the interface, v_s. Details concerning its solution will be discussed in the next section, while the present section focuses on providing the necessary conservation relations at the boundary.

Species Conservation at the Interface

For all species that may condense or vaporize at the interface, conservation requires that the mass flux from the liquid domain balance with that to the gas domain (all fluxes shown relative to the interface),

$$\dot{m}_{k,l} = \dot{m}_{k,g}.$$

In the gas phase, convection and diffusion are the governing transport mechanisms. As a result,

$$\dot{m}_{k,l} = \rho_{g,s} v_{g,s} Y_{k,g,s} + \left(-\rho_{g,s} D_{k,g,s} \left(\frac{\partial Y_{k,g}}{\partial r} \right) \right), \tag{3.16}$$

where the sign convention is such that convection and diffusion away from the interface are positive. A convenient variable used to describe the species flux at the interface is the fractional gasification rate, ε_k (see next section for details regarding its calculation). It is defined as the fraction of mass flux of species k to the total mass flux at the interface [6]:

$$\varepsilon_k = \frac{\dot{m}_{k,l,s}}{\dot{m}_{l,s}} = \frac{\dot{m}_{k,l,s}}{\rho_{l,s} v_{l,s}} = \frac{\dot{m}_{k,l,s}}{\rho_{g,s} v_{g,s}}, \tag{3.17}$$

where total mass conservation has been applied ($\rho_{l,s} v_{l,s} = \rho_{g,s} v_{g,s}$). Combining the result of Equation (3.17) and Equation (3.16), we arrive at

$$\rho_{g,s} v_{g,s} \varepsilon_k = \rho_{g,s} v_{g,s} Y_{k,g,s} - \rho_{g,s} D_{k,g,s} \left(\frac{\partial Y_{k,g}}{\partial r} \right)_s,$$

or, more explicitly in the form of species boundary conditions (from now on denoting $v_{g,s}$ as v_s),

$$\left(\frac{\partial Y_{k,g}}{\partial r} \right)_s = \frac{v_s (Y_{k,g,s} - \varepsilon_k)}{D_{k,g}}. \tag{3.18}$$

Energy Conservation at the Interface

The energy exchange at the interface may be expressed in terms of energy fluxes from the gas and liquid domains and the energy used for evaporation:

$$\left(\lambda_l \frac{\partial T_l}{\partial r} \right)_{s^-} - \left(\lambda_g \frac{\partial T_g}{\partial r} \right)_{s^+} = \rho_{g,s} v_s \sum_{k=1}^{K} \varepsilon_k L_k, \tag{3.19}$$

where L_k is the enthalpy of vaporization of species k.

3.4 Eigenvalue Solution

3.4.1 Temperature and Mass Fractions at the Interface

Since this section deals with terms at the interface, the subscript s will be dropped to simplify the notation. The problem as described above is not yet completely specified, since there are $2(K+1)$ interface boundary terms (species and temperature conditions on both sides of the interface) but only $K+1$ species and energy conservation equations (Equations (3.18) and (3.19)). The remaining $K+1$ equations result from considering temperature equality and species phase-equilibrium at the interface:

$$T_g = T_l, \tag{3.20}$$

$$X_{k,g} = x_{k,l}\frac{P_k}{P}, \tag{3.21}$$

where $X_{k,g}$ and $x_{k,l}$ denote molar concentrations at the interface in the gas and liquid phases, respectively, and P_k is the equilibrium vapor pressure of species k. Vapor pressures can be calculated for either single or multi-component liquid mixtures using a relation such as the Antoine equation [12]:

$$P_k = 10^{A_k - \frac{B_k}{T_l + C_k}} \tag{3.22}$$

where A_k, B_k, and C_k are empirically determined for each species. More complex alternatives to Equation (3.22) can easily be implemented in the model (section 6.9 addresses an alternative expression). Equation (3.21) is expressed in terms of mole fractions, whereas the variable that is solved for in the present formulation is the mass fraction. It can be easily extracted from mole fractions with the conversion:

$$Y_k = W_k X_k \sum_{k=1}^{K} \frac{Y_k}{W_k}. \tag{3.23}$$

Equation (3.23) has an implicit dependence on liquid concentration, liquid temperature, and total pressure via X_k and Y_k. Therefore, to express the evolution of the boundary condition (3.21) implicitly in time, the time variation of these variables is

necessary. For the present problem, the pressure variation in time is usually an input and not calculated. The liquid concentration and temperature, however, must abide to the respective liquid and gas conservation equations on both sides of the interface.

The solution of the problem (chapter 4) is carried out by integrating the conservation equations in time. As a result, equations (3.20) and (3.23) must be expressed as time derivatives (adopting shorthand notation for d/dt terms):

$$\dot{T}_g = \dot{T}_l, \tag{3.24}$$

$$\dot{Y}_{k,g} = W_{k,g}\left(\dot{X}_{k,g}\sum_{j=1}^{K}\frac{Y_{j,g}}{W_{j,g}} + X_{k,g}\sum_{j=1}^{K}\frac{\dot{Y}_{j,g}}{W_{j,g}}\right). \tag{3.25}$$

The $\dot{X}_k$ term can be expressed as

$$\dot{X}_{k,g} = \frac{\partial X_{k,g}}{\partial P}\frac{\partial P}{\partial t} + \frac{\partial X_{k,g}}{\partial T_l}\frac{\partial T_l}{\partial t} + \frac{\partial X_{k,g}}{\partial x_{k,l}}\frac{\partial x_{k,l}}{\partial t}. \tag{3.26}$$

Equations (3.24) and (3.25) provide the necessary closure and the problem is completely specified. There are now $2(K+1)+1+K$ unknowns (species and temperature interface boundary terms on both sides of the interface plus an eigenvalue plus fractional gasification ratios) and $2(K+1)+K$ equations ($K+1$ (3.18), $K+1$ (3.19), and K (3.25)). An additional equation can be obtained by observing that the fractional gasification ratios must conform to

$$\sum_{k=1}^{K}\varepsilon_k = 0. \tag{3.27}$$

The next section addresses how, by combining the equations above, it is possible to solve for the problem eigenvalue and the interface boundary terms.

3.4.2 Combining Equations and Eigenvalue Solution

Since phase equilibrium controls the problem at the interface, the governing processes at the interface are combined into Equation (3.25). In theory, for a problem where a total of K species are present in the liquid and gas phases, up to K simultaneous

equations (3.25) can exist.

It has been found in practice, however, that this is rarely the case [6]. For most physically significant problems only a few species will experience condensation or evaporation to a significant extent. In the case of most hydrocarbon evaporation and combustion problems, these species are the hydrocarbon itself and water. Both may condense ($\varepsilon_k < 0$) or evaporate ($\varepsilon_k > 0$) at the boundary. All other gas species may be considered to show no interaction with their liquid phase counterparts at the boundary, resulting in no condensation and no evaporation ($\varepsilon_k = 0$). The implication of this fact will become clear as Equation (3.25) is developed below.

Combining equations (3.2), (3.25), and (3.26),

$$\begin{aligned}
&-v_s\left(\frac{\partial Y_{k,g}}{\partial r}\right)_s + \frac{1}{\rho_g r^{a-1}}\frac{\partial}{\partial r}\left(r^{a-1}\rho_g D_{k,g}\frac{\partial Y_{k,g}}{\partial r}\right)_s + \frac{\dot{\omega}_k}{\rho_g} \\
&= W_{k,g}\left(\frac{\partial X_{k,g}}{\partial P}\frac{\partial P}{\partial t} + \frac{\partial X_{k,g}}{\partial T_l}\frac{\partial T_l}{\partial t} + \frac{\partial X_{k,g}}{\partial x_{k,l}}\frac{\partial x_{k,l}}{\partial t}\right)_s \sum_{j=1}^{K}\frac{Y_{j,g}}{W_{j,g}} \\
&+W_{k,g}X_{k,g}\sum_{j=1}^{K}\frac{1}{W_j}\left(-v_s\left(\frac{\partial Y_{j,g}}{\partial r}\right)_s + \frac{1}{\rho_g r^{a-1}}\frac{\partial}{\partial r}\left(r^{a-1}\rho_g D_{j,g}\frac{\partial Y_{j,g}}{\partial r}\right)_s + \frac{\dot{\omega}_j}{\rho_g}\right).
\end{aligned} \tag{3.28}$$

At this point, it remains to show how Equation (3.28) forms a set of K equations that only have the eigenvalue v_s and fractional gasification ratios ε_k as unknowns. The equations can then be expressed as non-linear functions of v_s, whose root is the problem eigenvalue, and quickly solved for by a Newton-Raphson iteration in the computer code.

All of the species spatial derivatives in Equation (3.28) can be replaced using Equation (3.18):

$$\left(\frac{\partial Y_{k,g}}{\partial r}\right)_s = \frac{v_s(Y_{k,g,s} - \varepsilon_k)}{D_{k,g}}.$$

These spatial derivatives occur both explicitly in the convective terms and implicitly in the diffusion terms above. Although there are still other terms to be examined,

the incorporation of Equation (3.18) into (3.28) immediately shows that $(\partial Y_{k,g}/\partial r)_s$ terms will be eliminated and replaced by functions whose only unknowns are v_s and ε_k.

To show that the remaining unknown terms in Equation (3.28), $(\partial T_l/\partial t)_s$ and $(\partial x_{k,l}/\partial t)_s$, are dependent only on v_s and ε_k, energy and species conservation across the boundary are considered. First addressing $(\partial T_l/\partial t)_s$, Equations (3.3) and (3.9) can be combined with (3.24) to obtain

$$-v_g\left(\frac{\partial T_g}{\partial r}\right)_s + \frac{1}{\rho_g C_{p,g} r^{a-1}}\frac{\partial}{\partial r}\left(r^{a-1}\lambda_g\frac{\partial T_g}{\partial r}\right)_s - \frac{1}{\rho_g C_{p,g}}\sum_{k=1}^{K}\dot{\omega}_k h_k = \frac{1}{\rho_l C_{p,l}}\frac{1}{r^{a-1}}\frac{\partial}{\partial r}\left(r^{a-1}\lambda_l\frac{\partial T_l}{\partial r}\right)_s . \tag{3.29}$$

Since $(\partial T_g/\partial r)_s$ and $(\partial T_l/\partial r)_s$ are related by

$$\left(\lambda_l\frac{\partial T_l}{\partial r}\right)_s - \left(\lambda_g\frac{\partial T_g}{\partial r}\right)_s = \rho_{g,s} v_s \sum_{k=1}^{K}\varepsilon_k L_k, \tag{3.30}$$

Equations (3.29) and (3.30) can be combined to arrive at

$$\left(\frac{\partial T_l}{\partial r}\right)_s = f_{T_{l\partial r}}\left(v_s, \varepsilon_1, ..., \varepsilon_K\right), \tag{3.31}$$

which implies

$$\left(\frac{\partial T_l}{\partial t}\right)_s = \frac{1}{\rho_l C_{p,l}}\frac{1}{r^{a-1}}\frac{\partial}{\partial r}\left(r^{a-1}\lambda_l\frac{\partial T_l}{\partial r}\right)_s = f_{T_{\partial t}}\left(v_s, \varepsilon_1, ..., \varepsilon_K\right). \tag{3.32}$$

This is the desired result, since it shows that, due to the contribution of Equation (3.32), the $(\partial T_l/\partial r)_s$ term in equation (3.28) will be replaced by functions whose unknowns are still only v_s and ε_k. We note that Equations (3.31) and (3.32) introduce the effect of the energy equation to the solution of the eigenvalue. The detailed forms of $f_{T_{l\partial r}}$ and $f_{T_{\partial t}}$ are presented in Appendix B.

Finally, it remains to be shown that $(\partial x_{k,l}/\partial t)_s$ is a function of only v_s and ε_k to close the problem. It should be noted that this term will only exist in the case of

a multi-component liquid layer, since it is identically zero for the single-component liquid. Therefore, single-component calculations (problems considered here) can avoid the additional computational complexity of the remainder of this section.

Since

$$x_{k,l} = \frac{Y_{k,l}/W_{k,l}}{\sum_{j=1}^{K} Y_{j,l}/W_{j,l}}$$

and again employing the shorthand notation for time derivatives, this term can be expressed as

$$\dot{x}_{k,l} = \frac{\dot{Y}_{k,l}/(W_{k,l}\sum_j Y_{j,l}/W_{j,l}) - Y_{k,l}/W_{k,l}\sum_j \dot{Y}_{j,l}/W_{j,l}}{(\sum_j Y_{j,l}/W_{j,l})^2}. \tag{3.33}$$

From Equation (3.8) $\dot{Y}_{k,l}$ has already been determined as

$$\dot{Y}_{k,l} = \frac{A_l}{V_l \rho_l}\left(\dot{m}_k - \rho_l Y_{k,l}\frac{dr_s}{dt}\right), \tag{3.34}$$

where $\dot{m}_k = \dot{m}_{total}\varepsilon_k = \rho_{g,s} v_s \varepsilon_k$ and $dr_s/dt = -v_s \rho_{g,s}/(\rho_{l,s} - \rho_{g,s})$, from (3.17) and (3.5), respectively (similarly for $\dot{Y}_{j,l}$ in the summations). It is therefore shown that

$$\left(\frac{\partial x_{k,l}}{\partial t}\right)_s = f_{x_{k\partial t}}(v_s, \varepsilon_1, ..., \varepsilon_K). \tag{3.35}$$

Again, this is the desired result, since it shows the substitution of $(\partial x_{k,l}/\partial t)_s$ terms in Equation (3.28) will result in the incorporation of functions of v_s and ε_k only. The detailed form of $f_{x_{k\partial t}}$ is also presented in Appendix B.

Now that Equation (3.28) has been shown to only have unknowns v_s and ε_k it is possible to solve the eigenvalue problem. Considering the case of a two-component liquid mixture where $\varepsilon_k = 0$ for all but, say, the fuel and water species, there are a total of 3 unknowns: ε_f, ε_w, and v_s. In this case, there will be two equations of the form (3.28) plus the constraint of equation (3.27).

The result is a closed system of equations, with three equations and three unknowns. Although the inclusion of more than two species in the convection/evaporation process is not physically significant and will not be pursued, the argument above could

extend to any number of $\tilde{K}+1$ equations and $\tilde{K}+1$ unknowns, where $\tilde{K} \leq K$.

3.5 Result: Interface Boundary Terms

The last section showed that the problem to be solved includes $\tilde{K}$ Equations (3.28), 1 Equation (3.27), and $\tilde{K}+1$ unknowns ($\tilde{K}$ number of ε_k and one eigenvalue v_s) in the general case of $\tilde{K}$ unknown ε_k (for the problems treated here, $\tilde{K}=1$ and $\varepsilon_f = 1$). Once ε_k and v_s have been solved for, they can be used to determine the problem's species and temperature "boundary" conditions at the liquid-gas interface from the equations used to substitute for their values into (3.28) (also summarized below).

Gas-side Species Neumann Condition

$$\left(\frac{\partial Y_{k,g}}{\partial r}\right)_s = \frac{v_s(Y_{k,g,s}-\varepsilon_k)}{D_{k,g,s}}, \tag{3.36}$$

Gas-side Temperature Neumann Condition

$$\left(\frac{\partial T_g}{\partial r}\right)_s = \frac{1}{\lambda_{g,s}}\left(f_{T_{l\partial r}}(v_s,\varepsilon_{1\ldots K}) - \rho_{g,s}v_s\sum_{k=1}^{K}\varepsilon_k L_k\right), \tag{3.37}$$

Liquid-side Species Dirichlet Condition

$$\left(\frac{\partial Y_{k,l}}{\partial t}\right)_s = \frac{A_l v_s}{V_l \rho_l}\left(\rho_{g,s}\varepsilon_k + \rho_l Y_{k,l}\frac{\rho_{g,s}}{\rho_{l,s}-\rho_{g,s}}\right), \tag{3.38}$$

Liquid-side Temperature Neumann Condition

$$\left(\frac{\partial T_g}{\partial r}\right)_s = f_{T_{l\partial r}}(v_s,\varepsilon_{1\ldots K}). \tag{3.39}$$

The problem is thus completely specified, with imposed boundary conditions (section 3.3.1) at o and ∞ and calculated boundary conditions (3.36)-(3.39) at the liquid-gas interface.

Chapter 4

Solution Procedure

This chapter describes the procedure used in solving the problem. The solution is shown in terms of a problem consisting of a single species in the liquid phase. Flowchart #2 in Appendix D summarizes the logical steps in the process of determining the solution and outputs from imposed boundary and initial conditions. All boxes containing inputs for the problem are bordered by double lines. The box labeled "governing equations solution procedure" refers to the procedure described in chapter (3).

The "solution vector" shown in the chart is the vector that contains the profiles of the species and temperature in both the liquid and gas domains for all points in the discretization. Applying the method of lines (MOL) to the problem domain, for K species profiles, one temperature profile, and one boundary location, with N_g spatial points in the gas phase and N_l points in the liquid phase, the solution vector $\mathbf{v}$ consists of $(N_g(K+1)+N_l+1)$ elements:

$$\mathbf{v} = (Y_{g1,1}, Y_{g2,1}, ..., Y_{gK,1}, T_{g1}; Y_{g1,2}, ..., T_{g,N_g}; T_{l,1}, ..., T_{l,N_l}; r_s).$$

A coordinate transform is applied to the original variables to simplify the governing equations. A meshing routine is then applied to the transformed domain to obtain a discretization that allows for efficient and well resolved solution of the profiles.

Given a certain arrangement of the solution vector, it is possible to specify the

structure of the Jacobian matrix that will be used for time integration. The solution vector can be arranged so that the structure of the Jacobian is banded, which allows for more efficient memory usage and matrix inversion. The sparse Jacobian version of the ordinary differential equation solver LSODES [5] is used to integrate the problem in time. LSODES also solves for the values in the Jacobian by difference quotients.

Due to the stiff nature of the system of equations, the option for backward differentiation formulas was enabled in the integrator, with order up to 5. The integrator usually employs a modified Newton iteration method to perform the time integration, which automatically determines when a new Jacobian must be calculated based on error estimates. The integrator package has been modified in order to allow for a check concerning the difference between the meshes of current and previous times steps. If this difference is greater than some arbitrarily set maximum (i.e., the mesh distribution no longer resolves the profiles properly) the modified LSODES allows for a break from the integrator and a redefinition of the solution vector based on a new, more appropriate mesh.

4.1 Coordinate Transforms

Coordinate transforms are applied to the governing equations in the liquid and gas phases to aid in solving the problem. These transforms are useful in both simplifying the form of the equations and in keeping track of information concerning the boundary movement in each domain.

4.1.1 Gas Phase Coordinate Transform and Equations

The following spatial coordinate transform is applied to the physical coordinate r in the gas phase [7]:

$$\eta(r) = \int_{r_s}^{r} (r')^{a-1} \rho_g \, dr'. \tag{4.1}$$

This transform weighs the density of gases with the original coordinate location.

If the location of the liquid-gas interface (the left boundary of the gas domain) r_s is known, the original physical coordinates can be restored by the following reverse transformation

$$r(\eta) = \left[r_s^a + a \int_{\eta_s=0}^{\eta} \left(\frac{1}{\rho_g} \right) d\eta \right]^{1/a}. \tag{4.2}$$

The time coordinate is transformed by $\tau(t) = t(\tau)$. One advantage of applying these transforms is that mass conservation is automatically satisfied and eliminated from the original system of equations. This will also lead to the expression of the original derivatives in terms of transformed derivatives.

Observing mass conservation in a control volume that extends from the moving liquid interface location, r_s, to a stationary point $r > r_s$,

$$\int_{r_s}^{r} \frac{\partial \rho_g}{\partial t} r^{a-1} \, dr + \int_{(r_s^{a-1}\rho_{g,s}v_s)}^{(r^{a-1}\rho_g v)} d(r^{a-1}\rho_g v) = 0 \tag{4.3}$$

leads to

$$\frac{\partial}{\partial t} \int_{r_s}^{r} (r^{a-1}\rho_g) \, dr - \left[r^{a-1}\rho_g \frac{\partial r_{c.v.}}{\partial t} \right]_{r_s}^{r} + r^{a-1}\rho_g v - r_s^{a-1}\rho_{g,s}v_s = 0, \tag{4.4}$$

where $(\partial r_{c.v.}/\partial t)_r = 0$ and $(\partial r_{c.v.}/\partial t)_{r_s} = -(\rho_g/\rho_l)v_s$ (Equation (3.5)). Equation (4.4) can be rewritten as

$$\frac{\partial \eta}{\partial t} = \dot{m}_s - r^{a-1}\rho_g v \tag{4.5}$$

where

$$\dot{m}_s = \left[r^{a-1}\rho_g v_g \left(1 + \frac{\rho_g}{\rho_l} \right) \right]_{r=r_s} \tag{4.6}$$

is the mass flux at the liquid-gas interface. From Equation (4.1), we also observe that

$$\frac{\partial \eta}{\partial r} = r^{a-1}\rho_g. \tag{4.7}$$

Finally, Equations (4.5) and (4.7) can be used to express the original spatial and time derivatives in terms of the transformed coordinates:

$$\frac{\partial}{\partial t} = \frac{\partial}{\partial \tau}\frac{\partial \tau}{\partial t} + \frac{\partial}{\partial \eta}\frac{\partial \eta}{\partial t} = \frac{\partial}{\partial \tau} + (\dot{m}_s - r^{a-1}\rho_g v)\frac{\partial}{\partial \eta} \tag{4.8}$$

and

$$\frac{\partial}{\partial r} = \frac{\partial}{\partial \tau}\frac{\partial \tau}{\partial r} + \frac{\partial}{\partial \eta}\frac{\partial \eta}{\partial r} = r^{a-1}\rho_g \frac{\partial}{\partial \eta}. \tag{4.9}$$

Transformed Equations

Lagrangian derivatives for any quantity β can be obtained from Equations (4.8) and (4.9),

$$\frac{\partial \beta}{\partial t} + v\frac{\partial \beta}{\partial r} = \frac{\partial \beta}{\partial \tau} + \dot{m}_s \frac{\partial \beta}{\partial \eta}, \tag{4.10}$$

which can be substituted into the physical coordinates conservation equations (3.2) and (3.3) to yield:

$$\frac{\partial Y_{k,g}}{\partial \tau} = -\dot{m}_s \frac{\partial Y_{k,g}}{\partial \eta} + \frac{\partial}{\partial \eta}\left(r^{2(a-1)}\rho_g^2 D_k \frac{\partial Y_{k,g}}{\partial \eta}\right) + \frac{\dot{\omega}_k}{\rho_g}, \tag{4.11}$$

$$C_{p,g}\frac{\partial T_g}{\partial \tau} = -C_{p,g}\dot{m}_s \frac{\partial T_g}{\partial \eta} + \frac{\partial}{\partial \eta}\left(r^{2(a-1)}\rho_g \lambda_g \frac{\partial T_g}{\partial \eta}\right) + \sum_{k=1}^{K}\frac{\dot{\omega}_k h_k}{\rho_g}. \tag{4.12}$$

Equation (3.1) (mass conservation) has already been satisfied by deriving the transforms above (Equation (4.3)). One major advantage that has resulted from the coordinate transform is that the gas velocity at any location, $v(r)$, has been replaced by $\dot{m}_s$. This term is constant in r and is the mass flux associated with v_s, the problem eigenvalue (see section 3.4 for details on the solution of v_s). In addition, the movement of the liquid-gas interface is automatically corrected in the transformed coordinates by the dr_s/dt term from Equation (4.4) (also dependent on v_s).

4.1.2 Liquid Phase Coordinate Transform and Equations

The liquid mass conservation is much simpler than its gas-phase equivalent (the liquid is treated as an incompressible substance with a known mass flux out, $\dot{m}_s$), resulting in a considerably simpler transform procedure. Since the density is constant, it does not need to be included it in the transform, which only needs to reflect the nature of the boundary movement [7]:

$$\phi(r) = \frac{r}{r_s}. \tag{4.13}$$

From Equation (3.9), the transformed energy equation in the liquid phase can be obtained:

$$\frac{\partial T_l}{\partial t} = \frac{1}{\rho_l c_l r_s^2 \phi^{a-1}} \frac{\partial}{\partial \phi}\left(\phi^{a-1}\lambda_l \frac{\partial T_l}{\partial \phi}\right) \tag{4.14}$$

for a stationary mesh in the liquid phase ($\partial\phi/\partial t = 0$). Since the liquid is treated as a lumped model for species conservation, the transform is not applicable to the species equation.

4.1.3 Considerations Regarding Coordinate Transforms

Since the gas-phase coordinate transforms are spatial integrals of density, the spacing of a mesh that is spread uniformly in transformed coordinates will be dependent on the gas density distribution when mapped to physical coordinates. Temperature is the variable with the strongest (inverse) influence on density. The result is that in areas of high temperatures, physical coordinates will have larger spacing than areas of low temperatures. This can be seen from Equation (4.2). For a uniform mesh in transformed coordinates, mesh point i will be mapped from η_i to r_i as

$$r_i = \left[r_s^a + a\int_0^{\eta_i}\left(\frac{RT_g}{p}\right) d\eta\right]^{1/a}. \tag{4.15}$$

A high temperature would lead to a low density and consequently a large value for the integral term in Equation (4.15). Therefore, if temperature is increasing, r_i

will be larger than r_{i-1}, which increases the physical coordinate spacing.

This temperature effect on the spacing can be important if not enough definition results in insufficient resolution in physical coordinates to resolve a given profile. The next section covers the distribution of mesh points in the gas-phase in transformed coordinates. The meshing is done in such a way as to emphasize the details of the profiles in *transformed* coordinates, ignoring the effect described above. Therefore, the resulting *physical* mesh spacing was checked for each run to assure good definition in the physical coordinate spacing.

It was found that under most circumstances the meshing method described in the next section was sufficient to assure a well distributed mesh in both transformed and physical coordinates. Under some conditions, notably when a steep (in relation to the length of the problem domain) rise in the temperature profile existed near a very thin hot zone (i.e., thinner than a flame), the above problem was encountered. Since these situations were only encountered under artificially unfavorable conditions, they were not found to be important for the present problem.

4.2 Gas Domain Adaptive Meshing Procedure

In order to aid the solution process of the gas phase equations, an adaptive meshing procedure was employed to the transformed coordinate domain. Although this method could be applied to either transformed or physical coordinates, the equations are solved in the transformed space, which therefore require better resolution than the physical space. As an experiment, adaptive meshing in transformed and physical spaces were attempted. It was found that the code performance improved significantly in the former case (run times decreased by a factor of 5 compared to static meshes) and actually deteriorated in the latter.

4.2.1 General Procedure

The meshing procedure [8] is based on the equidistribution of a mesh function $W(\eta)$ (described in the next section) on domain η. First, W is calculated as a function of given criteria in η. It is important that the criteria be such that $\frac{\partial W}{\partial \eta} \geq 0$ for all η. Once W is obtained, it must be normalized so that $W(0) = 0$ and $W(\eta_L) = 1$, where η_L is the right boundary of the domain of length L. The normalized W can then be uniformly divided from 0 to 1 on its axis and back-interpolated onto η for a new mesh distribution.

Figure (4-1) is a schematic representation of the process. For a given function $f(\eta)$ of arbitrary profile (represented by the dashed line), a mesh function $W(\eta)$ (solid line) can be obtained to emphasize the regions of steep gradients. By uniformly distributing $W(\eta)$ on the y-axis and back interpolating it onto η, a new mesh is obtained.

Algorithmically, the procedure is as follows:

1. $W(\eta)$ is calculated for $\eta = [0, \eta_L]$;
2. W is normalized in η and equally divided in its own axis into W_i, $1 \leq i \leq N$, where N is the number of desired points in the new mesh;
3. For point i from 1 to N:
   (a) if $i = 1$, locate the first new point at $\eta_0 = 0$;

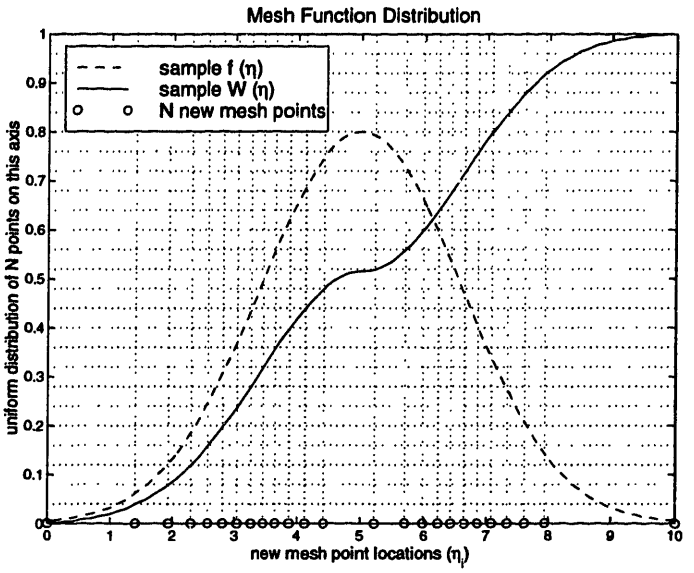


Figure 4-1: $W(\eta)$ is uniformly distributed on the y-axis and back-interpolated onto η. The new mesh distribution reflects characteristics of the profile of $f(\eta)$ from which $W(\eta)$ is derived (see next section)

(b) if $1 < i < N$, locate a new point by linearly back-interpolating from W_i onto η_i (Figure 4.1):

$$\eta_i = \eta_a + (\eta_b - \eta_a)\frac{W_i - W_a}{W_b - W_a}$$

where a and b refer to points on the old mesh between which W_i is found;

(c) if $i = N$, locate the last new point at $\eta_N = \eta_L$; end.

The procedure will then result in a mesh distribution on η that reflects the profile of W. Regions of large $\partial W/\partial\eta$ will produce dense point distributions in the new mesh, while regions of small $\partial W/\partial\eta$ will produced widely spaced distributions. This procedure automatically accommodates for any number of N desired points, which may change every time the meshing procedure is carried out.

One obvious limitation of the procedure is that the resolution of W depends on the distribution of the previous mesh. If the previous mesh does not accurately resolve W, its profile will not be correctly back-interpolated onto η, and an imperfect distribution could result. For this reason it is important to remesh frequently during the problem

time integration whenever the profiles are changing rapidly in time. However, since the mesh is a new finite discretization of the domain, there is a time penalty for each remesh due to the fact that a new Jacobian matrix must be calculated for every new discretization.

It is therefore necessary to determine an appropriate time during the calculation in which to perform the remeshing on the domain. As will be described in section 4.3, the time integration is performed by the LSODES package. The most costly part of the time integration procedure is the calculation of the Jacobian matrix, which must be carried out whenever the modified Newton iteration does not converge. Therefore, in order to avoid the calculation of a new Jacobian due only to remeshing, the meshing procedure is called between Jacobian calculations as determined in LSODES (see Flowchart #2). It was found that the Jacobian must be recalculated most frequently when the profiles are changing rapidly in time, resulting in a frequency of remeshing that adequately resolved the domain for all cases observed. Finally, an additional constraint was imposed in that the remeshing procedure is only called when a difference in W between an old and a potential new mesh is greater than a predetermined, arbitrary percentage.

4.2.2 Weight Functions $W(\eta)$

$W(\eta)$ may be arbitrarily specified to emphasize any aspect of the profiles to be resolved. For the problem in question, however, the governing equations require a certain degree of resolution in particular regions of the profiles for accurate finite-difference calculations. From Equations (4.11) and (4.12), it is evident that the first and second derivatives of the species and temperature profiles in η must be well resolved. In addition, we recognize that all the species and temperature profiles may be included in determining W, and so define

$$W(\eta) = \frac{\sum_{k=1}^{K+1} w_k(\eta)}{\sum_{k=1}^{K+1} w_k(L)}, \tag{4.16}$$

which includes the weight functions for all K species and temperature profiles and

normalizes the results with respect to the highest values of the mesh functions (at $\eta = L$ since $\frac{\partial W}{\partial \eta} \geq 0$ for all η).

For any function $Y_k(\eta)$ (such as the profile of a particular species mass fraction or gas temperature), weight functions for Equation (4.16) can be defined:

$$w_k(\eta) = K_k + a_k \frac{\int_0^\eta \left|\frac{\partial Y_k}{\partial \eta}\right| d\eta}{\int_0^L \left|\frac{\partial Y_k}{\partial \eta}\right| d\eta} + b_k \frac{\int_0^\eta \left|\frac{\partial^2 Y_k}{\partial \eta^2}\right| d\eta}{\int_0^L \left|\frac{\partial^2 Y_k}{\partial \eta^2}\right| d\eta}, \tag{4.17}$$

where a_k, and b_k are constants that weigh the first and second derivatives of the profile of Y_k, respectively [8]. If the constants are chosen such that $a_k/b_k > 1$, for example, the first derivative will be more strongly reflected on W and it will dominate the distribution of points when remeshing is performed. The constant K_k has no dependence of Y_k and provides a smoothing of W regardless of the other weight criteria. As $K_k/a_k \to \infty$ and $K_k/b_k \to \infty$ the points (after normalization and remeshing) approach a perfectly uniform distribution.

There are different ways in which K_k may be chosen. According to [8], it is convenient to define a certain factor f such that

$$f = \frac{\int_0^L W_{K_k=0}\, d\eta}{\int_0^L W_{K_k}\, d\eta} \tag{4.18}$$

where $W_{K_k=0}$ and W_{K_k} represent the weight function calculated with $K_k = 0$ and $K_k \neq 0$. Under most circumstances f will be of the order of 1.0 since the smoothing will not have a drastic impact on the integral of the mesh function. Different K_k will result in different f, which must be arbitrarily picked so that it will produce well distributed meshes for the particular geometry in question. A recommended value of f [8] is $f = 1.5$ for spherical geometries. It was found that f between 1.1 and 1.3 were satisfactory for all geometries considered here. The value of $f = 1.1$ was chosen for all cases and the corresponding K_k used in Equation (4.16).

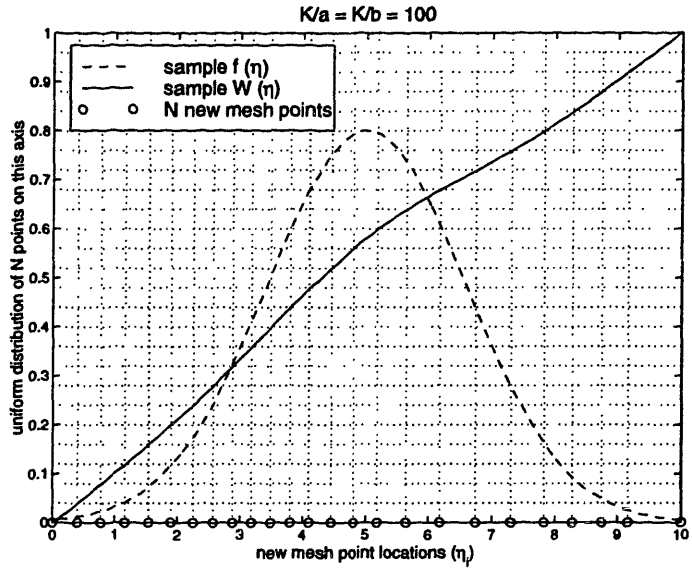


Figure 4-2: New mesh distribution of Figure (4.1) with $K/a = K/b = 100$

4.2.3 Mesh Function Examples

Figure (4-1) was obtained with the following parameters in determining $W(\eta)$: $K/a = K/b = 0.5$ (since there is only one curve in this example, only one K, one a, and one b are necessary). In order to display the smoothing effect of increasing K mentioned above, Figure (4-2) shows the resulting mesh of the same function with $K/a = K/b = 100$. It shows that the new mesh function is smoother than in the previous figure, resulting in a more uniformly distributed mesh.

The fraction between the integral of the two mesh functions (Equation 4.18) is $f = 1.05$ in this case. This value illustrates the observation that a small change in f may distributed the mesh significantly. This sensitivity will decrease as the gradients in the original function increase. For the gradients present in combustion profiles, the values of f mentioned above are more appropriate.

Figure (4-3) shows the mesh distribution for a set of profiles similar to those of a calculated flame. The fuel species has been given grater weight than other species ($a_{fuel}, b_{fuel} = 5a_{other}, 5b_{other}$) and $f = 1.1$. The profiles shown are normalized to their largest value in the domain (mesh function and other species profiles not shown). The mesh is shown to be more refined in the region of large gradients in the fuel profile,

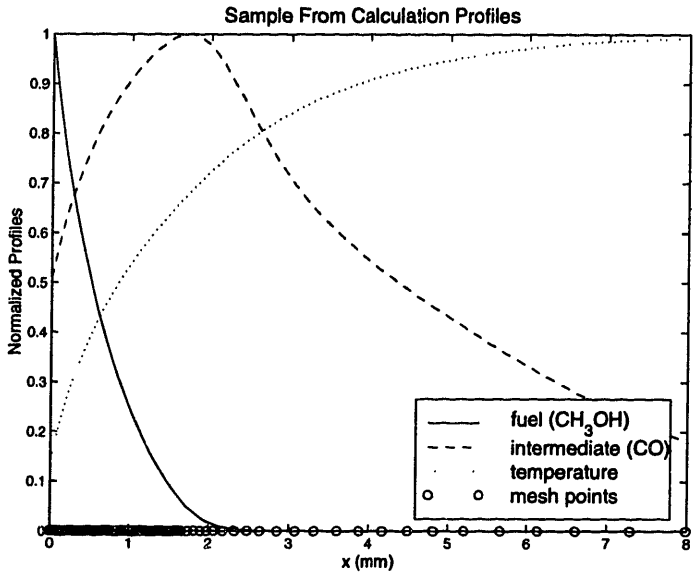


Figure 4-3: Mesh distribution for flame-like profiles (sample calculation)

due to its larger weight. As the gradients of all species decrease (for increasing x in this case) the mesh becomes sparser, as desired.

4.3 Numerical Aspects and Time Integration

4.3.1 Discretization and Differencing

The equations to be integrated over time (Equations (4.11) and (4.12)) include a convective, a diffusive, and a reactive term. The convective and diffusive terms must be discretized and the reactive term may be seen as the main source of stiffness in the equations. It is therefore expected (as will be shown in the next section) that a stiff solver procedure must be used to solve the equations. This procedure is the backward differentiation formula (BDF) method and is implemented by the use of the LSODES package.

The implementation of the method of lines is executed through the meshing procedure described above (given an initial mesh of the problem). From consideration of simpler integration schemes it can be shown [13] that central and upwind differencing for the diffusion and convection terms, respectively, are both stable and accurate,

and were therefore used in the code. Upwind differencing is first order in space and central differencing is a second order. Although higher orders were attempted, they proved to be unstable.

4.3.2 Time Integration

Once the problem has been discretized, the governing equations can be expressed in the following form [13]:

$$\mathbf{A}\frac{\partial \mathbf{v}}{\partial t} = \mathbf{g}(\mathbf{v}, t) \tag{4.19}$$

where $\mathbf{g}$ represents the conservation equations (4.11)-(4.13), $\mathbf{A} = \mathbf{I}$ here, and $\mathbf{v}$ is the solution vector. As mentioned above, the sparse Jacobian version of the LSODES package was used for time integration [5]. It implements the Gear algorithm (a specific form of BDF formulation) to discretize the problem in time, which allows for an implicit term of order ω, where ω is the number of previous time steps to be included. The resulting implicit equation is solved by a Modified-Newton Iteration procedure,

$$[\mathbf{A}(\mathbf{v}, t_j) - \Delta t_j \sigma_o \mathbf{J}(\mathbf{v}, t_j)](\mathbf{v}^{k+1}(t_j) - \mathbf{v}^k(t_j)) = -h\Delta t_j \sigma_o \mathbf{F}(\mathbf{v}, t_j), \tag{4.20}$$

where $\mathbf{v}^{k+1}(t_j)$ is the solution vector for a new time step Δt_j, and σ_o is a Gear coefficient. The term $\mathbf{g}$ as well as other Gear coefficients are in $\mathbf{F}$ of Equation (4.20) (see [5] and [13] for details). The structure of the Jacobian matrix $\mathbf{J}$ ($J_{i,l} = \partial F_i / \partial v_l$) is supplied directly to LSODES which allows for sparse structures in order to conserve memory and improve inversion. Since the problem includes first and second order difference terms in $\mathbf{g}$, and thus in $\mathbf{F}$, the structure of the Jacobian is block tridiagonal. For N equations to be solved, each block is of size N. From the solution vector, it is evident that $N = (N_g(K+1) + N_l + 1)$.

LSODES recalculates the Jacobian whenever the Modified-Newton Iteration pro-

cedure fails to converge. Since each new mesh definition redefines the solution vector **v**, the Jacobian matrix must be recalculated whenever remeshing is performed. This is an numerically intensive procedure, and thus remeshing was performed only between Jacobian calculations as determined in LSODES. This required a slight code modification to LSODES, which allows for a check and exit if a new Jacobian is to be calculated (see Flowchart #2). This method proved satisfactory for all cases attempted.

Chapter 5

Model Verification

5.1 Analytical Comparison: Droplet Evaporation

In order to verify the stability and accuracy of the model, it is necessary to compare its solution to the governing equations with known analytical or experimental results. For certain simplified cases, it is possible to obtain analytical solutions for the evaporation and combustion processes near a liquid droplet or layer. This section compares the analytical quasi-steady-state solution of an evaporating, non-reacting droplet with numerical results.

Analytical Solution

Under quasi-steady-state conditions, an evaporating droplet attains constant liquid-gas interface temperature and species concentrations. As a result, the problem reduces to a mass and energy balance at the interface with no unsteady terms. To simplify the analytical solution of the problem further, constant gas and liquid properties have been assumed. Considering only mass and energy conservation in the gas phase (and dropping the g subscript) [14]:

$$v_s r_s^2 = v r^2, \tag{5.1}$$

$$\frac{1}{r^2}\frac{d}{dr}\left(\lambda r^2 \frac{dT}{dr}\right) - \rho C_p v \frac{dT}{dr} = 0. \tag{5.2}$$

Since the process is in quasi-steady-state, the species balance at the interface is not required in the solution of v_s. This occurs since at constant temperature, the species mass fraction at the interface must be constant and $\dot{Y}_{k,s} = 0$. Therefore Equation (3.25) is not required for the solution and only Equation (3.3) (in the form of Equation (5.2)) is necessary. Equations (5.1) and (5.2) may be combined and written as:

$$\frac{d}{dr}\left(r^2\frac{dT}{dr}\right) - \frac{v_s r_s^2}{\alpha r^2}\left(r^2\frac{dT}{dr}\right) = 0, \tag{5.3}$$

which is a second-order ordinary differential equation of T with eigenvalue v_s. Under quasi-steady-state conditions, there is no heat transferred to the interior of the droplet since the entire droplet temperature is taken as constant. Therefore, the energy balance at the interface provides a Neumann left boundary condition for the above differential equation:

$$\left(\frac{dT}{dr}\right)_s = \frac{\rho v_s L_t}{\lambda}. \tag{5.4}$$

The remaining Dirichlet left boundary condition can be imposed as a known T_s. Although T_s is formally an unknown, at quasi-steady-state conditions (which implies long time scales) and sufficiently high gas temperatures, it can be assumed to be very close to the boiling point temperature of the liquid, $T_s = T_b$. For the right boundary conditions, $T_{r\to\infty} = T_\infty$ and $(dT/dr)_{r\to\infty} = 0$ can also be imposed. Solving (5.3) with the above boundary conditions leads to

$$T_\infty - T_b = \frac{L_t}{C_p}\left(e^{v_s r_s/\alpha} - 1\right),$$

which may be expressed in a more convenient, non-dimensional form:

$$\frac{v_s r_s}{\alpha} = \ln\left(1 + \frac{C_p(T_\infty - T_b)}{L_t}\right). \tag{5.5}$$

Equation (5.5) gives the well-known result that at quasi-steady-state conditions, the mass flow at the interface due to evaporation is linearly proportional to the droplet radius [14]:

$$\dot{m} = 4\pi r_s^2 \rho v_s = \left(\frac{4\pi\lambda r_s}{C_p}\right) \ln\left(1 + \frac{C_p(T_\infty - T_b)}{L_t}\right). \tag{5.6}$$

Numerical Results

The numerical calculations were carried out by the method described in chapters 3 and 4, with the exception that constant properties were imposed for the gas and liquid phases and chemistry was turned off in the gas phase. The solution therefore underwent unsteady transients before approaching the quasi-steady-state conditions described above. A convenient measure of time may be defined as the liquid-state Fourier number

$$F_l = \frac{\alpha_l t}{r_s^2}. \tag{5.7}$$

From Equation (5.5)

$$\zeta = \frac{v_s r_s}{\alpha_g}, \tag{5.8}$$

and

$$\theta = \frac{C_p(T_\infty - T_b)}{L_t} \tag{5.9}$$

can be formed. ζ can be considered a nondimensional form of the mass flux out of the liquid-gas interface, and θ can be considered the driving potential for this mass flux:

$$\zeta_{ss} = \ln(1 + \theta), \tag{5.10}$$

where ζ_{ss} is the quasi-steady-state solution for ζ.

A simple configuration for droplet evaporation is to assume that a droplet at initial temperature $T_l(r) = T_i < T_b$ is deployed into a static, constant pressure atmosphere at a temperature $T_g(r) = T_\infty > T_b$. As $F_l \to \infty$, the interface temperature will arrive at a steady-state value near T_b (provided T_∞ is large enough) and gradients in the liquid

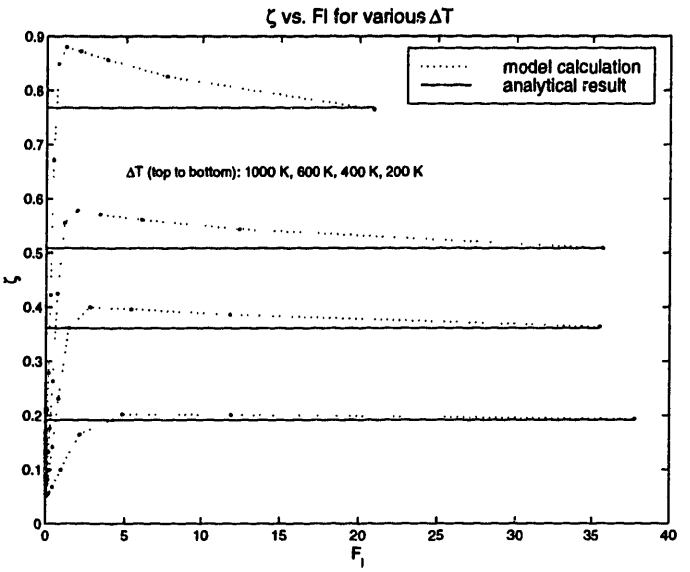


Figure 5-1: Plot of ζ vs. F_l for various ΔT: comparison between analytical and calculated results

phase will vanish. This will lead to the conditions from which the quasi-steady-state solution (Equation (5.10)) was derived.

Figure (5-1) is a plot of calculated results of $\zeta(F_l)$ for various driving potentials θ. Dotted lines represent the results from fully-modeled transient calculations (valid for all F_l), and solid lines represent the result for quasi-steady-state conditions of Equation (5.10) (valid for large F_l only).

If the model is stable and accurate, the transient solutions should approach the quasi-steady-state solutions as $F_l \to \infty$. It is apparent from Figure (5-1) that this is the case. In addition to verifying calculation results, the figure also reveals certain transient aspects of the evaporation process of a droplet.

It shows that at small F_l the evaporation rate is lower than at quasi-steady-state conditions. This occurs because very early in the process a large amount of heat transferred to the droplet due to temperature gradients near the interface is conducted to its interior and used to raise its temperature. As interior temperatures begin to approach the interface temperature T_b, less heat is used for raising interior liquid temperatures, resulting in more heat available for the evaporation process. From the figure, this is seen to occur in the region $1 < F_l < 5$, depending on ΔT.

For small F_l it is expected that temperature gradients (the driving force for evaporation) near the interface are large, and relax to smaller quasi-steady-state gradients as F_l increases. It is therefore expected that the transient solutions approach the steady-state solutions from above. Therefore, once the liquid temperatures have risen sufficiently close to T_b, the evaporation rate overshoots the quasi-steady-state solution.

Evaporation becomes strong enough to account for all the heat transferred from the hot gases (which at this stage is no longer used to raise liquid temperatures) and begins to fall as temperature gradients decrease. Finally, as temperatures in the gas phase for $F_l \to \infty$ approach a steady-state profile, $\zeta(F_l) \to \zeta_{ss}$.

5.2 Experimental Comparison: Droplet Combustion

An alternative form of writing mass conservation for an evaporating droplet is [14]:

$$d((d_s/d_o)^2)/dt = -2\dot{m}/\pi\rho r_s d_o^2, \tag{5.11}$$

where d_s is the droplet diameter at anytime and d_o the initial diameter. Since Equation (5.6) also applies to the case of droplet evaporation under combustion conditions, and it is evident from the equation that $\dot{m} \sim r_s$, the following observation is made

$$K = d((d_s/d_o)^2)/dt = -2\dot{m}/\pi\rho r_s d_o^2 = -\left(\frac{8\lambda}{\rho C_p d_o^2}\right)\ln\left(1+\frac{C_p(T_\infty - T_b)}{L_t}\right), \tag{5.12}$$

where K is constant in time (for quasi-steady-state conditions on which Equation (5.6) depends). This is the "d-squared" law for droplet combustion and is typically used as a measure of quasi-steady evaporation rates for droplets. Figure (5-2) is a plot of calculated (d_s/d_o) vs. time for a droplet of initial diameter $d_o = 1.0$ mm, in

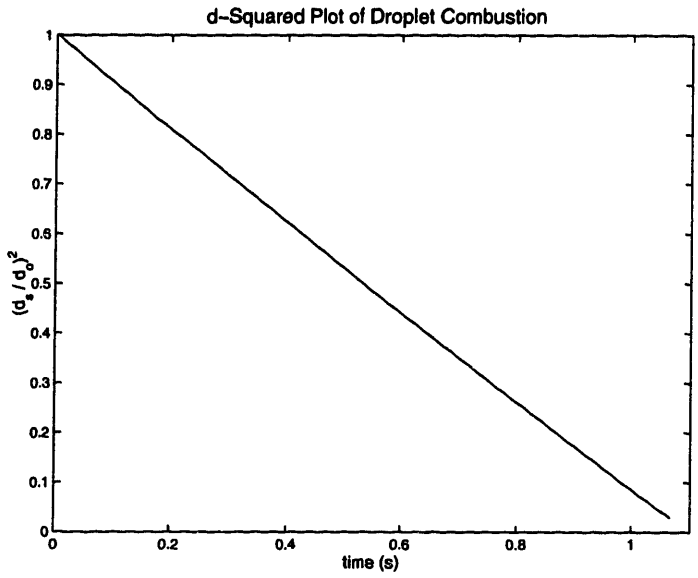


Figure 5-2: Calculated "d-squared" plot of methanol droplet combustion; original droplet diameter of 1.0 mm, $p = 1$ atm, ambient gases at $T_a = 300$ K. From the slope of the line, $K = 0.9$/s; experimental results by Cho show $K \sim 0.65$-0.8/s

a quiescent environment of pressure $p = 1$ atm and gas temperature of $T_{a\to\infty} = 300$ K. It clearly shows that quasi-steady-state conditions dominate most of the process since (d_s/d_o) is linear for $t > 0.01$ s. This has been widely observed in experiments [14, 15, 6] for droplets of this diameter. From the graph, it is possible to obtain the value of $K = 0.9$.

Available experimental results of Cho [15] for droplets in micro-gravity environments have produce values of K which vary between 0.65/s and 0.8/s for the same conditions as those of the calculation. This deviation from the expected d-squared law behavior has been observed in past studies [15, 6] and is mostly attributed to other possible effects at the droplet surface. In particular, condensation of water (which has not been modeled here) decreases the amount of available heat to drive evaporation slowing down the process significantly at large t (decreasing K). Additional error may be caused by inaccuracies in the reaction data and chemical kinetic mechanism. The fact that the d-squared law has been clearly observed in the figure and that the value of K is close to experimental values imply that the model is successful in capturing the behavior and time scales of evaporation and oxidation of liquid methanol.

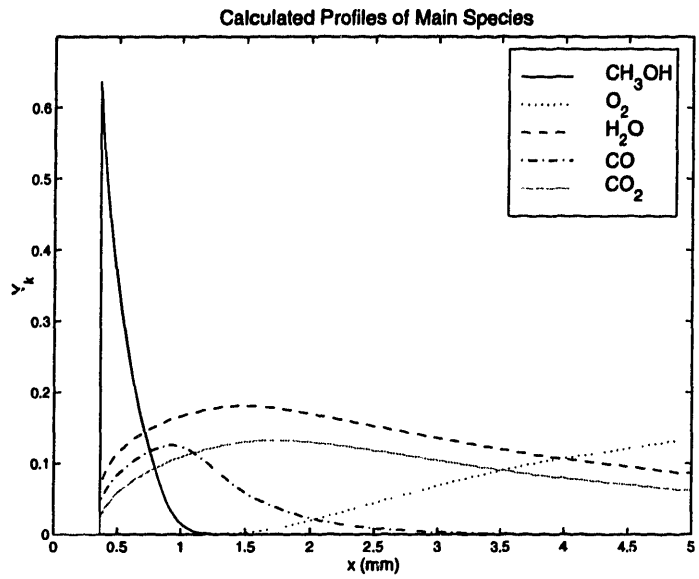


Figure 5-3: Major species profiles for quasi-steady methanol droplet combustion. Results from above calculations at $t = 0.5$ s

Figures (5-3) and (5-4) show calculated results that are not available from experiments. These are the major and radical species concentration profiles near the liquid droplet at time $t = 0.5$ s for the calculations above. From Figure (5-2) it is clear that at this time quasi-steady conditions have been attained. The figures show that for these conditions, a diffusion flame exists at approximately 0.85 mm from the liquid-gas interface, which has a radius of 0.4 mm. They also show that at 0.5 s, the mass fraction at the interface is roughly $Y_{f,e} = 0.64$ and decreases rapidly to zero at the flame. This provides the driving force for evaporation and the decrease of the droplet diameter shown in figure (5-2). The value of $Y_{f,e}$ will be shown to play a crucial role in the results presented in the next chapter.

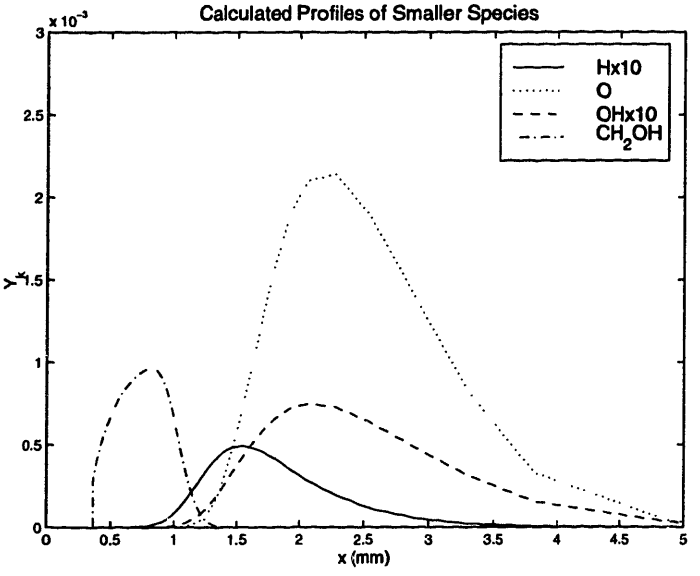


Figure 5-4: Radical and intermediate species profiles for quasi-steady methanol droplet combustion. Results from above calculations at $t = 0.5$ s

Chapter 6

Calculation Results

6.1 Fuel Type: Methanol

A baseline liquid fuel was used to study the effects of different parameters on the evaporation and oxidation of a thin liquid layer on a smooth surface. Methanol (CH_3OH) was chosen due to its relatively small chemical kinetic mechanism. It has been studied extensively and several mechanisms are available for a variety of calculations. Since the present work is concerned with flame calculations, a model that has been developed and verified for such a configuration is desirable. This work uses the chemical kinetic mechanism of Chen [16], which includes 15 species and 23 reactions, and has been developed for diffusion flames. It should be noted that the model easily accepts any fuel for which chemical kinetic, transport, and thermodynamic data is available.

Liquid-phase thermodynamic properties for pure methanol were obtained from well established values in the literature [17]. The vapor pressure curve was calculated from experimentally obtained coefficients for the Antoine equation [12], but can also be determined form fundamental thermodynamic consideration of the problem [18]. This has been pursued in section 6.9 in order to study the effect of certain key properties that control the evaporation process.

This chapter presents all the relevant results obtained for calculations regarding a liquid layer exposed to flames under a variety of conditions. The aim of the chap-

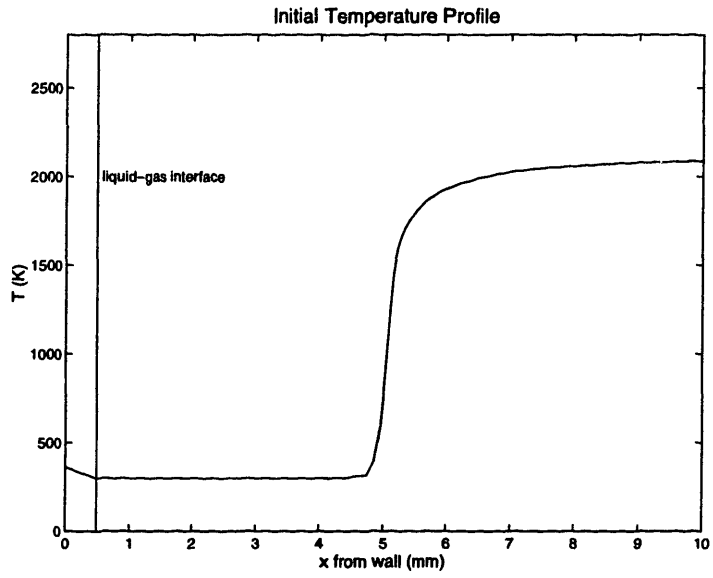


Figure 6-1: Initial temperature profile for constant pressure calculation at 1 atm

ter is to describe underlying heat and mass transfer mechanisms that affect total evaporation and subsequent oxidation of a liquid layer of a given initial thickness.

6.2 Liquid Fuel Evaporation and Oxidation due to Constant Pressure Flames

This section presents results from calculations of a flame approaching a fuel liquid layer and the resulting evaporation and condensation under constant pressure conditions. Figure (6-1) is a plot of the initial temperature profile for a 1 atmosphere calculation. Initial species and temperature conditions were imposed so that the boundary layer had not yet been formed initially. It will be shown that this has a small impact on the solution. The initial flame location for different pressures was arbitrarily imposed so that the flame would reach the interface at different times in order to facilitate viewing of the plots. Initial time is therefore an arbitrary measure, and since pressure is constant during the process, only relative times are important.

The unburned gas equivalence ratios for these calculations is $\phi = 0.9$. The wall is held at a fixed temperature of $T_w = 361$ K and is covered by a liquid layer of pure

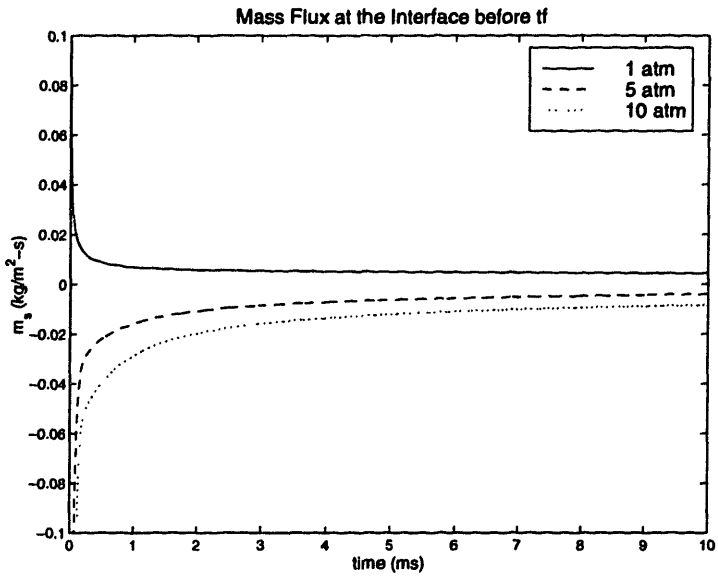


Figure 6-2: Mass flux $\dot{m}_s$ from a methanol liquid layer due to evaporation before flame reaches the liquid-gas interface

methanol. The initial temperature of the liquid layer is distributed linearly from the wall temperature at $x = 0$ mm to the unburned gas temperature of $T_g = 300$ K at $x = 0.5$ mm (the location of the liquid-gas interface). Calculations were performed with constant pressures of 1 atm, 5 atm, and 10 atm. Figure (6-2) shows the mass flux from the liquid layer due to evaporation for the three cases before the flame reaches the liquid-gas interface.

The formation of the boundary layer occurs quickly for these cases (before the flame reaches the interface which will be denoted as t_f). In the figure, the initial transients (for times $t < 2$ ms) are due to the development of species boundary layers. Initially, the mass flux will be positive or negative for evaporation or condensation, respectively, that may occur due to steep initial species gradients near the interface for thin, undeveloped boundary layers. As time progresses and boundary layer thicknesses increase, the mass fluxes approach much smaller, steady state values.

It is apparent from the figure that the formation of the boundary layer is different for the 1-atm versus the 5- and 10-atm calculations. In the 1-atm case, the mass flux approaches a positive steady-state value from above, whereas in the latter two cases it approaches a negative steady-state value from below. This is due to the

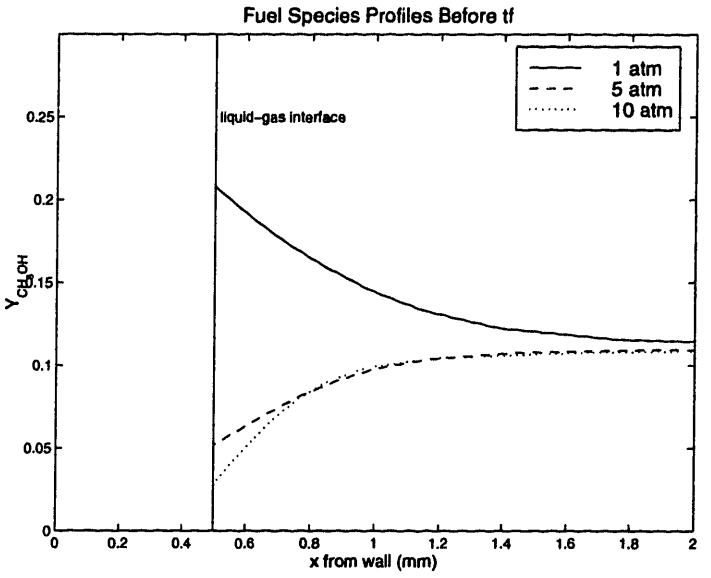


Figure 6-3: Developed species boundary layers before flame reaches the interface: $(\partial Y_f/\partial r)_s < 0$ for 1-atm case and $(\partial Y_f/\partial r)_s > 0$ for 5- and 10- atm cases

fact that at 1 atm (and calculated temperatures of approximately 300 K), the liquid-gas phase-equilibrium concentration (from Equation (3.21)) of $Y_{f,e} = 0.21$ is higher than the fuel concentration in the gas phase for $\phi = 0.9$, $Y_{f,\phi} = 0.12$. Therefore, a species boundary layer will form where $(\partial Y_f/\partial r)_s < 0$ for $t < t_f$, resulting in diffusion away from the interface and forcing evaporation to occur in order to maintain Y_f at the phase-equilibrium value. As the boundary layer develops, $|(\partial Y_f/\partial r)_s|$ becomes smaller and the evaporation mass flux approaches a smaller steady state value.

For the 5- and 10-atm cases, however, the phase-equilibrium concentration is lower than fuel concentration in the gas phase (which has not changed); namely, $Y_{f,e} = 0.05$ and $Y_{f,e} = 0.03$, respectively, versus $Y_{f,\phi} = 0.12$. The boundary layer will therefore be characterized by $(\partial Y_f/\partial r)_s > 0$ for $t < t_f$, resulting in diffusion toward the interface which requires condensation in order to maintain phase-equilibrium. Thus, the mass flux will start out negative and approach a small negative value. The fuel species profiles are shown in Figure (6-3) which illustrates the characteristics of the boundary layers described above.

Figure (6-4) shows the mass flux out of the liquid layer for the entire calculation, including $t > t_f$. From the figure, it is evident that each case has a spike in the

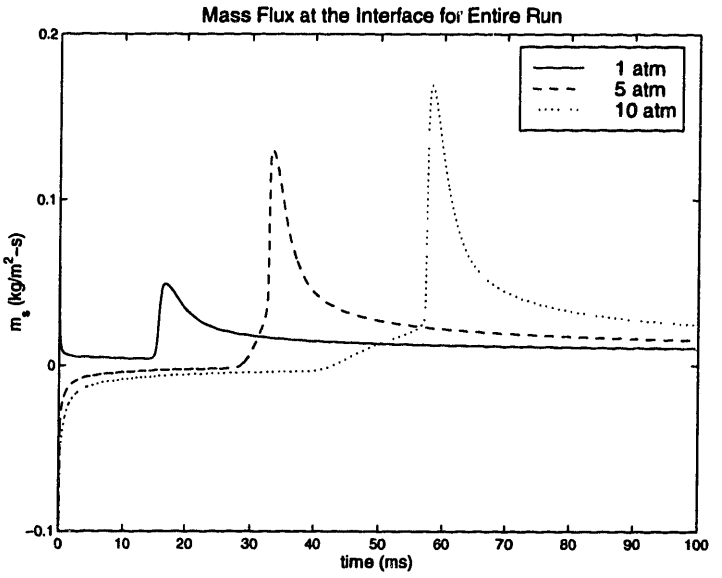


Figure 6-4: mass flux $\dot{m}_s$ from a methanol liquid layer due to evaporation for entire calculation

mass flux $\dot{m}_s$ out of the liquid layer. This reflects the instant $\sim t_f$ when the flame reaches the layer and transfers a large amount of heat to the interface, which drives evaporation. Due to arbitrary initial positioning of the flame, the 1-, 5-, and 10-atm cases have t_f of approximately 18 ms, 35 ms, and 60 ms, respectively.

The figure shows that high pressures will result in higher levels of evaporation. Two factors may account for this observation: the thermal environments and the species distributions of the different cases. Figure (6-5) is a plot of temperature profiles for the three cases at arbitrary times before the flame reaches the interface. Although the gradients at the flame become progressively steeper for higher pressures, the burned gas temperatures are not significantly different, and are unlikely to account for the large difference in mass fluxes for the different cases.

Although there is little difference in the thermal environment before the flame reaches the interface, the species profiles are significantly different. Figures (6-3) and (6-6) show that the low pressure case is characterized by an abundance of fuel and lack of oxygen near the interface due to evaporation for $t < t_f$. For the higher pressures, the situation is reversed due to condensation for $t < t_f$.

As the flame approaches the liquid layer, the heat transfer will drive evaporation

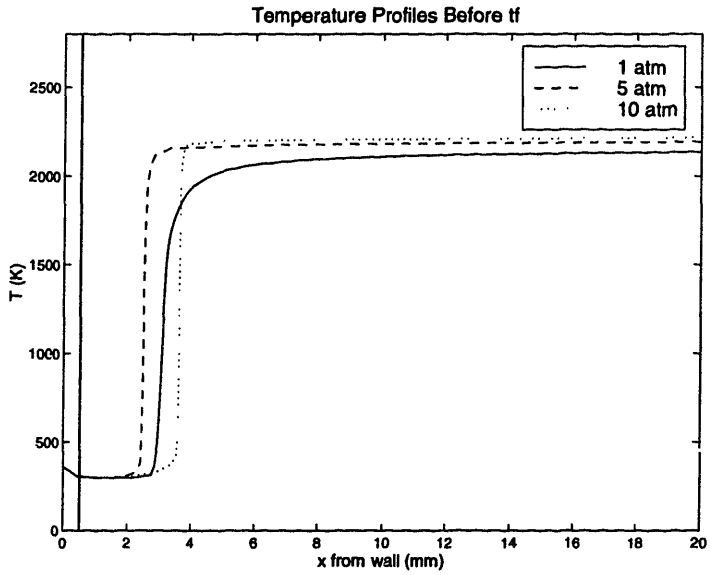


Figure 6-5: Temperature profiles before flame reaches the liquid layer for 1-, 5-, and 10-atm runs

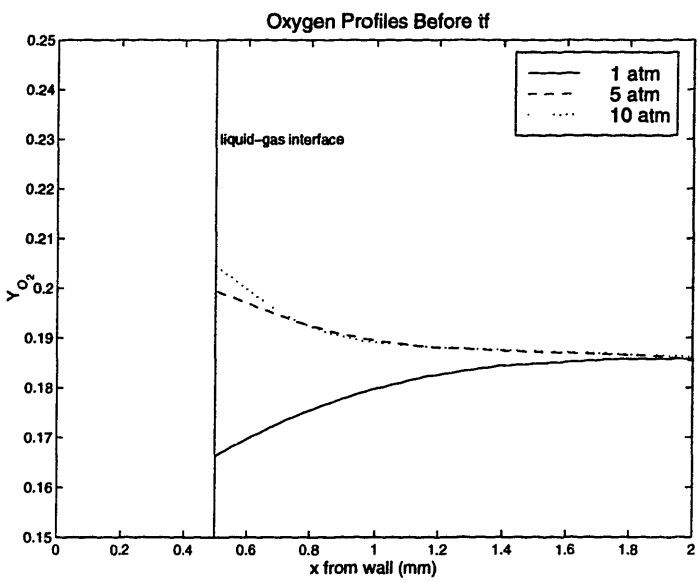


Figure 6-6: Oxygen profiles before the flame reaches the liquid layer for 1-, 5-, and 10-atm runs

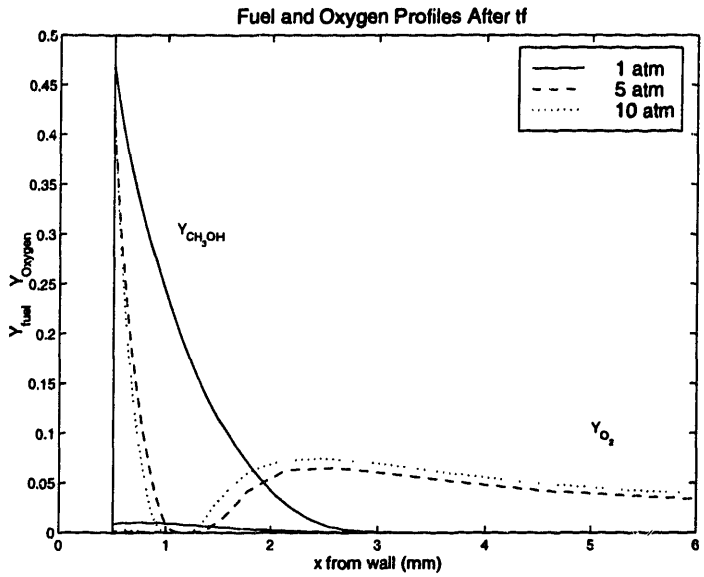


Figure 6-7: Species profiles shortly after flame reaches the liquid layer for 1-, 5-, and 10-atm runs

which introduces fuel to the hot gases. Since the low pressure case is rich near the wall, the small amount of remaining oxygen is quickly consumed by evaporated fuel. For the high pressure cases, however, the lean gases near the wall are composed of a sufficiently large amount of oxygen to allow for a short-lived diffusion "flame" to exist near the interface. As fuel is evaporated into the hot gases, oxygen is available from the lean environment that exists near the interface before t_f to sustain this flame. Figure (6-7) shows the fuel and oxygen distribution shortly after t_f for the three different pressures.

The diffusion flame created by the process described above serves as a source of heat and a sink of fuel near the interface, both contributing as forcing potentials for evaporation and accounting for the differences observed in Figure (6-4). Higher pressures will have lower phase-equilibrium concentrations before t_f which will form leaner environments near the interface. Therefore, as pressure increases, more oxygen is available to sustain diffusion flames near the wall for larger amounts of time and more evaporation will occur.

Figure (6-8) shows the heat transfer into the liquid layer at the interface and illustrates the above observations. For high pressure cases, sharp spikes in heat transfer

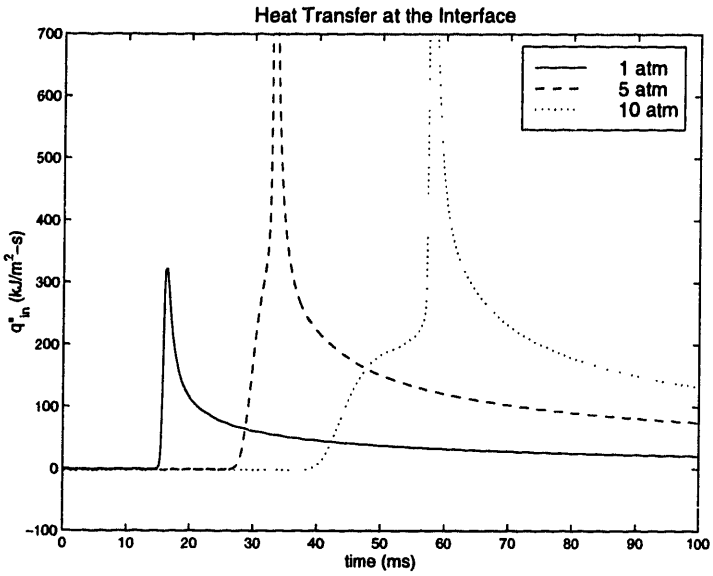


Figure 6-8: Heat transfer into liquid layer for 1-, 5-, and 10-atm runs

into the liquid layer occur shortly after the flame reaches the interface. They are due to the existence of the diffusion flame near the wall, which, as is shown by the figure, serves as a strong heat source for evaporation. The temperature profiles shortly after t_f in Figure (6-9) show that the existence of the diffusion flame results in a large temperature gradient near the interface, which provides the large amount of heat observed.

Finally, Figure (6-10) shows the impact of the processes described above on the thickness of the liquid layer, represented by the position of the liquid-gas interface, x_l. Condensation is seen to occur for the high pressure cases before t_f. For all cases, once the flame reaches the wall evaporation consumes the liquid layer, resulting in a reduction of x_l. As expected, the rate of evaporation for the higher pressure cases is greater than that for lower pressures, mainly due to the existence of the diffusion flame. For the first ten milliseconds after flame arrival, for example, the 1-atm case evaporates the liquid layer at a rate of approximately 0.04 mm/s. The 10-atm case, in contrast, evaporates at a rate of 0.15 mm/s.

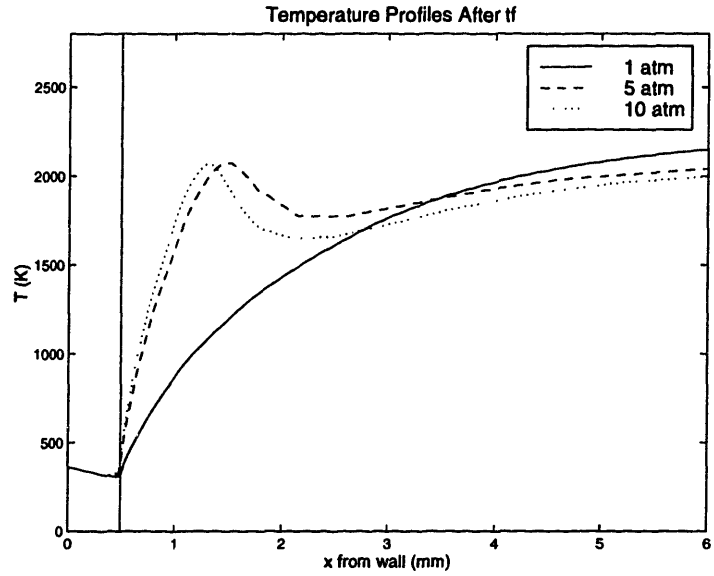


Figure 6-9: Temperature profiles after flame reaches the liquid layer for 1-, 5-, and 10-atm runs

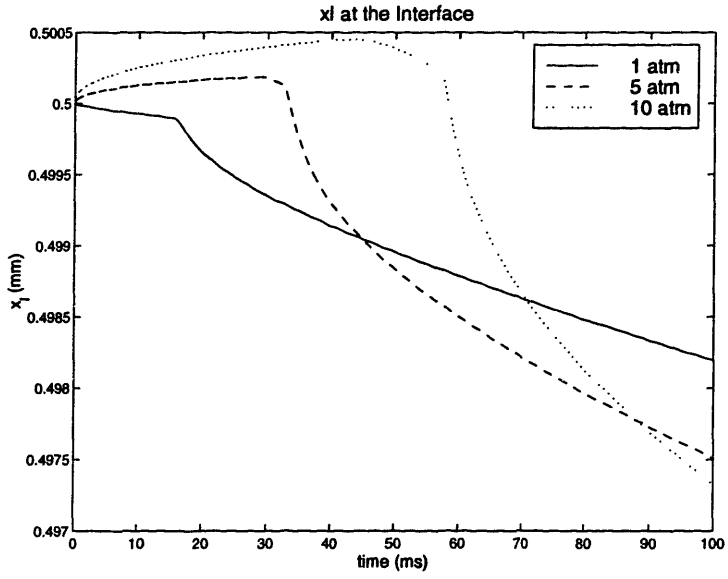


Figure 6-10: Liquid layer thicknesses for 1-, 5-, and 10-atm runs

6.3 Liquid Fuel Evaporation with Varying Pressures and Gas Temperatures

Gases in reciprocating combustion devices rarely operate at constant pressures. In internal combustion engines, for example, a fixed amount of gas is compressed and ignited. This project is concerned with the mechanisms driving evaporation and oxidation at the wall of such devices.

It is therefore desirable to simulate the conditions that lead to evaporation and oxidation of a thin liquid layer deposited on a cool wall under similar pressure and temperature conditions to those found in actual engines. A pressure history for a single cycle is necessary as input, and can either be acquired experimentally or determined from calculations. Here, the pressure history is obtained from a zero-dimensional spark-ignition engine simulation by Poulos and Heywood [9]. Table (6.1) displays general engine specifications and operating conditions used as input in the simulation.

fuel type	methanol
equivalence ratio (ϕ)	0.9
engine speed	1500 rpm
wall temperature (T_w)	361 K
compression ratio	9.0
gross indicated mean effective pressure (GIMEP)	3.75 bar
engine bore	8.20 cm
engine stroke	8.80 cm
spark timing	MBT

Table 6.1: Engine specifications and operating conditions for simulated baseline case

In this section, only processes preceding the moment at which the flame reaches the wall are considered ($t < t_f$) for a liquid layer of initial thickness $x_{lo} = 0.2$ mm. A boundary layer forms near the wall before and during compression as unburned core gases enter the combustion chamber and are compressed adiabatically. Since this model only takes into account the processes that occur at very short distances from the wall, it is not necessary to model the initial stages of the flame kernel

development. The flame can be inserted into the solution domain such that it reaches the wall near peak pressure, which assumes near complete combustion of cylinder gases at this point. To accomplish this for the baseline case, the flame is introduced in the computation domain at a distance of 3.0 mm from liquid layer at 366 crank angle degrees (CAD) after top dead center of intake, or 6 degrees after top dead center of compression. The flame arrives at the liquid layer at approximately 377 CAD (which defines t_f), roughly the point of peak pressure. This section is only concerned with evaporation and condensation before this point. The next section addresses the processes associated with t in the vicinity of, and greater than, t_f.

Figure (6-11) shows the portion of the cycle discussed in this section. This engine cycle, operating at maximum break torque spark timing (MBT), is characterized by intake valve closing (IVC) at 241 CAD and spark at 344 CAD. Calculations were performed beginning at 200 CAD with step profiles for species and temperature at the liquid-gas interface and it was observed that thermal and species boundary layers were very nearly fully developed by IVC. At IVC, the pressure inside the combustion chamber is 0.51 atm. The boundary layer obtained from the calculations starting at 200 CAD and ending at IVC was used to impose initial conditions. The time interval for these calculations (from 241 CAD to 366 CAD) is of 13.9 ms, with spark occuring at 11.4 ms. In this section, time is referred to in relation to IVC (241 CAD).

As gases in the cylinder are compressed and combusted, the rise in pressure results in increasing temperatures of unburned gases near the liquid layer. Increasing gas temperatures in the boundary layer force liquid-gas interface temperatures, T_l, to rise as a result. As described in chapter 3, the evaporation process is controlled by heat transfer into the interface and the resulting effect on the required species mass fraction for phase-equilibrium:

$$X_{k,s} = \frac{10^{A_k - \frac{B_k}{T_l + C_k}}}{P}. \tag{6.1}$$

The inverse pressure dependence was observed in the last section for constant pressures. Under engine operation conditions, however, the pressure and temperature

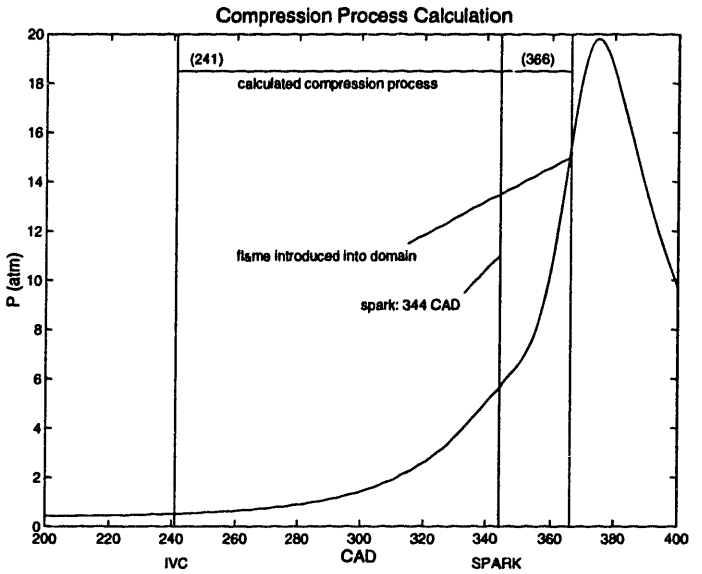


Figure 6-11: Compression process calculations

increase simultaneously during the processes, affecting equation (6.1) in opposite directions. Figure (6-12) is a plot of input P and calculated T_l prior to flame arrival for a 0.2 mm liquid layer. It shows that cylinder pressure and liquid surface temperature increase from 0.5 atm to 16.5 atm and from 308. K to 336. K, respectively. The goal of this section is to determine the relative effects of pressure and temperature on the overall evaporation or condensation of fuel from the liquid layer prior to flame arrival.

According to equation (6.1), increasing liquid surface temperatures promotes evaporation, while increasing pressure promotes condensation. Figure (6-13) is a plot of the evaporation mass flux and liquid layer thickness versus time. It shows that evaporation occurs early in the process due to the existing boundary layer similar to that of the 1-atm case of section (6.2). As the pressure begins to rise, however, $\dot{m}_s$ decreases and condensation begins to occur at approximately 2.7 ms (250 CAD). The liquid layer thickness, x_l, begins to increase for 2.7 ms $< t < t_f$. It is clear that, for the pressure and temperature rises shown in Figure (6-12), the effect of pressure dominates.

Figure (6-14) explicitly shows the species mass fraction decrease during this portion of the cycle. As stated earlier, the species mass fraction is a driving potential for evaporation or condensation. Initially, at $t = 0$ ms, $T_l = 308.$ K, and $P = 0.51$ atm,

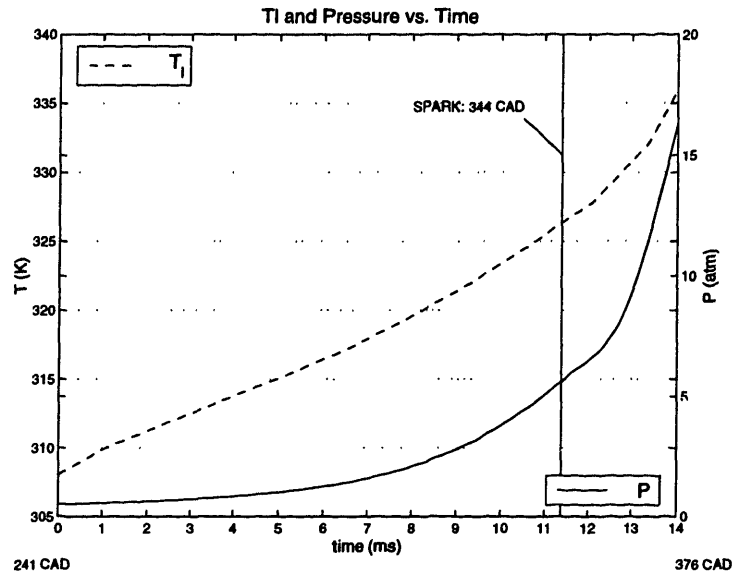


Figure 6-12: Liquid-gas interface temperature T_l and Pressure versus time for compression calculations

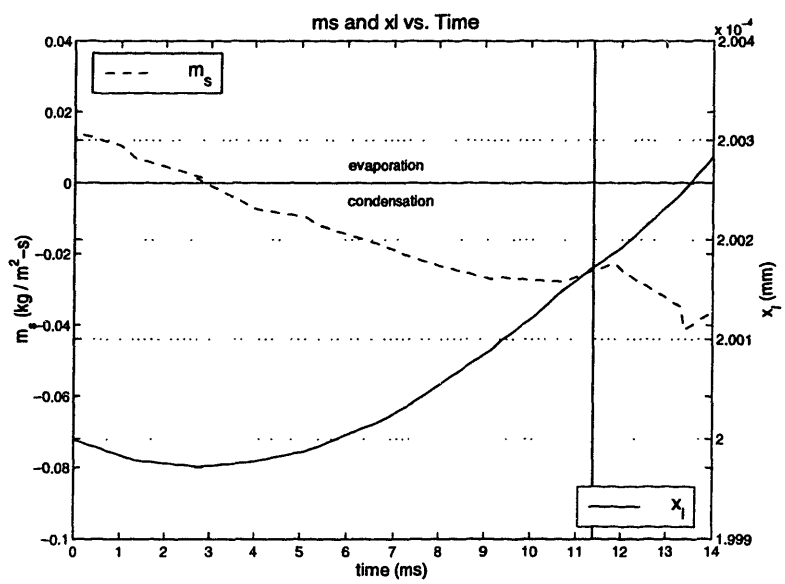


Figure 6-13: Evaporation mass flux and liquid layer thickness versus time for compression calculations

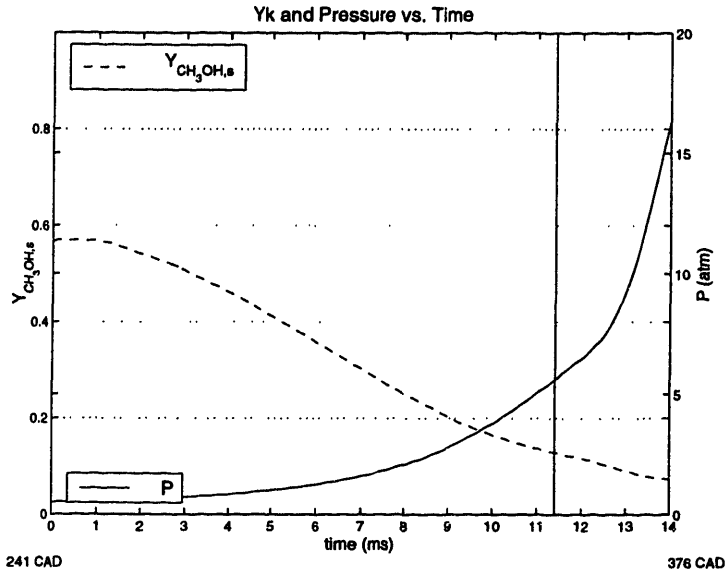


Figure 6-14: Evaporation mass flux and liquid layer thickness versus time for compression calculations

the fuel phase-equilibrium mass fraction is $Y_{f,e} = 0.56$. Since it is greater than the gas mass fraction of the core gases at $\phi = 0.9$, $Y_{f,\phi} = 0.12$, the resulting developed species boundary layer, characterized by $(\partial Y_f/\partial r)_s < 0$, requires evaporation. As pressure increases, this value drops significantly, attaining a value of $Y_{f,s} = 0.08$ near t_f. This value is below $Y_{f,e}$ and thus condensation is required. Figure (6-13) shows, however, that condensation occurs at approximately 2.7 ms, when the equilibrium mass fraction is approximately 0.5. Although this value is larger than the gas mass fraction, a boundary layer has not yet developed from conditions at $t = 0$. At $t \approx 2.7$ ms, $(\partial Y_f/\partial r)_s \approx 0$, after which point $(\partial Y_f/\partial r)_s > 0$ for 2.7 ms $< t < t_f$, requiring condensation. Figure (6-15) shows the species profiles at different times during this process.

In the time between IVC and t_f, the amount of condensation that occurs increases the liquid layer thickness by 0.3 μm (Figure (6-13)). In the next section, this increase will be shown to be negligible compared to amount of evaporation that results due to flame arrival.

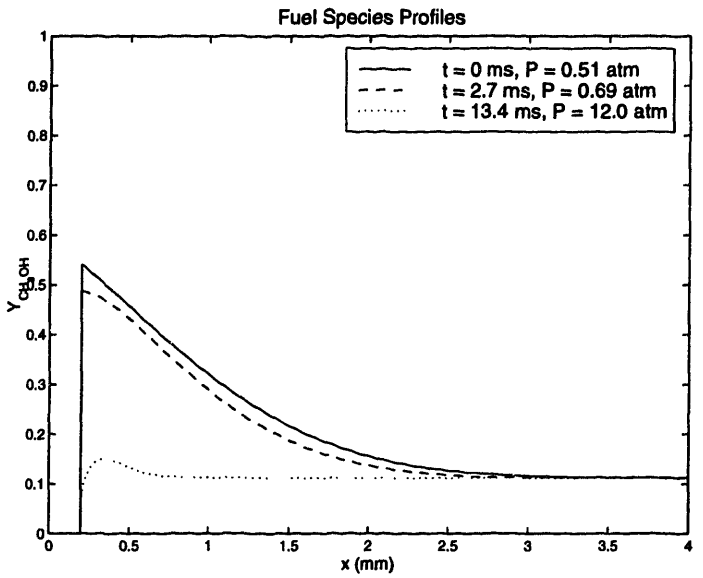


Figure 6-15: Fuel species profiles for different times during calculation; at $t \approx 2.7$ ms, $(\partial Y_f/\partial r)_s$ goes from negative to positive, resulting in condensation

6.4 Liquid Fuel Evaporation and Oxidation with Varying Pressure

This section describes the process in which the flame arrives at the liquid-gas interface. As described earlier, the model assumes that the gases in the combustion chamber are completely combusted at peak pressure. Therefore, the flame has been introduced into the solution domain so that it reaches the interface at maximum pressure ($dp/dt \approx 0$). Other than the introduction of the flame, the species and temperature profiles from calculations described in the last section (beginning at 200 CAD and ending at 366 CAD) were used as initial conditions for species and temperature profiles. This section refers to time relative to 366 CAD. Figure (6-16) shows the portion of the cycle addressed in this section. The calculations extend through expansion to the end of the exhaust stroke at 705 CAD, 37.7 ms.

Figures (6.17) and (6.18) show the development of the temperature and fuel species profiles during the calculation. According to the results from the last section, condensation of fuel species into the liquid layer occurs at $t = 0$. The flame reaches the liquid interface at $t_f \approx 2$ ms, at which time heat transfer from the flame causes fuel

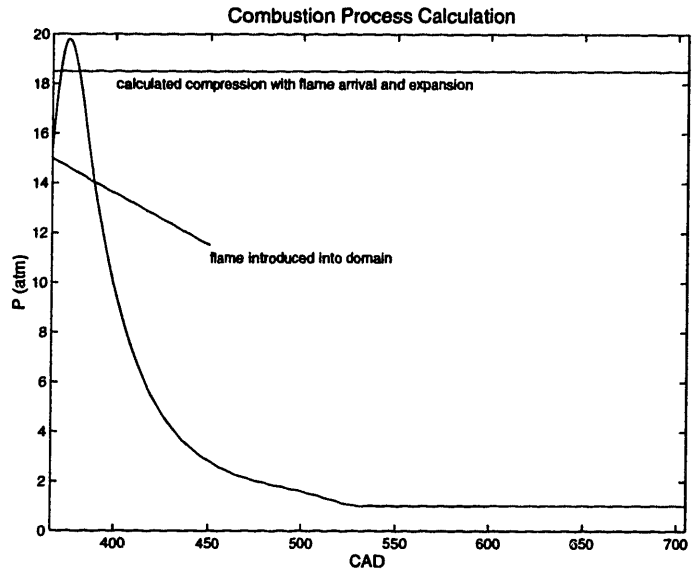


Figure 6-16: Compression with flame arrival and expansion

to evaporate from the liquid layer.

The figures show that at $t = 3.6$ ms, $t = 12.5$ ms, and $t = 37.5$ ms there is a distinctive peak in the temperature profiles. A similar peak was observed to occur in the high pressure cases of section (6.2) (see Figure (6-9)). It was shown that the high pressure cases were characterized by condensation at the interface before flame arrival, which resulted in an abundance of oxygen near the liquid layer during flame approach. This same situation occurs here due to condensation caused by increasing pressure during the cycle. The resulting lean region near the interface is able to support a diffusion flame, which can be defined to exist at the location δ_f where fuel species mass fraction in Figure (6-18) become zero.

Figure (6-17) shows that even for $t > 12.5$ ms, when core gas temperatures become less than 1500 K, oxidation still occurs in the diffusion flame (evident from the shape of the temperature profile) which maintains relatively high temperatures. The diffusion flame still serves as a sink of fuel at this point, although it can also be seen from Figure (6-17) that at this point there is little heat conduction into the liquid layer from the gases since the temperature gradients near the interface decrease significantly after 12.5 ms (a consequence of heat convected by and used for evaporation). As a result, for $t > 12.5$ ms, evaporation occurs mainly due to diffusion of fuel species from the

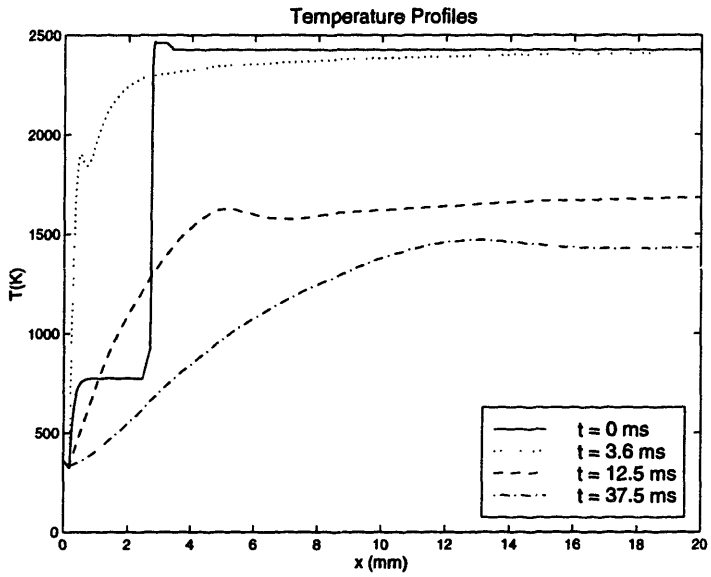


Figure 6-17: Temperature profiles for different times during calculation (t = 0 ms, t = 3.6 ms, t = 12.5 ms, and t = 37.5 ms)

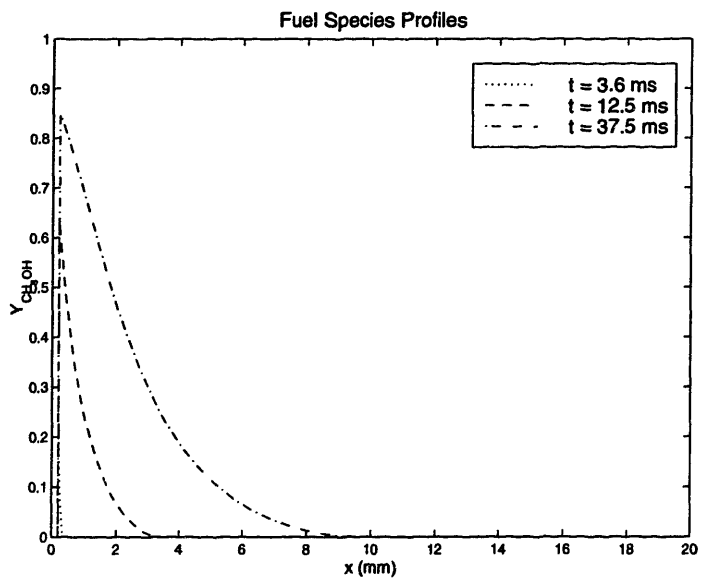


Figure 6-18: Fuel species profiles for different times during calculation

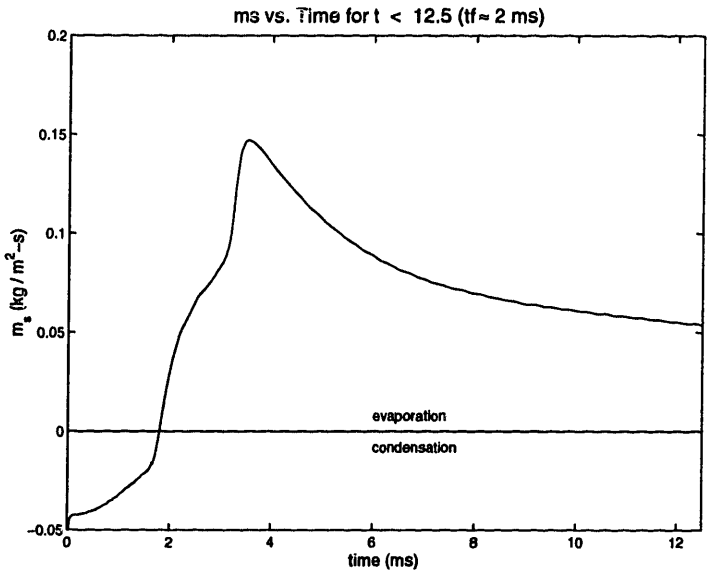


Figure 6-19: Evaporation mass flux for $0 < t < 12.5$ ms

boundary layer to the core gases, which is a much slower, quasi-steady process.

Figure (6-19) shows the evaporation mass flux from the liquid layer for $0 < t < 12.5$ ms. The figure shows that $\dot{m}_s$ peaks at 3.6 ms, which is after the initial flame arrival ($t_f \approx 2$ ms). This is when the diffusion flame is first established and is closest to the liquid layer. The value of $\dot{m}_s$ drops after this point as the flame moves away from the interface. From the profiles in Figure (6-18), the 3.6 ms flame is at $\delta_f = 0.6$ mm from the interface, causing a relatively high rate of evaporation of $\dot{m}_s = 0.15$ kg/m^2-s. At 12.5 ms, oxidation has ceased to occur and $\dot{m}_s$ approaches a steady-state value of order of 0.05 kg/m^2-s.

Figure (6-20) is a plot of the heat flux into the interface from the hot gases, $q''_{in} = \lambda_g(\partial T_g/\partial r)_s$, and the amount of heat used for evaporation, $q''_{evap} = \dot{m}_s L_t$ (note that it excludes the amount of heat that is conducted into the liquid layer, which is the difference between the curves shown, $q''_l = q''_{in} - q''_{evap}$). It shows that at approximately 6.5 ms the heat used for evaporation begins to exceed that coming from the hot gases. This is an indication that the lack of fuel near the boundary layer produced by the diffusion flame becomes the main driving force for evaporation at later stages in the cycle, since evaporation is larger than that due only to heat from the hot gases.

The figure also shows that q''_{in} peaks once when the flame arrives at the interface

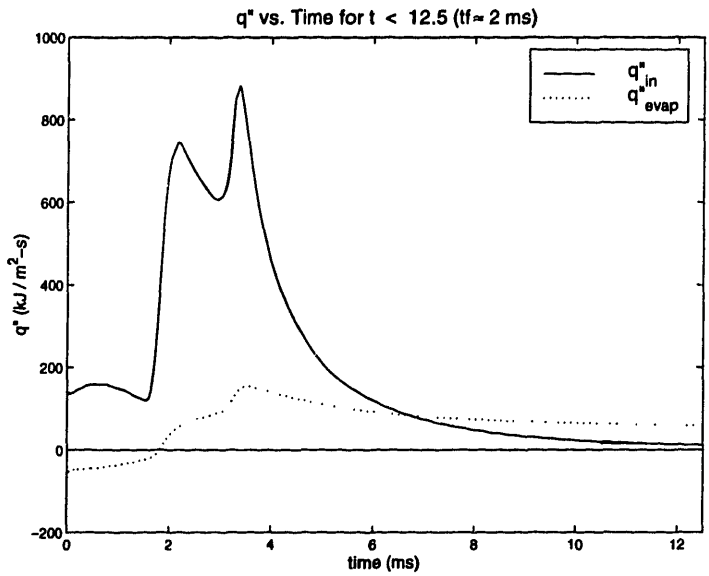


Figure 6-20: Heat fluxes q''_{in} and q''_{evap} at the interface for $0 < t < 12.5$ ms

(at 2 ms) due to the sharp initial sharp temperature gradients. A second peak occurs when the diffusion flame is first established (at 3.6 ms) as the flame releases a large amount of energy which diffused toward the liquid layer and becomes available for evaporation. Figure (6-21) shows that a maximum in interface temperature, T_l, is achieved at approximately 4 ms, due to the diffusion flame.

Figure (6-22) is a plot of the evaporation mass flux and liquid layer thickness for the entire calculation period; from $t = 0$ to $t = 37.3$ ms at the end of expansion (705 CAD). As expected, it shows that $\dot{m}_s$ approaches a small quasi-steady value for larger t. The resulting total change in the liquid layer thickness for this part of the calculation is $\delta x_{l,2} = 2.4$ μm, a factor of 8 times larger than condensation during the compression part of the cycle, $\delta x_{l,1}$. The 0.2 mm liquid layer has therefore experienced a percent reduction during this particular cycle of $\Delta x_l\% = (\delta x_{l,1} + \delta x_{l,2})/x_{l,o}$ =1.2%.

Figure (6-22) also shows a surprising rise in mass flux at approximately 15 ms. This was not observed in the constant pressure case and so cannot be a result of oxidation. It is an artifact of the particular input conditions; namely, the behavior of the pressure history at this time. Figure (6-23) shows the pressure and fuel species equilibrium mass fraction at the interface for this portion of the calculation. It shows that there is a small dip in the pressure history at this point in time, which corresponds

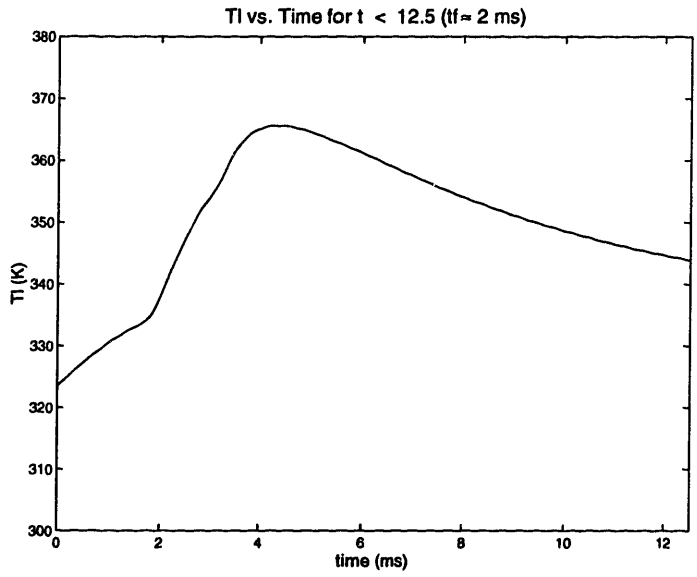


Figure 6-21: Interface temperature for $0 < t < 12.5$ ms

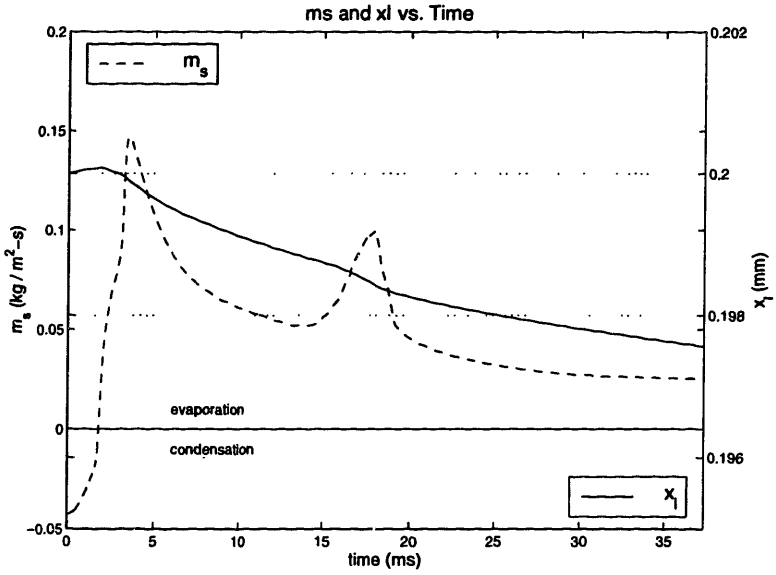


Figure 6-22: Mass flux and liquid layer thickness for entire calculation period, $0 < t <$ 37.7 ms

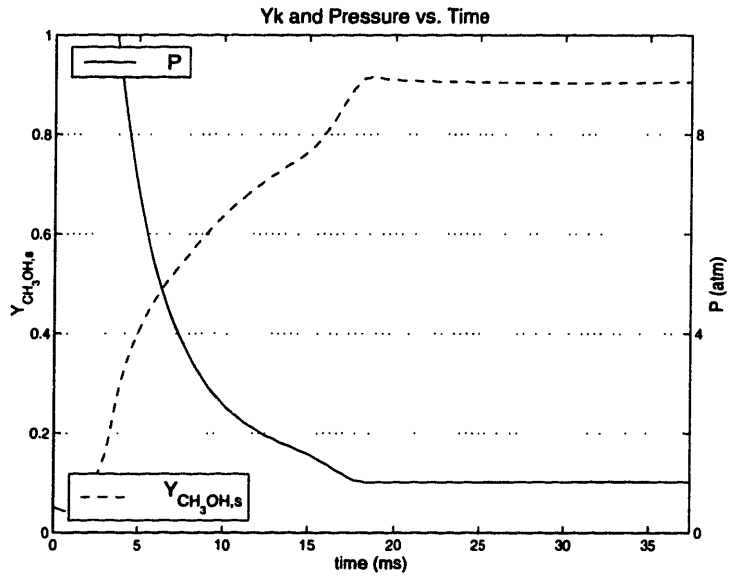


Figure 6-23: Pressure and fuel mass fraction at the interface for entire calculation period

to the opening of the exhaust valve in the simulation (EVO). The resulting pressure disturbance shows that the equilibrium mass fraction of the species at the interface is very sensitive to pressure under these conditions of near boiling point temperatures (at $t = 15$ ms, $P = 1.7$ atm, $T_l = 341.$ K, and $T_b \approx 350.$ K). As shown in the figure, $Y_{f,e}$ locally peaks at this time, which leads to the increase in $\dot{m}_s$ observed in Figure (6-22).

6.5 Unburned Fuel Survival Rate

An important practical application of the present model is the ability to calculate the amount of surviving unburned fuel in the gas phase. During the cycle, fuel is evaporated at the rates shown in Figure (6-22), which consumes liquid fuel and produces gaseous fuel that diffuses into hot gases and becomes available for oxidation. This section presents calculations of the total amount of liquid fuel that evaporates from the liquid phase for a particular cycle and compares it with the remaining fuel in the gas phase at the end of the cycle.

It has already been shown above that the diffusion flame near the liquid layer

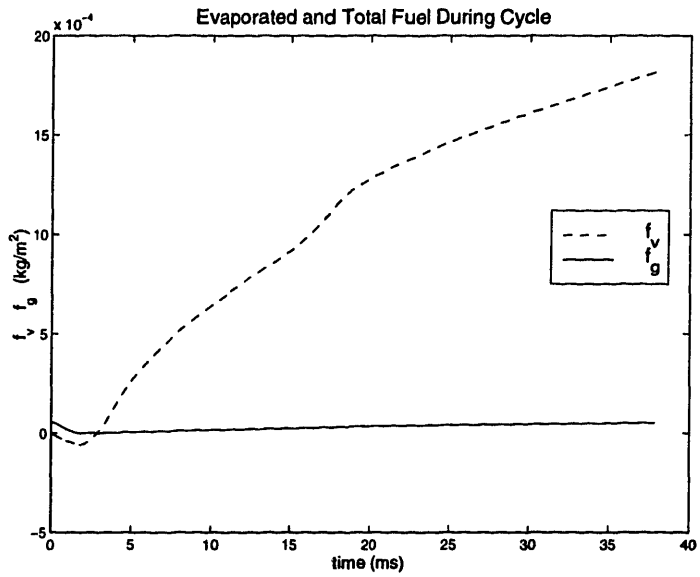


Figure 6-24: Total amount of fuel evaporated during cycle for $x_{lo} = 2$ mm

enhances evaporation by acting as an energy source and species sink. By enhancing evaporation, therefore, the flame acts to increase the amount of fuel in the gaseous phase. However, once the fuel enters the gas phase, the flame acts to decrease its amount by combustion which converts fuel to products. Since this in not a steady-state process, it is necessary to determine which mechanism is faster to predict the fuel survival rate.

The total amount of fuel evaporated from the liquid to the gas domain at any time can be obtained from the known evaporation rate $\dot{m}_s$,

$$f_v(t) = \int_{t_o}^{t} \dot{m}_s(t')\, dt'. \tag{6.2}$$

As noted above, however, this fuel diffuses into the diffusion flame and is combusted. The total amount of fuel present in the gas phase (for flat geometry) at any moment can be obtained from

$$f_g(t) = \int_{r_s}^{r_\infty} \frac{p}{RT} X_f(r', t)\, dr', \tag{6.3}$$

where X_f is the molar fraction of fuel at position r at time t. Figure (6-24) shows values of f_v and f_g during the cycle for a liquid layer of initial thickness of 0.2 mm,

and shows that a large amount of the evaporated fuel oxidizes throughout the cycle. In terms of emissions, it is desirable to obtain the survival rate of evaporated fuel per cycle:

$$f_s = \frac{f_g(\tau)}{f_v(\tau)}, \tag{6.4}$$

where τ is the time corresponding to the end of the cycle. For the case shown above, $f_s = 0.028$/cycle, meaning that at the end of this particular cycle, 2.8% of the evaporated fuel survives as unburned.

6.6 Effects of Liquid Layer Thickness on Evaporation and Oxidation

From the small value of $(\Delta x_l\%)$/cycle observed in the section 6.4, it is evident that several cycles are required to evaporate a liquid layer completely. For decreasing thicknesses, characteristics of the layer, such as total heat capacitance and conduction time scales, change. Therefore it is necessary to calculate the effect of different initial layer thicknesses before predicting the time for complete evaporation. Operating and boundary conditions for these calculations are identical to those in the previous section.

In this section, the evaporation and oxidation of seven initial thicknesses, x_{lo}, were observed: 0.5 mm, 0.4 mm, 0.3 mm, 0.2 mm, 0.1 mm, and 0.05 mm. Since the change in thickness due to condensation in the compression cycle is small compared to that due to evaporation at flame arrival, all the figures shown here represent time with relation to flame arrival, beginning at 366 CAD ($t = 0$) and ending at 705 CAD ($t =$ 37.7 ms), as in the previous section.

Figure (6-25) shows the decrease in liquid layer thickness, $\delta x = (x_l - x_{lo})$, for the different x_{lo} after flame arrival. For smaller x_{lo}, the initial heat capacity of the layer per unit area, $C_l = \int_o^{x_{lo}} \rho_l c_l \, dx$, decreases. Since the amount of heat initially available from the incoming flame is similar for all cases, smaller x_{lo} allow for faster increase

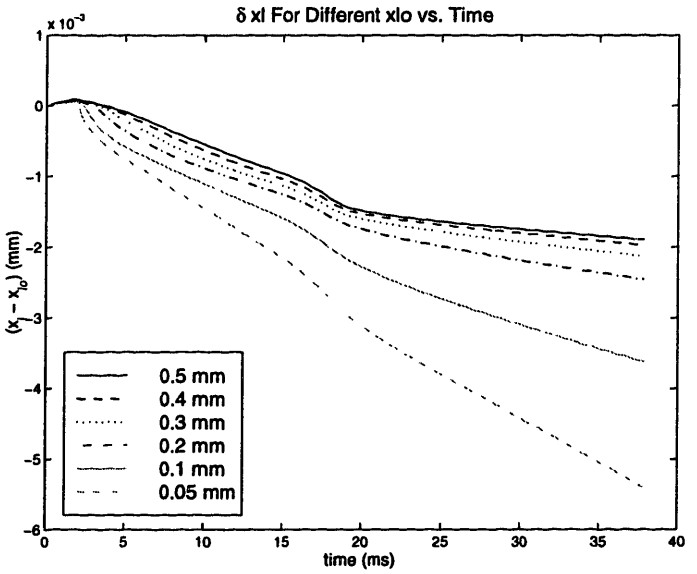


Figure 6-25: Decrease in liquid layer thickness during and after flame arrival for different initial thicknesses, x_{lo}

of interface temperature, T_l, which leads to faster evaporation. Figure (6-26) shows the change in T_l during the calculation for the different cases.

It is evident from the figure that as the flame first approaches the interface, $t < 2$ ms, thin liquid layers will experience significantly higher increases in T_l. According to the observations made in the previous section, the peak in T_l occurs at approximately the time when the diffusion flame is established and is closest to the interface. When this occurs, the smaller heat capacity of the thinner layer allow for larger and faster increases in T_l. The overall results is that for the 0.05 mm liquid layer a maximum temperature of 388 K is observed, compared to 347 K for the 0.5 mm layer.

This increase in T_l will result in large fuel species phase-equilibrium values at the interface, driving evaporation. Figure (6-27) shows the values of $Y_{f,e}$ during the cycle. As $x_{lo} \rightarrow 0$, $Y_{f,e} \rightarrow 1.0$ and $T_l \rightarrow T_b$ during later stages of the cycle.

It is also evident from Figure (6-26) that the maximum T_l occurs sooner for thinner layers. This suggests that the diffusion flame is established sooner for these cases. Figure (6-28) (note different time scales) supports this observation since it shows a second local maximum in heat transferred into the interface, q''_{in}, at earlier times for thinner layers. It also shows that the amount of heat transferred to the interface is

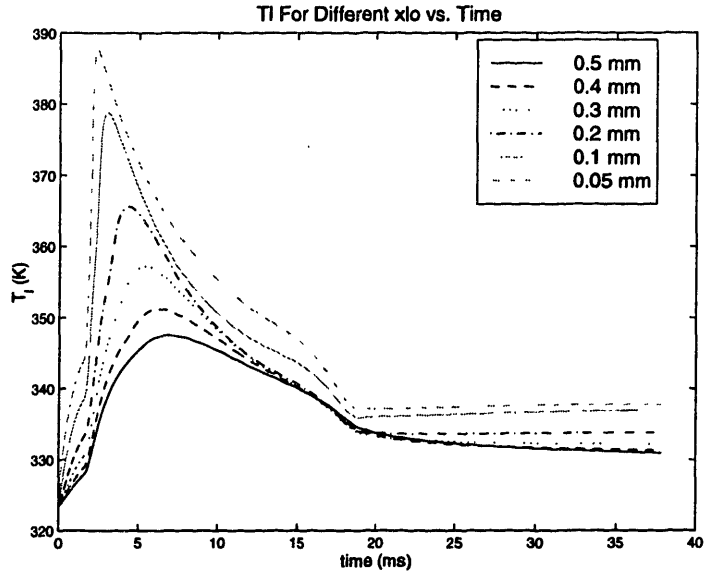


Figure 6-26: Liquid-gas interface temperature T_l during and after flame arrival for different initial thicknesses, x_{lo}

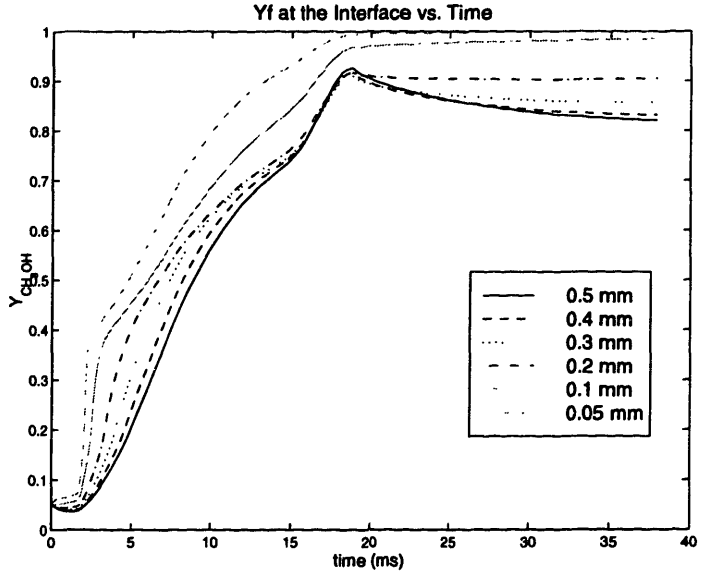


Figure 6-27: Phase-equilibrium mass fraction at the interface during calculation

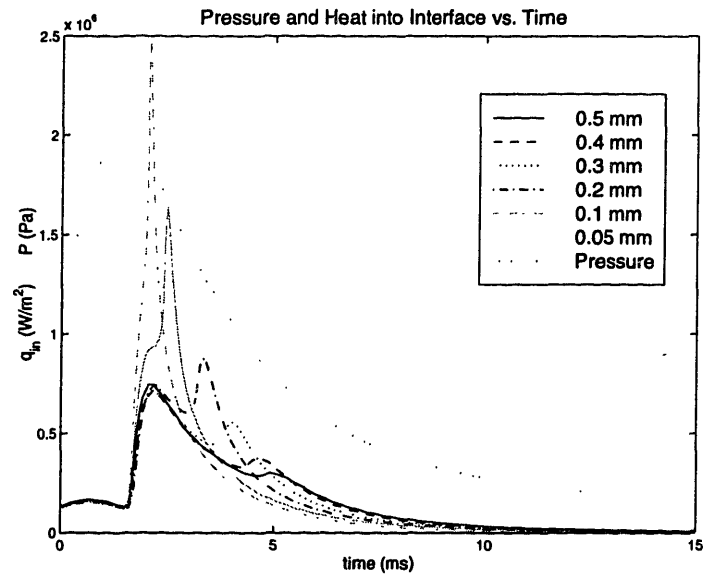


Figure 6-28: Pressure and heat flux q''_{in} into the interface for $0 < t < 12.5$ ms

initially larger for thin layers. In fact, for $\delta x < 0.1$ mm, the second peak becomes indistinguishable from the first; the initial heating of thin layers is sufficient to provide enough evaporation to eliminate the existence of the isolated lean region near the interface, and $\partial q''_{in}/\partial t > 0$ until resulting gradients begin to move away from the interface. This allows the flame to come closer to the interface, resulting in larger q''_{in} and more evaporation.

In order to confirm the observations made above, Figure (6-29) shows the profiles for the 0.05 mm and the 0.5 mm runs at the time that a flame is first discernible; that is, when the fuel and oxygen species first mutually approach zero from different sides. The 0.05 mm case occurs at t =2.1 ms, while the 0.5 mm case occurs at t =4.6 ms. It is also evident from the figure that the thinner liquid layer case produces a flame that is a factor of 3 closer to the interface than is the case for the thick layer. This accounts for the large difference in q''_{in} observed in Figure (6-28).

Figure (6-30) displays the mass flux at the interface during the cycle. As expected from the closer proximity of the resulting flame, initially thin liquid layers allow for the largest amount of evaporation. It is interesting to note that even though Figure (6-28) shows that the amount of heat transferred into the interface drops below that of initially thicker liquid layers for latter stages of the calculation, Figure (6-30) shows

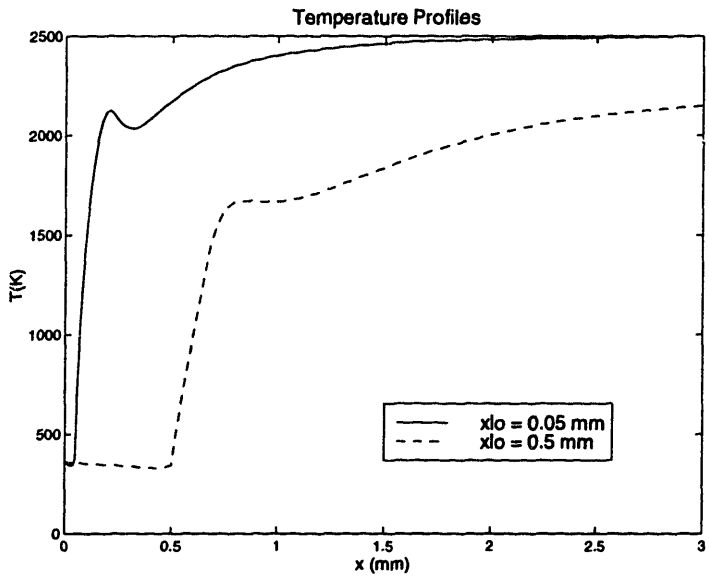


Figure 6-29: Fuel species and oxygen profiles for thick (x_{lo} = 0.5 mm) and thin (x_{lo} = 0.05 mm) liquid layers

that evaporation is still larger for the thin liquid layers. This occurs due to the fact that smaller C_l require less heat to raise the temperature of the surface to large values of T_l, resulting in larger $Y_{f,e}$ and evaporation driven mainly by species flux required to maintain phase-equilibrium.

Figure (6-31) shows the fuel survival rate for the various thicknesses above. It shows that the value of f_s is rather insensitive to initial thickness, indicating that the fraction of surviving fuel is controlled by the existence of the diffusion flame alone. Although thinner layers display faster evaporation, the fuel is convected and diffused into the flame and consumed as in the thicker layers. The fraction of fuel that survives oxidation (roughly 2.9% for all cases) is the fuel contained in the spatial domain outside the liquid layer.

6.7 Model for the Complete Evaporation of a Liquid Layer

Given the data obtained in the last section, it is natural to construct a model for the complete evaporation of a liquid layer under several repeated cycles. Each cycle

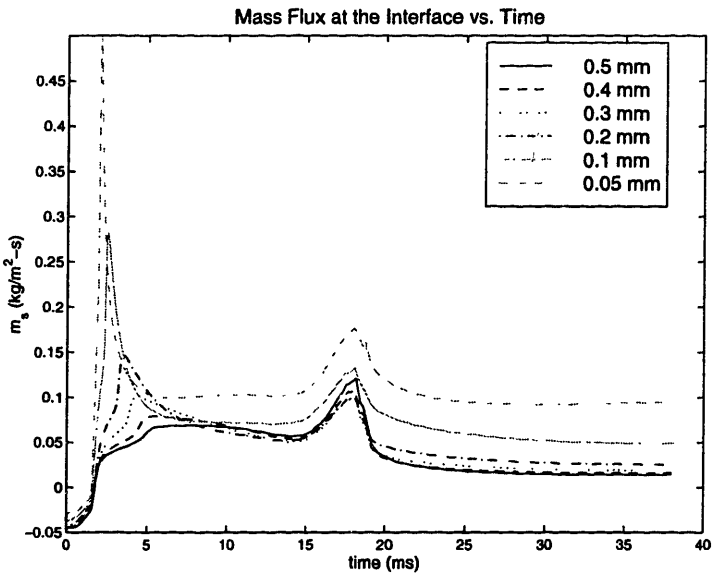


Figure 6-30: Evaporation mass flux from interface

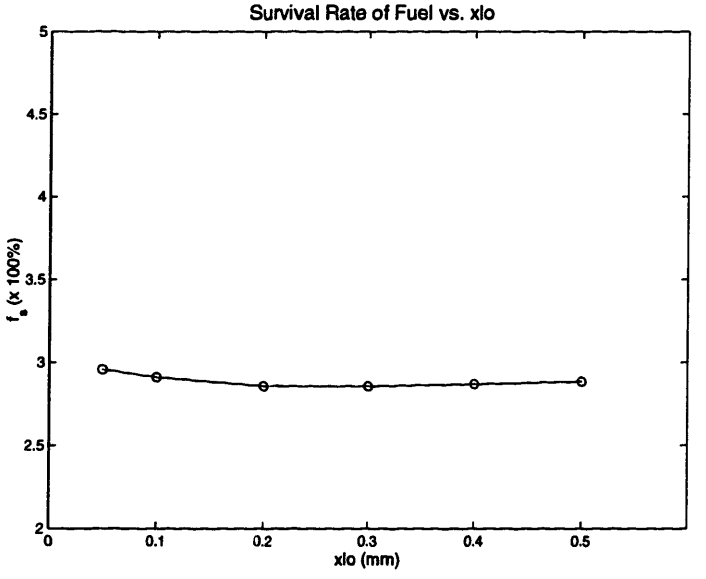


Figure 6-31: Fuel survival rate for different initial thicknesses, x_{lo}

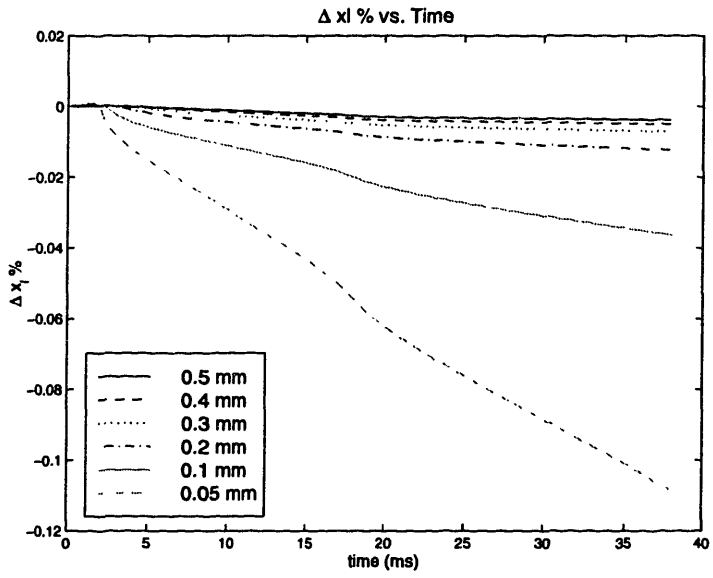


Figure 6-32: $\Delta x_l\%$ during cycle

reduces the liquid layer by an amount $(\Delta x_l\%)$/cycle, which, given all other conditions identical, is only a function of the initial liquid layer thickness for each cycle, x_{lo}. In terms of the situation modeled above, the operating and boundary conditions were held constant during the evaporation period of the layer. It is also possible to apply this model to operating conditions that change during the evaporation process.

Figure (6-32) is a plot of $\Delta x_l\%$ for the thicknesses discussed in the previous section. It clearly illustrates the differences in evaporation for the different cases. By the end of the cycle the 0.5 mm layer is reduced by less than 0.3%, whereas the 0.05 mm is reduced by 10.8%.

The data above can be used to integrate the change of the liquid layer thickness over time. The calculations in this section assume an initial liquid layer thickness of 0.5 mm at cycle #1. Δx_l/cycle (a function of x_{lo} only) is then subtracted from x_{lo} for each cycle as x_{lo} decreases. In order to obtain the necessary data, $(\Delta x_l\%)$/cycle must be known for all $0 < x_{lo} < 0.5$. It is computationally inefficient, however, to calculate the data for every x_{lo} in the solution (given 300 cycles, for example, this would required 300 separate calculations). This problem can be resolved by observing the fact that the liquid layer reduces in a predictable way during its lifetime. Therefore, given enough data points, an interpolation can be made to obtain $(\Delta x_l\%)$/cycle for values of

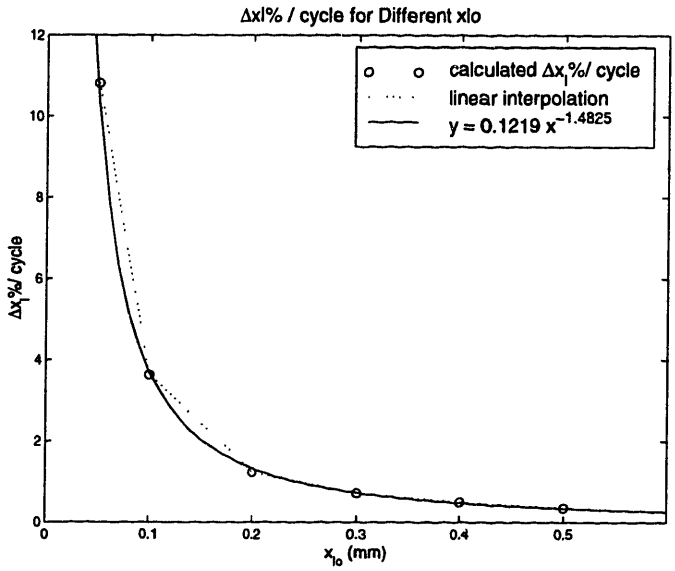


Figure 6-33: $\Delta x_l\%$/cycle for data and power fit

x_{lo} between the available data. Since the procedure consists of essentially integrating $(\Delta x_l\%)$/cycle over x_{lo}, the exact form of the interpolation is not important, provided it has small error for the available data.

Figure (6-33) shows the data obtained from calculations as a plot of $(\Delta x_l\%)$/cycle vs. x_{lo}. A linear interpolation could be used but does not provide a single fit for the entire domain. It was found that a power functions provides good fits for all the cases attempted. For the baseline case with a 0.5-mm liquid layer at cycle #1, the function used is shown in Figure (6-33). The maximum error is of 7.68% with error standard deviation of 0.0473.

Using the results given above, the cycle integration was performed for the baseline case, and the result is shown if Figure (6-34). In order to show the difference between results obtained from the power function fit and linear interpolation, both have been plotted. As shown in Figure (6-33), the linear interpolation will always predict an evaporation rate that is higher than the actual values. Therefore, the linear interpolation will provide the lower bound of evaporation time for a given set of data.

This observation is apparent in Figure (6-34), which shows that the liquid layer completely evaporates after approximately 189 cycles, versus 200 cycles obtained from the power function. For the regions between data points, the power function

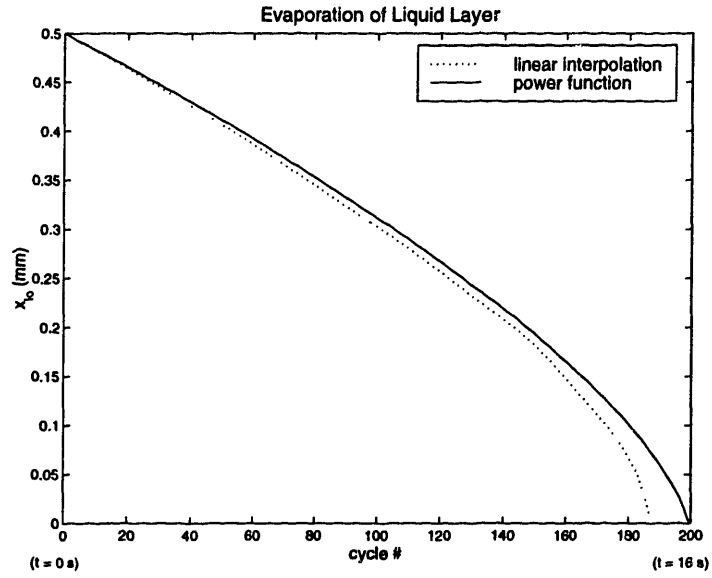


Figure 6-34: Evaporation of a 0.5 mm liquid layer for baseline case

approximation is expected to have smaller error than the linear interpolation, and so is closer to the desired results (although it is not possible to establish it as a lower or upper bound). The power function has been used for calculations presented in the later sections of this chapter.

Figure (6-34) is distinguished by a fairly constant initial slope. Therefore, only a few $(\Delta x_l\%)$/cycle points need to be calculated for this region ($x_{lo} > 0.4$ mm in this case). As the liquid layer becomes thinner, the absolute value of the slope increases until all the liquid is completely evaporated at 200 cycles. This is due to the fact that thinner liquid layers will be heated to higher temperatures, as shown in the previous section, which leads to faster evaporation. For x_{lo} smaller than 0.4 mm, more calculated data points allow for better resolution and more accurate results. Once x_{lo} becomes smaller than $\sim$ 0.01 mm, evaporation is completed within a few cycles, which may be less than the accuracy that is available due to possible sources of error. Therefore, once $x_{lo} \sim 0.01$ mm, evaporation can be said to occur within a cycle.

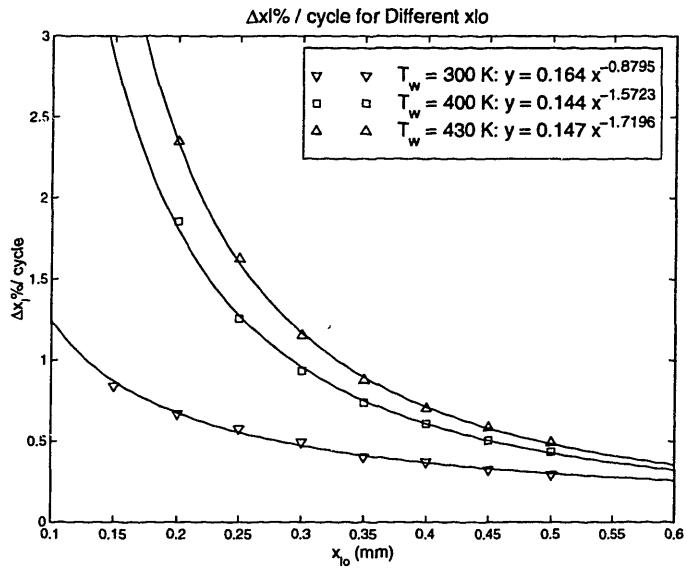


Figure 6-35: $\Delta x_l\%$ during cycle for varying wall temperatures

6.8 Effects of Wall Temperatures on Evaporation and Oxidation

The evaporation process seems to be a sensitive function of the thermal environment in the region of the liquid layer. It is therefore expected that wall temperatures may play an important role in the time for evaporation. This section quantifies the effect of possible operating values of wall temperature on evaporation characteristics of a liquid layer undergoing combustion cycles as described in previously.

The temperature of the wall, T_w, is a input parameter to the problem, and appears as a Dirichlet left boundary condition in the solution method. As mentioned above, baseline case calculations were performed with $T_w = 361$ K. With the exception of different values for T_w, the calculations in this section were performed with identical initial and operating conditions to those in the last section. Values of wall temperature chosen for this section are 300 K, 400 K, and 430 K. These values represent relatively cool operating conditions for a typical IC engine; usually experienced during the warm up period (when the engine is most likely to require flooding to ensure firing).

Figure (6-35) is a plot of $(\Delta x_l\%)$/cycle vs. x_{lo} for these wall temperatures and corresponding power fits. As expected, hotter walls produce larger changes in the

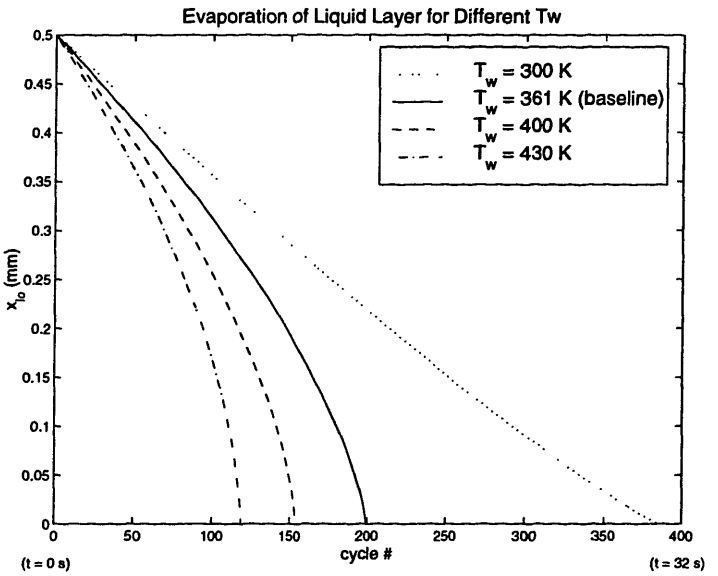


Figure 6-36: Evaporation of a 0.5 mm liquid layer for varying wall temperatures

liquid layer thickness for a cycle. Figure (6-36) displays total evaporation of a 0.5-mm liquid layer over time for the temperatures above.

Figure (6-36) shows that higher wall temperatures result in shorter times for total evaporation. It also shows that, for thicknesses of order less than 0.4 mm, the rate of evaporation per cycle (slope of the lines) will increase significantly until evaporation is complete for the cases in which T_w has values of 361 K, 400 K, and 430 K. This is not observed, however, in the case where T_w = 300 K, which shows a relatively constant rate of evaporation for the entire run with a slight decrease near complete evaporation. Compared to Figure (6-25) for the baseline case, Figure (6-37) shows that the change in thickness during a cycle is a relatively weak function of x_{lo} for the 300 K calculations.

This was found to be the case for all wall temperatures less than approximately 340 K. This temperature is very close to the boiling point for the liquid at later stages in the cycle, $T_b \approx 338$ K for all cases. The liquid interface cannot reach a temperature above T_b, which governs its heat transfer characteristics. If wall temperatures are greater than or near T_b, heat transfer can only occur *to* liquid layers from the wall and the gases. Evaporation is the only mechanism available for heat transfer *from* the liquid layer under these circumstances, which must be kept at temperatures lower

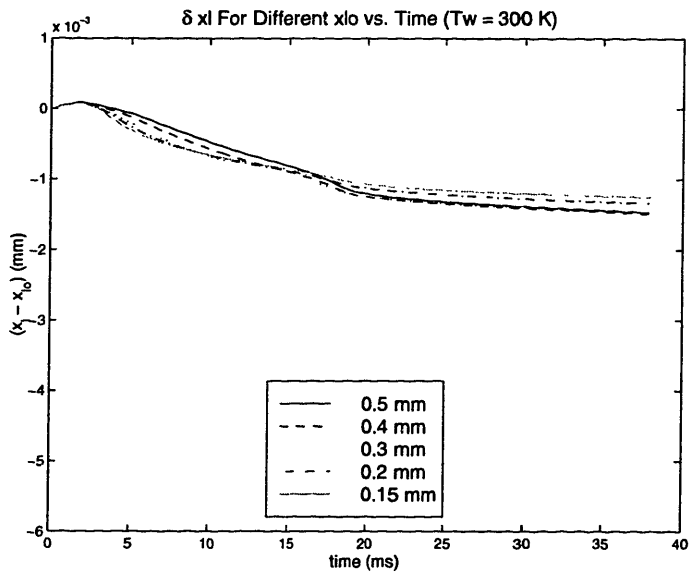


Figure 6-37: Decrease in liquid layer thickness during cycle for different initial thicknesses, x_{lo}, at $T_w = 300$ K

than T_b. This heat into the layer is therefore used to drive evaporation faster as the total heat capacity of the wall decreases with decreasing x_{lo}. If the wall temperature is below T_b, however, it is possible to transfer heat into the wall (away from the liquid layer interface).

This avenue of heat transfer provides a sink for the heat transferred to the liquid layer from the hot gases. Figure (6-38) supports this observation, since it shows that, for $T_w = 300$ K, T_l decreases for all liquid layer thicknesses throughout the entire cycle after the initial peak. It shows that as the liquid layer becomes thinner, the interface temperature drops faster. Eventually, as $x_{lo} \to 0$, $T_l \to T_w$, which slows evaporation since $T_w < T_b$. In contrast, Figure (6-26) for the baseline case showed that as $x_{lo} \to 0$, $T_l \to T_b$, which greatly enhanced evaporation. Therefore, enhanced evaporation for thinner liquid layers will occur for $T_w \geq T_b$, which is often the case in engines.

Figure (6-39) shows the survival rate for a liquid layer of a representative thickness of 0.5 mm for the different wall temperatures. It shows that as the value of T_w increases there is only a negligibly slight increase in the amount of surviving fuel. For all cases, the amount of surviving fuel at the end of the cycle is still on the order of

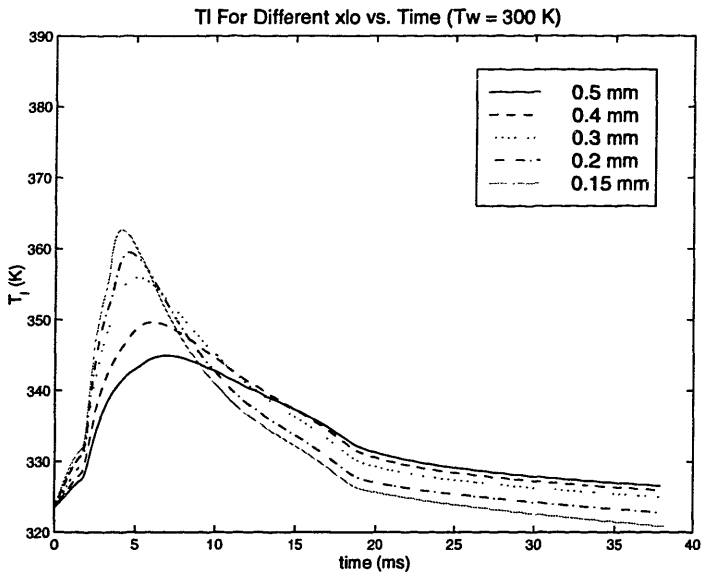


Figure 6-38: Liquid-gas interface temperature T_l during cycle for different initial thicknesses, x_{lo}, at $T_w = 300$ K

2.9%, similarly to the values of figure (6-31).

It can be concluded, then, that although T_w has a significant impact on the time required to vaporize a liquid layer, it has very little effect on the total amount of surviving fuel from the liquid layer. Previous results have shown, however, that as a flame approaches a dry wall or crevice [2], increased wall temperatures result in lower final emissions. This is due to the fact that hotter walls reduce the quench distance of the flame, allowing for a higher rate of oxidation. In the case of a wall covered by a liquid layer, however, this is not the case. The flame comes into contact with the liquid layer only, which is limited to a maximum temperature of T_b for any given pressure. For a particular pressure history, therefore, the quench distance is roughly constant, regardless of the wall temperature and liquid layer thickness. The liquid layer acts as a barrier, at a temperature of $\sim T_b$, between the wall and the hot gases.

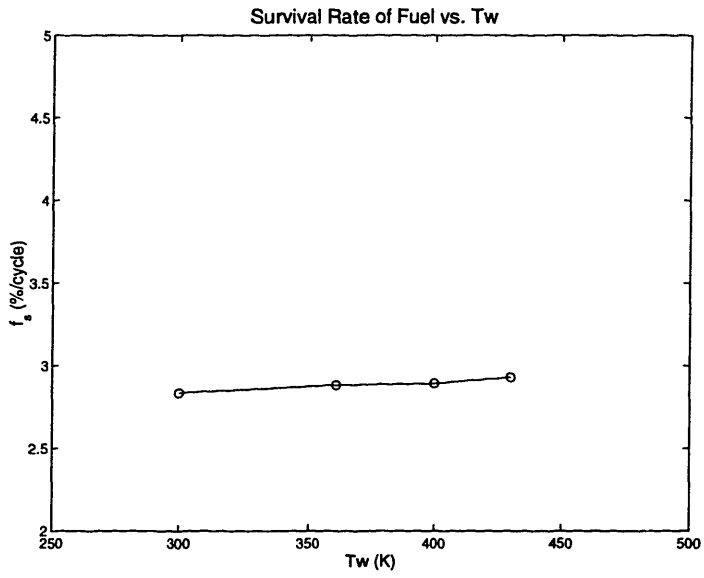


Figure 6-39: Liquid-gas interface temperature T_l during cycle for different initial thicknesses, x_{lo}, at $T_w = 300$ K

6.9 Effects of Volatility on Evaporation and Oxidation

6.9.1 Determining Vapor Pressure

One of the distinctive fuel properties that affect evaporation is the volatility of the liquid. The volatility can be represented by the amount of energy necessary to evaporate a certain mass of fuel: the heat of vaporization, L_t. This section addresses the effect of different volatilities on the evaporation of a liquid fuel layer. Calculations are shown in which the heat of vaporization is varied in relation to the baseline value of methanol.

The vapor pressure relation of equation (3.22) is a function of L_t. In the calculations shown in the previous sections, this relation was used with experimental values for A_f, B_f, and C_f for best accuracy. Since L_t will vary arbitrarily in this section, however, equation (3.22) must be derived accordingly in order to conserve energy and entropy in the system.

From the Clausius-Clapeyron relation [18], it is known that

$$\frac{dp}{dT} = \frac{L_t}{Tv_g}. \tag{6.5}$$

for a simple substance in liquid-gas phase equilibrium. Integration of Equation (6.5) with the ideal gas equation of state provides the equation for vapor pressure, which shows the dependence on L_t:

$$P_v/P_o = e^{(ML_t/RT_o)}e^{-(ML_t/RT)}. \tag{6.6}$$

Where M is the molar mass of the gas and R is the universal gas constant. Comparing to Equation (3.22), the parameters A_f, B_f, and C_f can be obtained,

$$A_f = \frac{\ln P_o + ML_t/RT_o}{\ln 10}, \tag{6.7}$$

$$B_f = \frac{ML_t}{R\ln 10}, \tag{6.8}$$

$$C_f = 0, \tag{6.9}$$

for

$$P_v/P_o = 10^{A_k - \frac{B_k}{T_l + C_k}}. \tag{6.10}$$

Equations (6.7)-(6.10) show the expected agreement with values of P_v obtained from experimentally determined A_f, B_f, and C_f (and used in previous calculations), as shown in Figure (6-40). Figure (6-41) shows that the impact on total oxidation is also small: for the baseline case, full calculation for total oxidation resulted in a difference of 3/258 cycles between the two methods of obtaining P_v. Since this is considered within possible sources of error, it is reasonable to use Equations (6.7)-(6.10) for the calculations in this section.

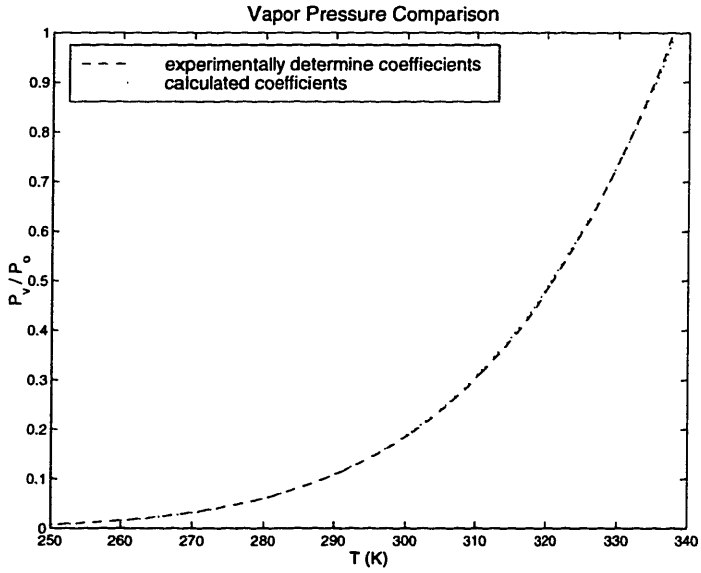


Figure 6-40: Comparison of vapor pressure for methanol as a function of temperature between experimental values of Antoine coefficients and results of equations (6.7)-(6.10)

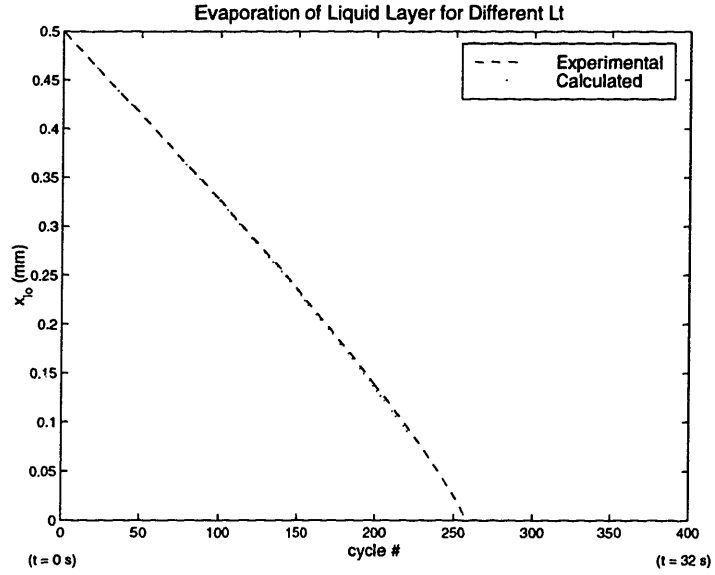


Figure 6-41: Impact of calculated versus experimental coefficients of P_v on total oxidation

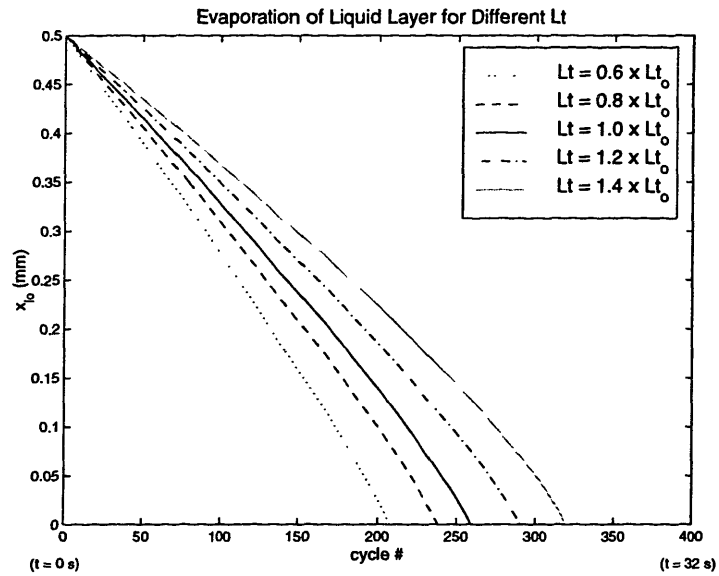


Figure 6-42: Impact of different values of L_t for total oxidation

6.9.2 Results

In this section, different values of L_t (and resulting P_v at fixed P_o) are imposed on a methanol liquid layer and the evaporation process calculated. L_t has been changed by factors of 0.6. 0.8, 1.0, 1.2, and 1.4 to the original value for methanol in order to observe the system's sensitivity to this parameter.

Figure (6-42) shows the total evaporation process for different L_t. Larger values require a longer amount of time for complete evaporation. For the values of L_t listed above, 207, 238, 258, 290, and 319 cycles were observed for complete evaporation of a 0.5 mm liquid layer, respectively. This might be expected due to the larger amount of energy required to evaporate a unit amount of mass. Since the total amount of energy released by the flame per unit fuel burned in all cases is relatively equal, vaporization rates must be smaller for larger L_t, as is shown in the figure.

Another mechanism that influences the vaporization process is the value of P_v for different L_t. From equations (6.7)-(6.10), Figure (6-43) shows that the vapor pressure (as a function of temperature) decreases for increasing L_t. This reduces the equilibrium fuel species mass fraction at the interface, also acting to slow the vaporization process and resulting in larger evaporation times.

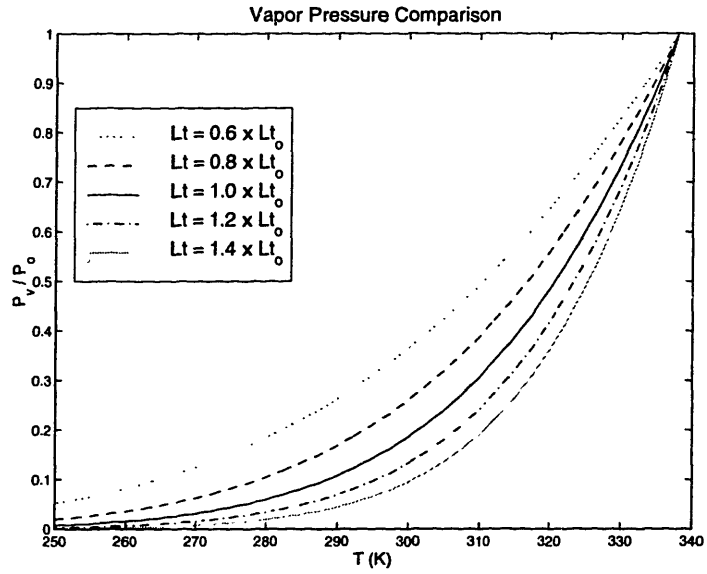


Figure 6-43: Impact of different values of L_t on P_v

Although vaporization rates are observed to have a sensitive dependence on L_t, Figure (6-44) shows that the vapor-phase fuel survival rate for a representative thickness of 0.1 mm, f_s, is quite insensitive to this parameter (roughly 2.9% as in previous results). As has been concluded above, this is due to the similarities in the diffusion flame processes that occur immediately after flame arrival. There is a slight decrease in f_s observed with increased L_t, however, which is apparently due to the lower fuel vapor pressures observed in Figure (6-44). The resulting lower equilibrium concentrations at the liquid-gas interface form fuel layers between the liquid and the flame that are characterized by lower fuel concentrations, resulting in lower unburned fuel emissions.

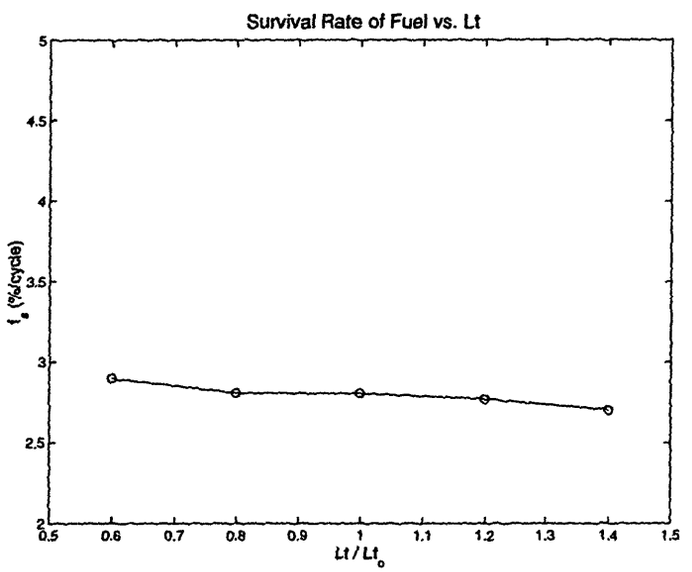


Figure 6-44: Impact of different values of L_t on f_s, $x_{lo} = 0.1$ mm

Chapter 7

Conclusions

The present version of this model was shown to be successful in modeling one dimensional evaporation, reaction-diffusion problems. The model solves for evaporation driven by species diffusion and heat transfer at the liquid-gas interface by employing an eigenvalue solution method. Given a chemical kinetic mechanism for the fuel in question, the model also solves for reaction in the gas phase as the species diffuse into hot gases (possibly forming a diffusion flame). Analytical and experimental comparisons with the numerical results were shown to result in good agreement.

Although the model can also be applied to either cylindrical or spherical geometries (droplet problems) this work is mainly concerned with evaporation processes that occur in a flat cartesian geometric configuration. In particular, the problem in question consists of a flat wall covered by a thin layer of liquid fuel. Conditions are such that a flame approaches the liquid layer as the pressure changes during the process. It is therefore possible to input pressure histories similar to those in operating engines in order to simulate such environments. The pressure history can be constant or arbitrarily specified to conform to any desired situation.

Constant pressure calculations showed that the time scales for the formation of fuel species boundary layers from the liquid interface are on the order of 5 ms. Such time scales are relatively short compared to the those of changing pressure in engine cycle near IVC, but of the same order of those after spark and before peak pressure (Figure (6-11)). Therefore, this region of the cycle is likely to be characterized by

non-fully developed boundary layers (an observation confirmed in Figure (6-15)).

Higher constant pressures produce larger rates of evaporation, despite similar initial (before flame arrival) thermal environments in all cases. This was attributed to the fact that high pressures form a lean region near the interface. Phase-equilibrium fuel mass fractions for pressures larger than roughly 3 atm were observed to result in lower than core gas fuel species mass fractions at $\phi = 0.9$. As a result, species boundary layers for higher pressures are characterized by larger amounts of oxygen before flame arrival. For 5- and 10-atm calculations, these regions provide enough oxygen to allow for the existence of a diffusion flame near the interface as fuel is evaporated after flame arrival. This was not observed to be the case for a 1-atm calculation, which showed a rich region near the interface before flame arrival, no diffusion flame after flame arrival, and much lower rates of evaporation.

Engine pressure histories were obtained from a zero-dimensional simulation [9] with operating conditions summarized in Table (6.1). During the compression part of the process, it was observed that the increase in pressure of core gases (and subsequence increase in core gas temperatures due to compression) resulted in an increase in liquid layer temperatures, T_l. Both pressure and temperature, however, act in opposite directions regarding promotion of evaporation. Under the conditions of the cycle, it was shown that the increase in pressure dominates the process, leading to overall condensation during this part of the cycle. This observation suggests that for the compression of gases, the pressure rise is likely to affect vaporization to a larger extent than the resulting change in liquid surface temperature due to changing gas temperatures. It was also noted that condensation begins before surface fuel mass fractions drop below those of the core gas. This is due to the fact that for these time scales, as noted above, a fuel boundary layer does not completely form, and condensation begins as soon as $(\partial Y_{fuel}/\partial r)_s > 0$.

Although condensation was observed for this portion of the cycle (before flame arrival), once the flame reaches the liquid layer at peak pressure, and the core gas temperatures increase substantially due to the flame, evaporation accounts for a much larger change in liquid layer thickness. In this situation, both processes act to pro-

mote evaporation; high temperatures and dropping pressure lead to increased phase-equilibrium fuel species concentrations at the boundary. Due to the previous pressure rise, the conditions at flame arrival are similar to those observed in the high constant pressure calculations. The relative abundance of oxygen produces diffusion flames in all cases observed. These flames serve as sources of energy and sinks of fuel throughout the expansion process, greatly enhancing vaporization. The existence of the diffusion flame is apparent in the $\dot{m}_s$ and q''_{in} vs. time plots as secondary peaks in the curves during the cycle (Figures (6-19) and (6-20)), the peak first attributed to the initial heat transferred from the arrival of the premixed flame. The existence of the diffusion flame is also confirmed by the temperature profiles after initial flame arrival (Figure (6-17)).

The initial liquid layer thickness was observed to have a significant impact on the rate of vaporization. Calculations performed with various thicknesses $x_{lo} < 0.5$ mm showed that as x_{lo} decreases, liquid surface temperatures rise faster and to higher values. Higher T_l increase the values of the phase-equilibrium concentrations at the interface, which require larger rates of vaporization to compensate for diffusion and maintain phase-equilibrium conditions. As a result, the amount of liquid evaporated per cycle increases dramatically as $x_{lo} \to 0$.

For a set of $\dot{m}_s$ vs. x_{lo} data points, it was observed that power functions fit the points adequately, leading to a satisfactory interpolation between points. This information was then used to construct a model for the total amount of time required for the evaporation of a liquid layer of given initial thickness for certain operating conditions. A 0.5-mm liquid layer under baseline conditions, composed of a methanol liquid layer, wall temperature of 361 K, and operating conditions of Table (6.1), required 200 identical cycles for complete evaporation. Once the liquid layer attained a thickness of $x_{lo} \approx 0.01$ mm, it evaporated in a single cycle.

The rate of vaporization was also shown to be a function of wall temperature, T_w. As expected, $\dot{m}_s$ increases for higher values of T_w due to heat transfer from the wall. It was also observed that at $T_w \approx T_b$, where T_b is the boiling point of the liquid, there is a change in the characteristics of the vaporization rate over the number

cycles since the liquid-gas interface is limited to temperatures $T_l \leq T_b$. For $T_w < T_b$ heat can be transferred to the wall, allowing liquid layers to more efficiently conduct heat from the hot gases to the wall as x_{lo} decreases, which provides a heat transfer avenue that is faster than evaporation, decreasing vaporization rates. Conversely, for $T_w > T_b$ heat can only be transferred to the liquid layer. In this case, to maintain the interface temperature below T_b, vaporization becomes increasingly faster for thinner layers that have lower heat capacities.

To observe the effect of volatility on the rates of vaporization, the heat of evaporation, L_t, was varied from the baseline case of methanol. The effect on vapor pressure was taken into account for these calculations. It was observed that higher values of L_t result in lower rates of vaporization due to two effects. Higher values of L_t require larger amounts of energy to vaporize a unit mass. Since the amount of energy from the flame is relatively equal for all cases, the amount of mass vaporized decreases for larger L_t. A more subtle effect that contributes to the decrease in vaporization is the fact that as L_t increases, the vapor pressure curve decreases for any given value of T_l (Figure (6-43)). The resulting decrease in fuel species phase-equilibrium values require lower rates of vaporization to counter the lower diffusion rates.

Interestingly, the survival rate of fuel evaporated, f_s, was consistently found to be relatively insensitive to x_{lo}, T_w, or L_t. Under all conditions, f_s was calculated to be approximately 2.9% per cycle. This is due to the fact that the diffusion flame formed after initial flame arrival is the controlling mechanism for oxidation. Therefore, higher rates of evaporation lead to higher rates of diffusion into the flame, while lower rates of evaporation lead to lower rates of diffusion. Under all conditions, the temperature limitation on the liquid-gas interface of $T_w < T_b$ insures that gas boundary layer temperatures for all the cases calculated were very similar. The overall result is that the total amount of surviving fuel is only very weakly affected by the parameters above.

Appendix A

Nomenclature

r	coordinate direction
t	time
CAD	crank angle degrees
r_s	location of liquid-gas interface
a	geometric configuration (1: cartesian, 2: cylindrical, 3: spherical)
g	gas phase subscript
l	liquid phase subscript
ρ	density of gas or liquid
Y_k	local mass fraction of species k
X_k	local mole fraction of species k in the gas phase
x_k	local mole fraction of species k in the liquid phase
T	local temperature of liquid or gas
P	pressure
P_k	partial pressure of species k
P_v	vapor pressure
D_k	diffusivity of species k
h_k	specific enthalpy of species k

C_p	constant pressure specific heat
C	liquid specific heat
λ	thermal conductivity
v_g	gas velocity at position r
v_s	gas velocity at the interface position
$\dot{\omega}_k$	mass production rate of species k
K	total number of species contained in system
$\tilde{K}$	number of species in the gas phase that interact with the liquid phase
Pe_l	liquid Peclet Number
$\dot{m}_k$	mass flux of species k leaving the liquid domain
$\dot{m}_s$	mass flux of fuel species at the liquid-gas interface
A_l	surface area of liquid domain
V_l	volume of liquid domain
o	subscript used for "left boundary"
∞	subscript used for "right boundary" at infinity
T_w	temperature of rigid wall
T_l	temperature of liquid-gas interface
q''_w	imposed heat flux at the wall
q''_{in}	heat flux into the liquid-gas interface from gas phase
q''_{evap}	Heat used in evaporation process
A_k	Antoine coefficient
B_k	Antoine coefficient
C_k	Antoine coefficient
W_k	molecular weight of species k
ε_k	fractional gasification rate of species k
t_f	point in time when flame first reaches liquid-gas interface
L_t	total heat of vaporization
f_s	survival rate of fuel from evaporated layer

Appendix B

Detailed Equations

Note: for a given discretization at the interface, s refers to the point at the interface, $(-)$ refers to the first point before s, and $(+)$ refers to the first point after s.

$$f_{T_{l\partial r}} = \frac{N_1 N_2 N_3}{D} \tag{B.1}$$

where

$$N_1 = v_s \sum_{k=1}^{K} \varepsilon_k L_k \left(\frac{v_s \rho_{g,s}}{\lambda_{g,s}} + \frac{1}{C_{p,g,s}(r_{(+)} - r_s)} \right),$$

$$N_2 = \frac{r_{(+)}^{(a-1)}}{r_s^{(a-1)} \rho_{g,s} C_{p,g,s}(r_{(+)} - r_s)} \left(\lambda_{g,(+)} \frac{\partial T_g}{\partial r} \right)_{(+)},$$

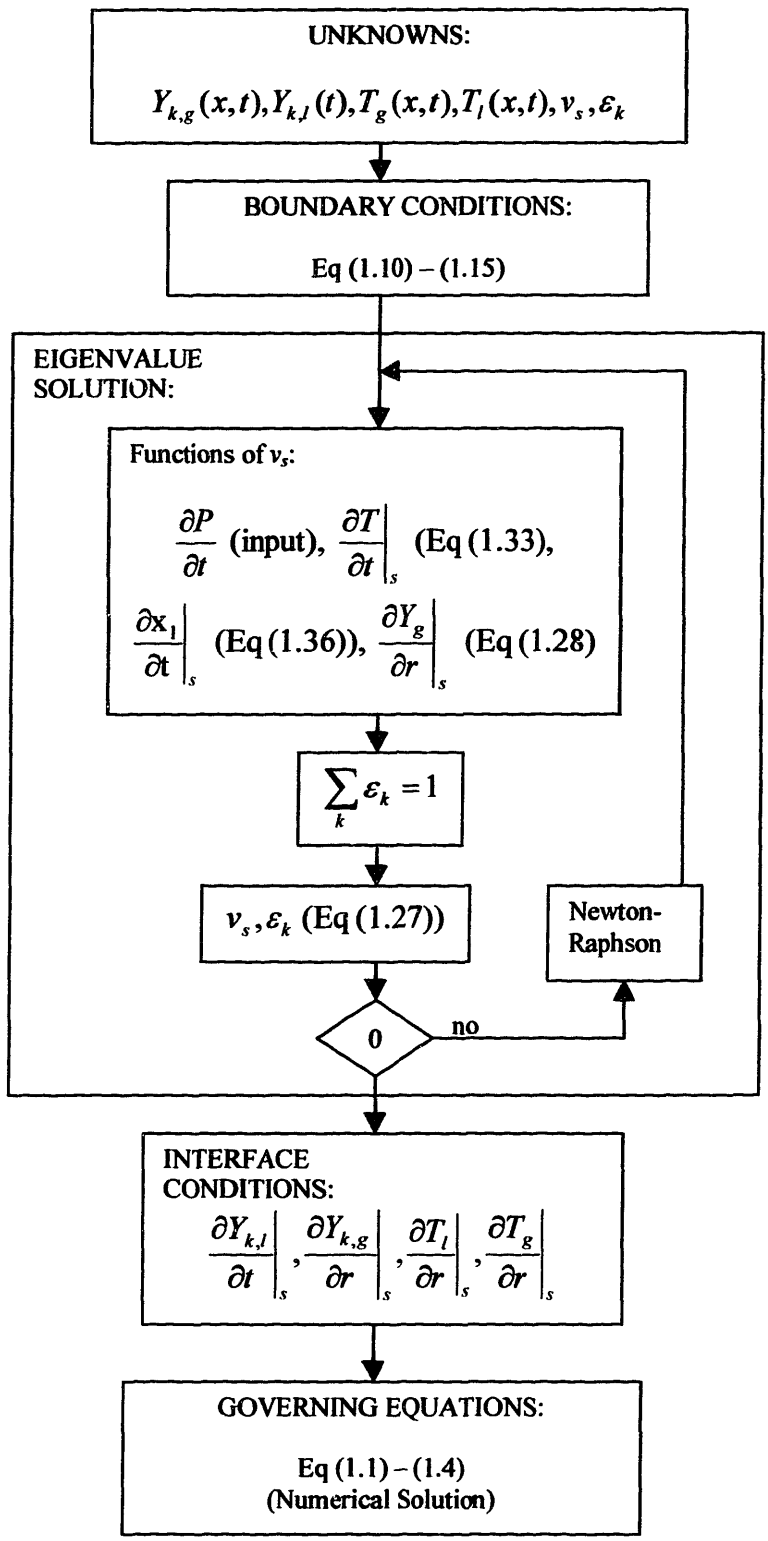
$$N_3 = \frac{r_{(-)}^{(a-1)}}{r_s^{(a-1)} \rho_{g,s} C_{p,g,s}(r_s - r_{(-)})} \left(\lambda_{g,(-)} \frac{\partial T_g}{\partial r} \right)_{(-)},$$

$$D = \frac{v_s \lambda_{l,s}}{\lambda_{g,s}} + \frac{\lambda_{g,s}}{\rho_{g,s} C_{p,g,s}(r_{(+)} - r_s)} + \frac{\lambda_{l,s}}{\rho_{l,s} C_{p,l,s}(r_s - r_{(-)})}.$$

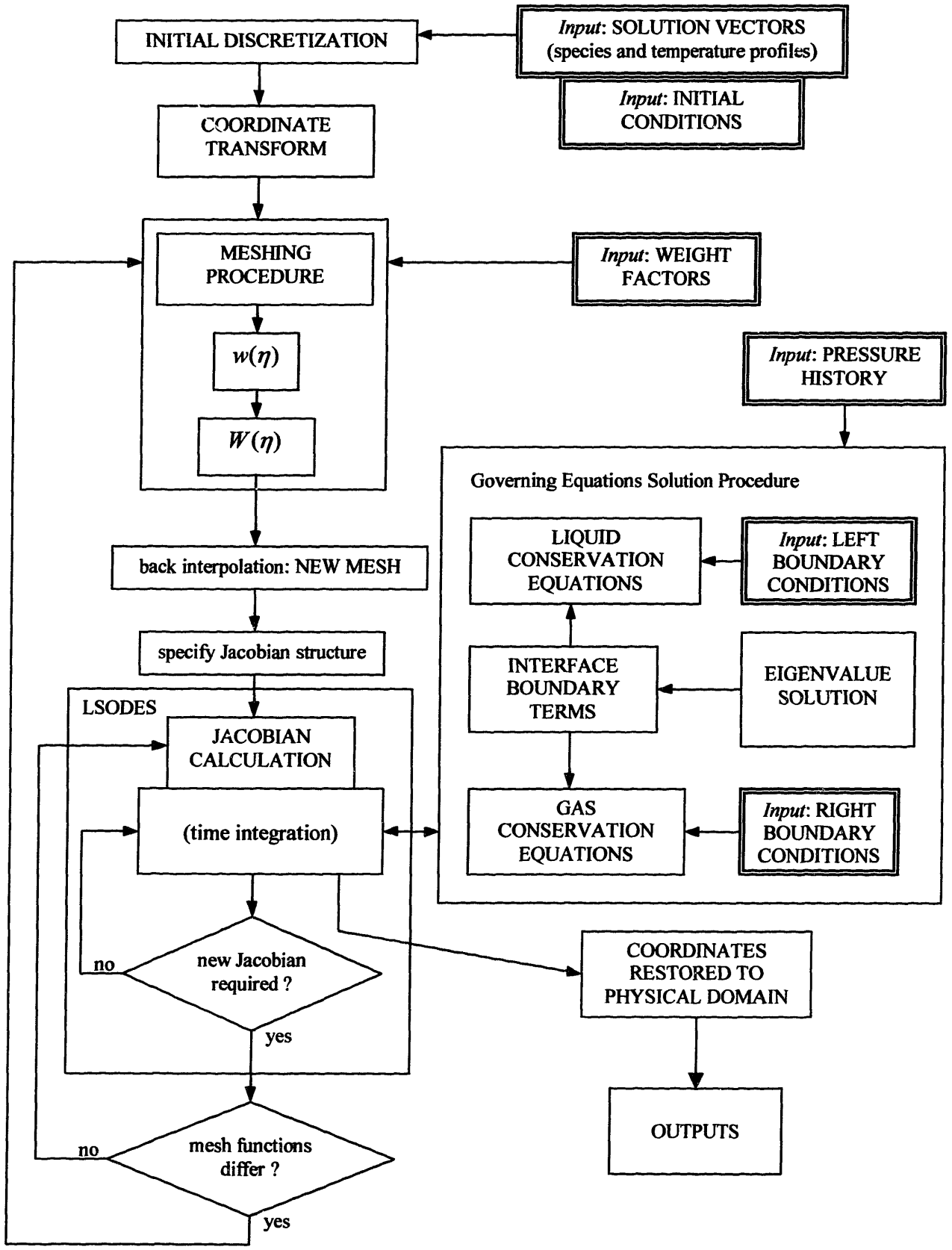
$$f_{T_{\partial t}} = \frac{1}{\rho_l C_{p,l} r_s^{a-1}(r_s - r_{(-)})} \left(r_s^{a-1} \lambda_{l,s} \frac{\partial T_l}{\partial r} - r_{(-)}^{a-1} \lambda_{l,-} f_{T_{l\partial r}} \right) \tag{B.2}$$

$$f_{x_{k\partial t}} = \frac{A_l v_s \rho_{g,s} (\varepsilon_k + Y_{k,l}) / (W_{k,l} \sum_j Y_{j,l}/W_{j,l}) - Y_{k,l}/W_{k,l} \sum_j A_l v_s \rho_{g,s} (\varepsilon_k + Y_{k,l})/(W_{j,l})}{(V_l \rho_l \sum_j Y_{j,l}/W_{j,l})^2}. \tag{B.3}$$

Appendix C – Flowchart #1



Appendix D – Flowchart #2



Appendix E

Code Listing

MAIN.F

```
c ************************************************************************
      program main
c ************************************************************************
c *                                                                      *
c *                                                                      *
c *                              1d_chem                                 *
c *                                                                      *
c *          Reactive/Diffusive/Convective Boundary Layer Problem        *
c *                                                                      *
c *                                                                      *
c ************************************************************************
c
c contacts ------------------------------------------------------------------
c
c Principal Investigator:  Simone Hochgreb, MIT       simone@mit.edu
c
c Authors:                 Ivan Oliveira, MIT         oliveira@mit.edu
c                          Chris O' Brian, MIT        cjobrien@mit.edu
c                          Kuo-Chun Wu, MIT
c
c Funded by the EPA Center for Airborne Organics
c & MIT Engines and Fuel Consortium
c
c     This program calculates the development of PDEs governing mass and
c energy conservation equations for a one-dimensional, reactive/diffusive/
c convective problem.  It can be used to calculate the species and temperature
c profiles on a flat or spherical coordinate system for a problem with solid
c (impermeable) or liquid boundaries.
c     This verson uses a coordinate transform of the spatial domain, the
c method of line to discretize the transformed coordinate, and a Backward
c Difference Formula package (LSODES) to solve the resulting ordinary
c differential equations.  The Sandia CHEMKIN gas-phase subroutines are used
c for information regarding gas properties.  This version also provides a
c remeshing option that greatly improves the calculation efficiency.  The
c remeshing algorithm is based on the equidistribution of a remesh function
c and back-interpolation of points onto the spatial domain.  Details are
c provided in headers for the respective routines.  All units are converted
c to MKS (kg, m, s).
c     This version allows for the simulation of problems involving liquid
c layers between the wall and combustion gases or liquid droplets.  The
c solution involves an eigenvalue problem which provides interface terms
c that serve as boundary conditons for both liquid and gas domains.
c Energy conservation is applied to the liquid phase to determine heat
c transfer characteristics and liquid layer temperatures.
c     Currently, examples of problems in which the program has been
c tested successfully include:
c
c       o diffusion of a cold boundary layer into hot gases
c       o engine hydrocarbon oxidation in the gas and liquid phases
c       o liquid droplet combustion (spherical)
c       o evaporation of liquid droplets
c       o flame into cold liquid layer on constant Temp wall
c       o effects of pressure on liquid layer evaporation/condensation
c
c                                                  Ivan Oliveira
c                                                   05/03/1999
c
c ============================ Summary of usage ================================
c
c   Four files are necessary in order to use this program.  They are:
c
c     1) cklink
```

```
c      2) tplink
c      3) initial
c      4) prop.f
c
c   1) 'cklink' is a standat link file from created in chemkin
c   2) 'tplink' is a transport property file created in tranfit
c   3) 'initial' is a user-generated input file that specifies the initial
c      conditions in the gas phase.  The required format is as follows:
c        o Line 1: labels of columns to follow (x, Y1...YK, T)
c        o Line 2 to n+1: values of initial conditions (x, Y1...YK, T)
c        o Remaining lines: input parameters:
c          Time stepping parameters
c            TINC       increment of time at which to write output data
c            TMAX       maxium time for run (at which program will stop)
c            rp         initial radius of liquid surface
c            corder     '0' for flat, '1' for cylindrical, '2' for spherical
c            turb       '0' for laminar, '1' for turbulence model
c            chem       '0' for chemistry off, '1' for chemistry on
c                Remesh parameters for gas-phase point distribution
c            remesh     'on' or 'off'
c            mindist    minimum distance allowed between points
c            maxdiff    maximm
c            meshpoints number of mesh points to distribute
c           Liquid mesh parameters
c            nptsl      number of points in the liquid mesh
c            fuel       index of fuel in cklink file
c         o Last line: 'END'
c      Notes:
c      - It is necessary that the first two x positions be close to each
c        other if given concentrations at the liquid-gas interface are not at
c        phase equilibrium.  Otherwize there may not be proper definition
c        before coordinate transform is executed.
c      - Comments may be inserted in the file if preceeded by '!'.
c      - Specing is not important between columns.  Empty lines are ignored.
c      - Species do not have to be in the same order as in cklink and
c        not all species must be specified (specified not species assume
c        an initial value of zero on the entire domain).
c      - rp, turb, chem, mindist, maxdiff, and meshpoints are optional
c      - after a point has been declared, the format:
c dx   #
c        may be used in the next line to indicate that the values are to
c        be linearly interpolated to next point, with separation #.
c      - The following example is for a methanol liquid droplet evaporating
c        in an atmosphere at room temperature.  Note the spherical geometry
c        specification, laminar and non-reacting options enabled, and mesh
c        point options provided.
c
c ------------------- sample 'initial' file (within lines) --------------------
c x       CH3OH O2 N2 H2O CO2 Temperature
c
c 0.00000  0.0000     .23000     0.7700     0.0000    0.0000  298.0
c
c 0.000001 0.0000     .23000     0.7700     0.0000    0.0000  298.0
c dx       0.0005
c
c 0.00105  0.0000     .23000     0.7700     0.0000    0.0000  298.0
c dx       0.0001
c
c 0.0056   0.0000     .23000     0.7700     0.0000    0.0000  298.0
c dx       0.005
c
c 0.015    0.0000     .23000     0.7700     0.0000    0.0000  298.0
c dx       0.01
```

```
c
c 0.050    0.0000     .23000     0.7700     0.0000     0.0000  298.0
c
c ! Time stepping parameters
c TINC       2.00E-01
c TMAX       4.00E+00
c rp         0.50E-03
c corder     2
c turb       0
c chem       0
c ! remesh parameters
c remesh     on
c mindist    1.00E-14
c maxdiff    0.1
c meshpoints 100
c ! liquid mesh parameters
c nptsl      30
c fuel       1
c END
c ------------------------------------------------------------------------
c
c      4) the file 'prop.f' defines the properties of the liquid that will
c         be used as a function of temperature.  These are simple fortran
c         files that MUST BE COMPILED WITH CODE when new liquids are to be
c         modeled.  Default is methanol.  See default 'prop.f' file for the
c         necessary format information and data required.
c
c
c ========================= End of Summary of usage ============================
c
      implicit none
c declarations ---------------------------------------------------------------
c constants:
      integer               npdemx, nptsmx, neqnmx
      parameter             (npdemx = 80, nptsmx = 205, neqnmx = npdemx*nptsmx)
c for liquid
      integer               npdel, nptsl, nptslor
c global variables:
      double precision      x(nptsmx), rx(nptsmx), ts, tcurr, xout(nptsmx),
     1                      pressure, dpdt, length, avespeed, tconst,
     2                      gasconst, mp, Qin, Qout, mold, Qinold, Qoutold,
     3                      minrp, Yold, Tempold, ignite, vrpold, Tw
      integer               npts, nspecies, nxeqn, turb, corder, chem, fuel
      common /fluxdat/      tcurr, mp, Qin, Qout, mold, Qinold, Qoutold,
     1                      Yold, Tempold, vrpold
c ptinfo used in stode()
      common /liquid/       nptsl, npdel, nptslor
      common /ptinfo/       x, rx, npts, nspecies, nxeqn
      common /pressdat/     pressure, dpdt, length, avespeed, tconst
      common /misc/         gasconst, minrp, turb, corder, chem, fuel
c main() variables:
      integer               i, j, k, l, npde, neqn, neqng, iteration
      real                  time(2), dtime, elapsed
      doubleprecision       u(neqnmx), y(npdemx), Temp(nptsmx), wtm(nptsmx),
     1                      rho(nptsmx), uout(neqnmx), ux(neqnmx), udot(nptsmx),
     2                      sumspec(neqnmx), tsmallest, tout, tmax, tinc,
     3                      time_ini, timeold, timesucc, rp, Templ(nptsmx)
      common /timedif/      timeold, timesucc
      logical               more
c tolerance declarations:
      integer               itol, itask
      double precision      atol(neqnmx), rtol
c CHEMKIN declarations:
```

```
      integer               leniwk, lenrwk, lencwk
      parameter            (leniwk = 7500, lenrwk = 90000, lencwk = 500)
      integer               ickwrk(leniwk), imcwrk(leniwk)
      double precision      rckwrk(lenrwk), rmcwrk(lenrwk)
      character             cckwrk(lencwk)*(16)
      common /ckspi/        ickwrk, imcwrk
      common /ckspr/        rckwrk, rmcwrk
      common /ckspc/        cckwrk
c general file declarations:
      integer               LINKCK, LINKMC, LCADPL, LINIT, LMESH, LINTEG,
     1                      LOUT, LSCREEN, LDATA, LRESULT, IcKK
      parameter            (LINKCK = 25, LINKMC = 35, LCADPL = 10, LINIT = 15,
     1                      LMESH = 12, LINTEG = 20, LOUT = 30, LSCREEN = 6,
     2                      LDATA = 18, LRESULT = 8)
c general input data:
      common /INITDAT/      tmax, tinc
c
c ---------------------------- optional integrators ---------------------------
c
c LSODES declarations for non-specified Jacobian structure:
c (comment LSODES* below and uncomment the following lines to activate
c  this option)
c      integer              istate, iopt, nnz, lenrat, mf, liw, lrw, lwm,
c     +                     itrace, ipminf
c      parameter           (mf = 222, nnz = 10, lenrat = 2,
c     +                     lrw = 900000, liw = 30)
c      double precision     rwork(lrw), jac, totime
c      integer              jacint, iwk(lrw), iwork(liw)
c LSODE declarations:
c (comment LSODES* below and uncomment the following lines to activate
c  this option)
c      integer              istate, iopt, ml, mu, mf, liw, lrw,
c     +                     itrace, ipminf
c      parameter           (mu = 3*12-1, ml = 3*12-1, mf = 25,
c     +                     lrw = 22 + 10*neqnmx + (2*ml + mu)*neqnmx,
c     +                     liw = 20 + neqnmx)
c      double precision     rwork(lrw), jac, totime
c      integer              jacint, iwork(liw), iwk(lrw)
c
c ------------------------- end optional integrators ------------------------
c
c LSODES* declarations for calculated Jacobian structure option (default):
      integer               istate, iopt, nnz, lenrat, mf, liw, lrw, lwm,
     1                      itrace, ipminf
      parameter            (mf = 22, nnz = 10, lenrat = 2,
c recommended for ~50 species: lrw = 3.2e6, liw = 1.5e6
     2                      lrw = 3.2e6, liw = 1.5e6)
      double precision      rwork(lrw), jac, totime
      integer               jacint, iwk(lrw), iwork(liw), ia_len, ja_len
c end of LSODES* declarations
c additional LSODE(S) declarations:
      integer               stats(3)
      common /Lwork/        iwork
c REMESH declarations (used in integrator stode() and read_initial()):
      integer               nptsnew, nptstotry, remeshflag, ormesh, nptsold
      logical               different
      double precision      a(npdemx), b(npdemx), S(nptsmx), xmeshold(nptsmx),
     1                      xjnew(nptsmx), ynew(nptsmx), maxdiff, mindis, avg,
     2                      meshTime, avgarray(npdemx), maxavg, mindistance,
     3                      Fmeshold(nptsmx)
      common /remesh1/      nptsnew, nptstotry, remeshflag, ormesh, different
      common /remesh2/      u, a, b, npde
      common /remesh3/      S, xjnew, ynew, maxdiff, mindistance, meshTime
```

```
c REMESH declarations (used in subs.f):
      common /Fmeshdata/    Fmeshold, xmeshold, nptsold
c external declarations:
      external                  dtime, func
c the following external subroutines define the boundary conditions and
c pde equations
      external              bcdef, pdedef
c used for Jacobian structure output:
      equivalence          (rwork(21), iwk(1))
c TEMPORARY variables
      double precision      dummy, A1, B1, C1, uxnew(npdemx), wtk(npdemx), sum
c temporary: liquid Mass Fractions
      double precision      Ylf, Ylw, dYfdt, dYwdt
      common /Liquidmass/   Ylf, Ylw, dYfdt, dYwdt
c -------------------------------------------------------------------------
c begin executable statements for main()

c -------------------------------------------------------------------------
c part i.
c open files, initialize variables, and set defaults.
c -------------------------------------------------------------------------
c open program files
c mout is a temporary file for convenient output of certain values over time
c default: ts, mp, Qin, Qout, rx(1), Yf, Temp (note: done in lsodes_1d.f)
      open(UNIT=27, file='mout')
      open(UNIT=LINKCK,  status='old', file='cklink',  form='unformatted')
      open(UNIT=LINKMC,  status='old', file='tplink',  form='unformatted')
      open(UNIT=LCADPL,  status='old', file='cad-pl',  form='formatted')
      open(UNIT=LINIT,   status='old', file='initial', form='formatted')
      open(UNIT=LINTEG,  status='new', file='1d_int',  form='formatted')
      open(UNIT=LDATA,   status='new', file='1d_dat',  form='unformatted')
      open(UNIT=LOUT,    status='new', file='1d_out',  form='formatted')
      open(UNIT=LRESULT, status='new', file='result',  form='formatted')
c cad-ptg is an output of the pressure, temperature, and gamma over CAD
c (useful for engine-type simulations)
      open(UNIT=21, file='cad-ptg')
c initialized CHEMKIN and TRANSPORT link files
      call ckinit(leniwk, lenrwk, lencwk, LINKCK, LOUT, ickwrk, rckwrk, cckwrk)
      call mcinit(LINKMC, LOUT, leniwk, lenrwk, imcwrk, rmcwrk)
      call ck_initial_knui(ickwrk)
c set values for constants
      gasconst = 8.31431d3                    /* unit is J/(Kmol*k) */
c initialized variables
      stats(1) = 0.0d0
      stats(2) = 0.0d0
      stats(3) = 0.0d0
      time_ini = 0.0d0
      iteration = 0
      totime = 0
c default is turbulence ON
      turb = 1
c default in chemistry ON
      chem = 1
      timeold = 0.0d0
      vrpold = 100.0d0
      do i = 1, npts
        rx(i) = 0.0d0
        Fmeshold(i) = 0.0d0
      enddo
      do k = 1, npde
        a(k) = 0.0d0
        b(k) = 0.0d0
      enddo
```

```
      ormesh = .true.
      more = .true.
      ts = 0.0d0
c default tolerances:
c     atol(1) = 1.0d-15
c     rtol = 5.0d-3
c default (droplet):
c     atol(1) = 1.0d-15
c     rtol = 1.0d-4
      atol(1) = 1.0d-15
      rtol = 1.0d-4
c first output time (increases by one order of magnitude until tinc):
      tsmallest = 1.0d-06
c minimum radius for calculation
      minrp = 1.0d-07
c LSODES variables
      itask = 1
      istate = 1
      iopt = 1
      iwork(6) = 1e10
      itrace = 0
      ipminf = 0
      itol = 1
c end LSODES variables
c set defaults
      corder = 0
      Tw = 361.0
      remeshflag = 1
      nptstotry  = 100
      mindistance = 1.0e-05
      maxdiff = 0.10
c initilized liquid points data (may change due to input file):
      fuel = 1
      nptsl = 30
      npdel = 2

c ------------------------------------------------------------------------------
c part ii.
c Read in all input data and restructure solution vector to include extra
c equations (such as for radius, velocity, liquid properties, etc.).  Each
c extra variable is put (in order) after the temperature variable for each
c point in the domain.  To add extra equations, set 'nxeqn' to the required
c number of extra equations and then add the variable after loop level 2
c (changes can be made where indicated by *).
c Default: 1 extra equation for rp
c ------------------------------------------------------------------------------
c read in initial and boundary conditions
      call read_initial(LINIT, LOUT, cckwrk, nspecies, npts, x, u,
     1                  rp, corder, Tw, turb, chem, ignite, nptsl, fuel, rtol)
c record original number of liquid points:
      nptslor = nptsl
c read in pressure data
      call read_cadpl(LCADPL, LOUT)
c check for excessive number of points
      if ( npts .gt. nptsmx ) then
         write (6,*) 'Number of points: ', npts,
     1               ' exceeds maximum limit: ', nptsmx
         stop
      endif
c (*) set the number extra (non species or temperature) equations
      nxeqn = 1
c shift all variables to accomodate extra equations in structure (default: rp)
      npde = nspecies + 1 + nxeqn
```

```
c loop level 1 for assigning stopping point for all values ahead of npde*i
      do i = 1, npts
        k = npde*i
c loop level 2 for shifting up all values for this extra equation
        do j = npts*(nspecies+1)+(nxeqn*i), k, -1
          u(j) = u(j-nxeqn)
        enddo
      enddo
      do i = 1, npts
c (*) extra equations inserted here: the integer after '+1+' should increase
c for each new equation.
c FIRST EXTRA EQUATION: radius
        u(npde*(i-1)+nspecies +1+ 1) = rp
      enddo
      neqng = npde*npts
      neqn = npde*npts + npdel*nptslor

c ---------------------------------------------------------------------------
c part iii.
c Determine initial conditions and profiles for coordinate transform and
c initial data output.
c ---------------------------------------------------------------------------
c determine initial pressure
      call interpolate(time_ini,pressure,dpdt,length,avespeed,tconst)
      if (nptsl .gt. 0) then
c get liquid properties (A1, B1, C1)
        call getlprop (u(npde), pressure,
     1                 dummy, dummy, dummy, dummy, A1, B1, C1)
c set all evaporation/condensation species to phase equilibrium at point 1
        call equil(u, nspecies, npde, A1, B1, C1)
      endif
c determine initial molecular weight, temperature, and density profiles
      do i = 1, npts
        do k = 1, nspecies
           y(k) = u(npde*(i-1) + k)
        enddo
        call ck_mmwy(y, ickwrk, rckwrk, wtm(i))
        Temp(i) = u(npde*(i-1) + nspecies+1)
        rho(i) = pressure*wtm(i)/(gasconst*Temp(i))
      enddo
c determine initial profiles for liquid phase
      if (nptsl .gt. 0)
     1      call liquidpr (u, rp, Temp(1), neqng, npdel, nptsl, corder, Tw)

c ---------------------------------------------------------------------------
c part iv.
c Perform initial coordinate transfom from spatial coordinate to density-
c weighted coordinate system.  The transformed (x) and physical (rx)
c coordinates may be used to solve the problem.  The mapping of the data and
c remeshing is done on the transformed coordinates (x) and the program output
c in in physical coordinates (rx).
c ---------------------------------------------------------------------------
      do i = 1 , npts
c adopt physical coordinates from input x
        rx(i) = x(i) + rp
c transform x into new coordinates, 'corder' is the order of coordinate system
c x = integral (rp, r, r^(corder)*rho*dr')
        if (i .eq. 1) then
          x(i) = 0.0
        else
          x(i) = x(i-1) + (rx(i)**corder*rho(i)+rx(i-1)**corder*rho(i-1))*0.5
     1                  * (rx(i)-rx(i-1))
        endif
```

```
      enddo
c initialize remaining remeshing variables dependent on input
      nptsnew = npts
      do i = 1, npts
        xjnew(i) = x(i)
        xmeshold(i) = x(i)
      enddo
      nptsold = npts

c -----------------------------------------------------------------------
c part v.
c Output first point before starting problem integration.
c The output is written to file LDATA as xout for the coordinate axis and
c uout for the species concentrations.  These arrays can be changed to allow
c any type of output desired.
c -----------------------------------------------------------------------
c convert to molar fractions
      do i = 1, neqng
        ux(i) = u(i)
      enddo
      do i = 0, npts-1
        call ck_y2x (ux(npde*i+1), ickwrk, rckwrk,
     1               ux(npde*i+1))
      enddo
c restore coordinates
      call coord_tran (x, rho, npts, rp, corder, rx)
c get integrated data
      call integrate_x (nspecies, npts, npde, rho, wtm, rckwrk,
     1                  ux, rx, corder, sumspec)
c shift data to accomodate liquid
      do i = 1, (nptslor)
        if (nptsl .gt. 0) then
          xout(i) = rp/(nptsl-1)*(i-1)
          do k = 1, nspecies
            uout((nspecies+1)*(i-1)+k) = 0.0d0
          enddo
          uout((nspecies+1)*(i-1)+nspecies+1) = u(neqng + npdel*(i-1) + 1)
        else
          xout(i) = 0.0d0
          do k = 1, nspecies
            uout((nspecies+1)*(i-1)+k) = ux(k)
          enddo
          uout((nspecies+1)*(i-1)+nspecies+1) = ux(nspecies+1)
        endif
      enddo
      do i = 1, npts
        xout((nptslor)+i) = rx(i)
        do k = 1, nspecies+1
          uout((nspecies+1)*(i-1)+k+(nspecies+1)*(nptslor)) =
     1                ux(npde*(i-1) + k)
        enddo
      enddo
c file output commands
      write(LDATA) npts+nptslor, nspecies+1
      call write_species (LDATA, cckwrk)
      write(LDATA) time_ini
      write(LDATA) (xout(i), i = 1 , (npts+(nptslor)))
      write(LDATA) (uout(i), i = 1, ((npts+(nptslor))*(nspecies+1)))
      write(LDATA) pressure
      write(LINTEG,1000) time_ini, (sumspec(i), i = 1 , nspecies)
c points screen update
      write (*,*) 'Number of original mesh points: ', npts
```

```
c ------------------------------------------------------------------------
c part vi.
c Begin time integration process.  This code uses a version of LSODES that has
c been modified to help in the remeshing process.  The process begins with a
c remeshing of the domain (if option is turned on).  Once a mesh has been
c established, the size of the problem may change due to a difference in the
c number of mesh points.  The Jacobian structure is then calculated for the
c current solution vector.  Once a structure has been determined the integrator
c (LSODES is the default) is called and provided with the jacobian and the name
c of the function 'func' that defines the ordinary differential equations.
c Integration is allowed to proceed until LSODES determines that a new Jacobian
c needs to be calculated to improve convergence.  At this point (in a modified
c version of LSODES) the remeshing procedure is called again and it determines
c if the mesh function has changed sufficiently to justify a new meshing.  If
c so, the program quits LSODES (with error message -8) and a new mesh is then
c calculated.  The reason the remeshing routine is called inside LSODES is that
c after reseting to a new mesh, a new Jacobian must always be calculated.  It
c is therefore much more effective to check before unnavoidable Jacobian
c calculations to see if the mesh could be improved before the calculation
c is actually carried out.
c ------------------------------------------------------------------------
      tout = 0.0

c ============================ main loop begins ================================

      do 30 while (more)
c update iteration number if not a new Jacobian exit
        if (istate .ne. -8) iteration = iteration + 1
c increment output time if not a new Jacobian exit.  Output time increases by
c one order of magnitude from the smallest value to the first requested time
c increment.
        if (istate .ne. -8) then
          if (tout .lt. tsmallest) then
            tout = tsmallest
          elseif (tout .lt. tinc/10.0) then
            tout = tout * 10.0
          elseif (tout .lt. tinc) then
            tout = tinc
          else
            tout = tout + tinc
          endif
        endif
        elapsed = dtime(time)
c assign weights to remesh coefficients (default)
        maxavg = 0.0
        do k = 1, npde
          do i = 1, npts
            S(i) = u(npde*(i - 1) + k)
          enddo
          call getavg(npts, S, avg)
          avgarray(k) = avg
          if (avg .gt. maxavg) maxavg = avg
        enddo
        do k = 1, nspecies+1
          if (abs(avgarray(k)) .gt. 1.0e-05) then
            a(k) = 1.0d0
            b(k) = 2.0d0
          else
            a(k) = 0.0d0
            b(k) = 0.0d0
          endif
        enddo
        a(nspecies+1) = real(nspecies)*a(nspecies+1)/3.0
```

```
        b(nspecies+1) = real(nspecies)*b(nspecies+1)/3.0
        a(npde) = 0.0d0
        b(npde) = 0.0d0
c        a(1) = 20.0
c        b(1) = 20.0
c -----------------------------------------------------------------------
c   part via. - Remesh update
c   If remesh option on, call remesh if first time through loop.  Otherwise it
c   will have been called inside LSODES and a new mesh xjnew(*) will exist.
c   It is therefore only necessary to update the problem onto this new mesh
c   at this point.
c -----------------------------------------------------------------------
c call remeshing routine and reset mesh for either the original mesh
c or a new Jacobian exit
        if ((remeshflag) .and. ((istate .eq. -8)
     1                    .or. (ormesh))) then
c save liquid temperature data
          do i = 1, nptslor*npdel
            Temp(i) = u(neqng + i)
          enddo
c for first time thgouth loop
          if (ormesh) then
            nptsnew = nptstotry
            mindis = mindistance
            call remesh (npts, npde, x, u, xjnew, nptsnew,
     1                   a, b, mindis, maxdiff, different)
          endif
c either a new mesh has been just obtained or it was obtained in LSODES;
c update the problem definition for new mesh here
          call interpall (npts, nptsnew, npde, x, xjnew, u)
c if change in number of points:
          neqng = npde*npts
          neqn = npde*npts + npdel*nptslor
c restore liquid data
          do i = 1, nptslor*npdel
            u(neqng + i) = Temp(i)
          enddo
          istate = 1
          stats(1) = stats(1) + iwork(11)
          stats(2) = stats(2) + iwork(12)
          stats(3) = stats(3) + iwork(13)
          ormesh = .false.
          write (*,*) 'Points differ at iteration no.: ', iteration
          write (*,*) 'Number of new points: ', npts
        endif

c -----------------------------------------------------------------------
c   part vib. - Calculate new jacobian structure
c   If the problem has been redefined (remesh has changed the meshing of the
c   spatial domain and remapped the variables) then a new Jacobian structure
c   must be provided to LSODES.  The default structure is a block diagonal
c   that includes all the species for each point and the two neighbouring
c   points to each side on the domain.  The definition of the Jacobian
c   structure is as follows: there are two vectors, 'ia' and 'ja', in iwork.
c   'ia' represents the number index in the 'ja' array where the reference
c   to non-zero rows for each column begins.  ja, includes the number of all
c   non-zero row for the specified column.  Each column is specified by its
c   index in 'ia'.
c -----------------------------------------------------------------------
        if (istate .eq. 1) then

c -----------------------------------------------------------------------
c part bi.
```

```
c the first array contains the starting positions in the second array
c for each column of the jacobian
c ------------------------------------------------------------------------
          ia_len = npts*npde + nptsl*npdel
          write(*,*) 'ia_len: ', ia_len
c location of first column index in iwork
          iwork(30 + 1) = 1
c assign column indexes for first point
          do i = 2, npde+1
            iwork(30+i) = iwork(30+i-1) + 2*npde
          enddo
c assign column indexes for intermediate points
          do i = npde+2, (npts-1)*npde+1
            iwork(30+i) = iwork(30+i-1) + 3*npde
          enddo
c assign column indexes for last point
          do i = (npts-1)*npde+2, npts*npde+1
            iwork(30+i) = iwork(30+i-1) + 2*npde
          enddo
        if (nptsl .gt. 0) then
c Liquid equations
c assign column indexes for first point
          do i = neqng+2, ((neqng+2)+npdel-1)
            iwork(30+i) = iwork(30+i-1) + 2*npdel
          enddo
c assign column indexes for intermediate points
          do i = ((neqng+2)+npdel), ((neqng+2)+npdel+(nptsl-2)*npdel-1)
            iwork(30+i) = iwork(30+i-1) + 3*npdel
          enddo
c assign column indexes for last point
          do i = ((neqng+2)+npdel+(nptsl-2)*npdel),
     1               ((neqng+2)+npdel+(nptsl-2)*npdel+npdel)
            iwork(30+i) = iwork(30+i-1) + 2*npdel
          enddo
        endif
c ------------------------------------------------------------------------
c part bii.
c the second array contains the indices of the filled rows for each
c column of the jacobian
c ------------------------------------------------------------------------
          ja_len = npde*npde*npts + npde*npde*(npts-1)*2
        if (nptsl .gt. 0) then
          ja_len = ja_len +  npdel*npdel*nptsl + npdel*npdel*(nptsl-1)*2
        endif
          write(*,*) 'ja_len: ', ja_len
c location of first row index in iwork
          iwork( 30 + ia_len + 1 ) = ja_len+1
c assign row indexes for first point
          do i = 1, npde
            do j = 1, 2*npde
              iwork( 30+ia_len+iwork(30+i)+j ) = j
            enddo
          enddo
c assign row indexes for intermediate points
          k = 1
          do i = npde+1, npde*(npts-1)
            if ( (i - npde*(i/npde)) .eq. 1 ) then
              k = k + 1
            endif
            do j = 1, 3*npde
              iwork( 30+ia_len+iwork(30+i)+j ) = j + npde*(k-2)
            enddo
          enddo
```

```
c assign row indexes for last point
          do i = npde*(npts-1)+1, npts*npde
            do j = 1, 2*npde
              iwork( 30+ia_len+iwork(30+i)+j ) = j + npde*(npts-2)
            enddo
          enddo
        if (nptsl .gt. 0) then
c Liquid equations
c assign row indexes for first point
          do i = neqng+1, ((neqng)+npdel)
            do j = 1, 2*npdel
              iwork( 30+ia_len+iwork(30+i)+j ) = j
            enddo
          enddo
c assign row indexes for intermediate points
          k = 0
          l = 0
          do i = ((neqng)+npdel+1), ((neqng)+npdel+(nptsl-2)*npdel)
            l = l + 1
            if ( (l - npdel*(l/npdel)) .eq. 1 ) then
              k = k + 1
            endif
            do j = 1, 3*npdel
              iwork( 30+ia_len+iwork(30+i)+j ) = j + npdel*(k-1)
            enddo
          enddo
c assign row indexes for last point
          do i = ((neqng)+npdel+1+(nptsl-2)*npdel),
     1               ((neqng)+npdel+1+(nptsl-2)*npdel+npdel)
            do j = 1, 2*npdel
              iwork( 30+ia_len+iwork(30+i)+j ) = j + npdel*(nptsl-2)
            enddo
          enddo
        endif
c ------------------------------------------------------------------------------
c part iii.
c optional output file
c ------------------------------------------------------------------------------
c         open( unit = 14, file = 'structFile.txt' )
c         write( 14, '(I10,I10)' ) ia_len, ja_len
c         do i = 1, ia_len
c           write ( 14, '(I10)' ) iwork( 30+i )
c         enddo
c         do i = 1, ja_len
c           write (14, '(I10)' ) iwork( 30+ia_len+1+i )
c         enddo
c         close( 14 )
c         stop
        endif

c ------------------------------------------------------------------------------
c   part vic. - Call to LSODES
c ------------------------------------------------------------------------------
c ------------------------------------------------------------------------------
        call lsodes(func,neqn,u,ts,tout,itol,rtol,atol(1),itask,
     1              istate, iopt, rwork, lrw, iwork, liw, jac, mf)
c ------------------------------------------------------------------------------
c ------------------------------------------------------------------------------

c ------------------------------------------------------------------------------
c   part vid. - Data out during integration
c ------------------------------------------------------------------------------
c update molecular weight and density profiles
```

```
        do i = 1, npts
          do k = 1, nspecies
             y(k) = u(npde*(i-1) + k)
          enddo
          call ck_mmwy(y, ickwrk, rckwrk, wtm(i))
          Temp(i) = u(npde*(i-1) + nspecies+1)
          rho(i) = pressure*wtm(i)/(gasconst*Temp(i))
        enddo
        rp = rx(1)
c convert solution vector to molar fractions
        do i = 1, neqng
          ux(i) = u(i)
        enddo
        do i = 0, npts-1
          call ck_y2x (ux(npde*i+1), ickwrk, rckwrk,
     1                 ux(npde*i+1))
        enddo
c restore coordinates
c        call coord_tran (x, rho, npts, rp, corder, rx)
c get integrated data
        call integrate_x (nspecies, npts, npde, rho, wtm, rckwrk,
     1                    ux, rx, corder, sumspec)
c shift data to accomodate liquid
      do i = 1, (nptslor)
        if (nptsl .gt. 0) then
          xout(i) = rp/(nptslor-1)*(i-1)
          do k = 1, nspecies
            uout((nspecies+1)*(i-1)+k) = 0.0d0
          enddo
          uout((nspecies+1)*(i-1)+nspecies+1) = u(neqng + npdel*(i-1) + 1)
        else
          xout(i) = 0.0d0
          do k = 1, nspecies
            uout((nspecies+1)*(i-1)+k) = ux(k)
          enddo
          uout((nspecies+1)*(i-1)+nspecies+1) = ux(nspecies+1)
        endif
      enddo
      do i = 1, npts
        xout((nptslor)+i) = rx(i)
        do k = 1, nspecies+1
          uout((nspecies+1)*(i-1)+k+(nspecies+1)*(nptslor)) =
     1                 ux(npde*(i-1) + k)
        enddo
      enddo
c file output commands
        write (LRESULT,*) iteration, ':', ts,  '   order: ', iwork(14)
        write (LRESULT,*) 'istate = ', istate
        if ((tout. ge. tsmallest) .and. (istate .ne. -8)) then
          write(LDATA) tout
          write(LDATA) npts+nptslor
          write(LDATA) (xout(i), i = 1 , (npts+(nptslor)))
          write(LDATA) (uout(i), i = 1, ((npts+(nptslor))*(nspecies+1)))
          write(LDATA) pressure
          write(LINTEG,1000) tout, (sumspec(i), i = 1,nspecies)
c writing mout information
          write(27,9898) ts, mold, Qin, Qout, rx(1), u(1), Temp(1)
 9898     format (4(e18.6),e21.10,2(e18.6))
          call flush(27)
        else
c if lsodes exit not on desired output time step:
          write (*,*) '(not writing to file)'
        endif
```

```
        call flush(LDATA)
        call flush(LINTEG)
        call flush(LRESULT)
c screen out for new Jacobian exit
        if (istate .eq. -8) then
          write (*,*) 'Quitting before recalculating jacobian...'
        endif
c file and screen out for iteration information
        write (6,*) ts, '   order: ', iwork(14)
        elapsed = dtime(time)
        totime = totime + elapsed
c screen out for LSDOES error
        if ((istate .ne. -8) .and. (istate .lt. 0)) then
          write (6,*) 'LSODES ERROR! istate = ', istate
        endif
c outter loop variable
      more = tout .lt. tmax

   30 continue

c ============================== main loop ends ================================

c -------------------------------------------------------------------------------
c part vii. - Printing integration statistics at the end of run
c -------------------------------------------------------------------------------
c update statistics
      stats(1) = stats(1) + iwork(11)
      stats(2) = stats(2) + iwork(12)
      stats(3) = stats(3) + iwork(13)
c file out
      write (LRESULT,*) (u(npde*(i-1)+nspecies+1), i = 1 , npts)
      write (LRESULT,*) ' '
      write (LRESULT,*) '1) Steps:          ', stats(1)
      write (LRESULT,*) '2) f evals:        ', stats(2)
      write (LRESULT,*) '3) jac eval:       ', stats(3)
      write (LRESULT,*) '4) order (last):   ', iwork(14)
      write (LRESULT,*) '5) lenrw:          ', iwork(17), lrw
      write (LRESULT,*) '6) leniw:          ', iwork(18), liw
      write (LRESULT,*) '7) nnz:            ', iwork(19)
      write (LRESULT,*) ' '
      write (LRESULT,*) '1d code done.'
      write (LRESULT,*) ' '
c screen out
      write (6,*) ' '
      write (6,*) '1) Steps:          ', stats(1)
      write (6,*) '2) f evals:        ', stats(2)
      write (6,*) '3) jac eval:       ', stats(3)
      write (6,*) '4) order (last):   ', iwork(14)
      write (6,*) '5) lenrw:          ', iwork(17), lrw
      write (6,*) '6) leniw:          ', iwork(18), liw
      write (6,*) '7) nnz:            ', iwork(19)
      write (6,*) ' '
      write (6,*) '1d code done.'
      write (6,*) ' '

      stop

c format statements
 1000 format (<nspecies+1>(1pe13.4))
 9999 format (e19.9,f19.4)
 9997 format (e19.9,e19.4,e19.4,e19.4)

      end
```

```
* ************************** END OF MAIN PROGRAM ****************************

c ---------------------------------------------------------------------------

* ********************** BEGINNING OF MAIN SUBROUTINES ***********************


c ***************************************************************************
      subroutine coord_tran (x, rho, npts, rp, corder, rx)
c ***************************************************************************
c *                              COORD_TRAN                                 *
c *     This subroutine performs a coordinate transformation that restores  *
c *  the transformed coordinat x back to original coordinates rx.           *
c *                                                              03/08/1999 *
c ***************************************************************************
c
c  Input variables:
c
c  x[npts]    = transformed coordinate vector (kg)
c  rho[npts]  = density at all points (kg/m^3)
c  npts         = number of points in spatial domain
c  rp           = initial location of transform (droplet radius) (m)
c  corder       = coordinate order
c                 corder = 0 : flat coordinates (flat wall)
c                 corder = 1 : cylindrical coordinates
c                 corder = 2 : spherical coordinates (droplet)
c
c  Output variables:
c
c  rx[npts]   = physical spatial coordinate vector (m)
c                                                                  08/25/98
      implicit none
c argument variables:
      double precision      x(*), rho(*), rp
      integer               npts, corder
      double precision      rx(*)
c local variables:
      integer               i
      double precision      intg, n
c end of variable declarations

      n = real(corder)
      rx(1) = rp
      intg = 0.0
      do i = 2 , npts
        intg = intg + (((1.0/rho(i))+(1.0/rho(i-1)))*0.5)*(x(i)-x(i-1))
        rx(i) = (rp**(n+1) + (n+1) * intg)**(1.0/(n+1))
      enddo

      return
      end

c end of subroutine coord_tran() ******************************************

c ***************************************************************************
      subroutine write_u (u, npde, npts, stopflag, pauseint)
c ***************************************************************************
c *                                WRITE_U                                  *
c *     This subroutine can be called at any time to write out the contents *
c *  of the solution vector on the screen.                                  *
c *                                                              03/08/1999 *
c ***************************************************************************
```

```
c
c  Input variables:
c
c  u[npde*npts]      = solution vector
c  npde              = number of pdes in vector
c  npts              = number of coordinate points
c  stopflag  = stops program execution after all points if '1'
c  pauseint  = pauses execution after pauseint points
c                 set to '0' for no pauses
c
c  Output variables:
c
c  none
c                                                                    08/28/98
      implicit none
c argument variables:
      double precision      u(*)
      integer               npde, npts, stopflag, pauseint
      integer               i, k
c end of variable declarations

      do i = 1, npts
        if (pauseint) then
          if (i/pauseint*pauseint-i .eq. 0) pause
        endif
        write (*,*)
        write (*,*) 'point: ', i
        do k = 1, npde
          write (*,*) k, u(npde*(i-1) + k)
        enddo
      enddo
      if (stopflag) stop

      return
      end

c end of subroutine write_u() ************************************************

c ****************************************************************************
      subroutine integrate_x(nspecies, npts, npde, rho, wtm, rckwrk,
     1                       ux, rx, corder, sumspec)
c ****************************************************************************
c *     This subroutine integrates the species concentrations in the given   *
c *  domain.  Results are returned on a molar basis.                         *
c *                                                          03/08/1999     *
c ****************************************************************************
c
c  Input variables:
c
c  nspecies  = number of species
c  npts              = number of coordinate points
c  npde              = number of pdes in solution vector
c  rho       = density profile for all points
c  wtm       = molecular weights for all points
c  rckwrk    = real work array from CHEMKIN subroutines
c  ux        = solution vector IN MOLAR CONCENTRATIONS
c  rx        = physical (non-transformed) coordinate system
c  corder       = order of coordinate system
c
c  Output variables:
c
c  sumspec[nspecies] = integration of species:
c            flat: per unit area
```

```
c            cylindrical: per unit length
c            spherical: total
c
c     Note: it is necessary to transform the solution vector u(*) into
c  a molar-based vector ux(*) before calling the routine.  Otherwise the
c  returned values will be mass-based.
c                                                                  08/28/98

      implicit none
c argument variables:
      integer                nspecies, npts, npde, corder, i, k
      double precision       rho(*), wtm(*), ux(*), rx(*), rckwrk(*),
     1                       sumspec(*)
      double precision       factor
c end of variable declarations

c initialize sums
      do k = 1 , nspecies
        sumspec(k) = 0.0d0
      enddo
c determine correct coordinate factor
      factor = 1.0
      if (corder .eq. 1) factor = 3.14159
      if (corder .eq. 2) factor = 4 * 3.14159
c integrate
      do k = 1, nspecies
        do i = 2, npts
c if desired constraint on integration limit, insert where commented c'
c'          if (rx(i) .le. 0.01) then
            sumspec(k) = sumspec(k) + 0.5*(
     1                 (rho(i)*ux(npde*(i-1)+k)/wtm(i)*rx(i)**corder)+
     2                 (rho(i-1)*ux(npde*(i-2)+k)/wtm(i-1)*rx(i-1)**corder)
     3                                    )*(rx(i)-rx(i-1))
c'          else
c'            if (k .eq. 1) write (*,*) rx(i)
c'          endif
          enddo
          sumspec(k) = factor * sumspec(k)
        enddo

        return
        end

c end of subroutine integrate_x() ******************************************


*------------------------------------------- Interpolate pressure
*==============================================================================
* Interpolate pressure corresponding to the input time
*==============================================================================

      subroutine interpolate(time,p,dpdt,l,barsp,t)

      implicit none
      double precision time, p, dpdt, l, barsp,t
      double precision ca, stroke
      parameter (stroke = 8.8d-2)
      integer i
      integer cadInit
      double precision speed(1),cad(720), press(720), length(720)
      COMMON /CADPLDAT/ speed, cad, press, length, cadInit
```

```
*
* Convert input (time) into corresponding crank angle
*
      If(time .EQ. 0.0d0) then

c        i = 1
c        do while ( (cadInit .ge. cad(i) .and. cadInit .le. cad(i+1))
c     1             .and. (i < 720) )
c           i = i+1
c        enddo

         p = press(1)
         dpdt = (press(2)-press(1))/(cad(2)-cad(1))
     :   *(speed(1)/60.0*360.0)
         l = length(1)
         barsp = speed(1)/60.0*2.0*stroke
         t = (cad(1)-375)/(speed(1)/60.0*360.0)
         return
       else
         ca = cad(1) + time * (speed(1)/60.0*360.0)
       endif

*
* Interpolation
*
       do i = 1 , 719
          if(ca .GE. cad(i) .AND. ca .LE. cad(i+1)) then
            p = press(i) + (ca-cad(i))/(cad(i+1)-cad(i))*
     :          (press(i+1)-press(i))
            dpdt = (press(i+1)-press(i))/(cad(i+1)-cad(i))
     :             *(speed(1)/60.0*360.0)
            l = length(i) + (ca-cad(i))/(cad(i+1)-cad(i))*(length(i+1)
     :          -length(i))
            goto 100
          endif
       enddo

       write (*,*)
      : 'Current crank angle degree cannot be bracketed by input crank angle
      : list!'

       stop

       p = press(719)
       dpdt = 0.0d0
       l = length(719)

  100 continue

*
* Calculate average engine speed and time constant used for turbulent model
*

       barsp = speed(1)/60.0*2.0*stroke
       t = (cad(1)-375)/(speed(1)/60.0*360.0)

       return
       end


c ***************************************************************************
       subroutine func       (neqn, ts, u, udot)
c ***************************************************************************
```

```
c *                                  FUNC                                  *
c *     func() is the function subroutine that is called by the ODE solver  *
c *  (eg. LSODES) to define the time derivative of all the differential     *
c *  equations involved in the problem.  The routine solves the conservation *
c *  equations for species, temperature, and boundary location.  Pressure is *
c *  only an initial condition and is therefore also solved and the output  *
c *  stored in a file.  The routine assumes an adiabatic expansion process; *
c *  convection, diffusion, and reaction terms for species; convection,     *
c *  conduction, and reaction for heat; and the possibility of an evaporating *
c *  boundary interface.  External routines, such as pdedef() and bcdef()   *
c *  are called to define the necessary terms for the respective equations.  *
c *  The algorithm currently used is a method of lines for discretizing the *
c *  problem domain with a Backward Difference Formula ODE solver to solve  *
c *  resulting equations.                                                   *
c *                                                          03/08/1999     *
c ***************************************************************************
c
c  Input variables:
c
c  neqn            = TOTAL number of ordinary differential equations
c  ts        = current time (set by integrator) (s)
c  u[neqn]      = solution vector (current values)
c
c  Output variables:
c
c  udot[neqn]       = unsteady term for solution vector
c                                                                  08/26/98
      implicit none
c argument variables:
      integer              neqn, nptsl, npdel, nptslor
      double precision     ts, u(*)
      double precision     udot(*)
c common variables:
      integer              nptsmx, npts, nspecies, nxeqn, npdemx
      parameter           (npdemx = 80, nptsmx = 205)
      integer              turb, corder, chem, fuel
      double precision     timeold, timesucc, x(nptsmx), rx(nptsmx),
     1                     gasconst, tcurr,
     2                     pressure, dpdt, length, avespeed, tconst,
     3                     minrp
      common /timedif/     timeold, timesucc
      common /ptinfo/      x, rx, npts, nspecies, nxeqn
      common /pressdat/    pressure, dpdt, length, avespeed, tconst
      common /misc/        gasconst, minrp, turb, corder, chem, fuel
      common /liquid/      nptsl, npdel, nptslor
c CHEMKIN variables:
      integer              leniwk, lenrwk
      parameter           (leniwk = 7500, lenrwk = 90000)
      integer              ickwrk(leniwk), imcwrk(leniwk)
      double precision     rckwrk(lenrwk), rmcwrk(lenrwk)
      common /ckspi/       ickwrk, imcwrk
      common /ckspr/       rckwrk, rmcwrk
c local variables:
      integer              npde, i, k, neqng
      double precision     rho(nptsmx), Temp(nptsmx), wtm(nptsmx),
     1                     m(npdemx), p(npdemx), q(npdemx), r(npdemx),
     2                     Mtot(nptsmx, npdemx), Mx(nptsmx, npdemx),
     3                     Ptot(nptsmx, npdemx), Qtot(nptsmx, npdemx),
     4                     Rtot(nptsmx, npdemx), Rxx(nptsmx, npdemx),
     5                     y(npdemx), Yx_l(npdemx), Yx_r(npdemx), drpdt,
     6                     mp, vrp, rp, Qin, Qout, mold, Qinold, Qoutold,
     7                     flux_fd(nptsmx), flux_bd(nptsmx), Yold, Tempold,
     8                     dTdt_l, Yt_l(nptsmx), vrpold, Tldot(nptsmx),
```

```
     9                       dTldr_o, Tr_l
      common /fluxdat/       tcurr, mp, Qin, Qout, mold, Qinold, Qoutold,
     1                       Yold, Tempold, vrpold
c temporary: liquid Mass Fractions
      double precision       Ylf, Ylw, dYfdt, dYwdt, errYt
      common /Liquidmass/    Ylf, Ylw, dYfdt, dYwdt

c end of variable declaration

c ---------------------------------------------------------------------------
c part i.
c Begin routine by setting the appropriate number of pde's, the updated volume
c and pressure and the current molecular weights, temperatures, and densities
c for the domain.  Also update the physical coordinate system.
c ---------------------------------------------------------------------------
      npde = nspecies + 1 + nxeqn
      neqng = npde*npts
c units for pressure are N/m^2:
      call interpolate (ts, pressure, dpdt, length, avespeed, tconst)
      do i = 1, npts
c y(*) for current molecular weight calculation only
        do k = 1 , nspecies
          y(k) = u(npde*(i - 1) + k)
        enddo
c units for mean molar weight are kg/Kmole:
        call ck_mmwy(y, ickwrk, rckwrk, wtm(i))
c unit of temperature is K:
        Temp(i) = u(npde*(i - 1) + nspecies+1)
c units of density are kg/m^3:
        rho(i) = pressure*wtm(i)/(gasconst*Temp(i))
      enddo
      rp = u(nspecies+1+1)
      if (rp .le. minrp) then
        if (corder. eq. 2) then
          write (*,*) '* Minimum rp reached: ', rp
          stop
        else
          nptsl = 0
        endif
      endif
      call coord_tran(x, rho, npts, rp, corder, rx)

c ---------------------------------------------------------------------------
c part ii.
c Determine required interface variables: mass flux out, spatial derivative
c at interface, and rate of boundary movement.  This part is also used for
c defining the appropriate boundary condition terms.
c ---------------------------------------------------------------------------
      if (nptsl .gt. 0) then
c preliminary data required for bcdef
        dTldr_o = ( u(neqng + npdel*(nptsl-1) + 1)
     1              - u(neqng + npdel*(nptsl-3) + 1) )
     2            / (rp/(nptsl-1)*2.0)
c determine terms at the boundary: eigenvalue + boudnary conditions
        call bcdef  (u, ts, Temp, wtm, rho, rp, Yt_l, dTldr_o, nptsl, Tr_l,
     1               Yx_l, Yx_r, dTdt_l, drpdt, vrp, mp, Qin, Qout, vrpold)
c solve for liquid-phase equations (gas-phase done in pdedef)
        call liqdef (u, Tldot, rp, neqng, nptsl, npdel, dTdt_l, Tr_l, corder,
     1               pressure)
      else
c if no liquid layer
        dTldr_o = 0.0d0
        do k = 1, nspecies + 1
```

```
          Yx_l(k) = 0.0d0
          Yx_r(k) = 0.0d0
          Yt_l(k) = 0.0d0
        enddo
        vrpold = 0.0d0
        dTdt_l = 0.0d0
        drpdt = 0.0d0
        vrp = 0.0d0
        mp = 0.0d0
        Qin = 0.0d0
        Qout = 0.0d0
      endif

c -----------------------------------------------------------------------------
c part iii.
c Define the required pde coefficients for the gas phase equations which have
c the form: udot(i,k) = (Mtot(i,k)*du(i,k)/dx + d(Rtot(i,k)*du(i,k)/dx)/dx +
c Qtot(i,k)) / Ptot(i,k).  First the coefficients are aquired from pdedef()
c for all species at each point as p, q, and r.  Then they are assigned to
c 2-dimensional arrays:
c Ptot(i,k) = density (the same for all species) at gridpoint i;
c Qtot(i,k) = chemical production rate for species k at gridpoint i;
c Rtot(i,k) = diffusion coefficent (mass) for species k at gridpoint i.
c -----------------------------------------------------------------------------
      do i = 1, npts
c update y(*) for all k (from complete u and previous T) at x(i):
        do k = 1, nspecies+1
          y(k) = u((i-1)*npde + k)
        enddo
c define all coefficients at point i
        call pdedef (y, ts, rx(i), Temp(i), wtm(i), rho(i),
     1               rp, mp, nspecies, m, r, q, p)
c define 2-dimensional arrays
        do k = 1, nspecies+1
          Mtot(i,k) = m(k)
          Rtot(i,k) = r(k)
          Qtot(i,k) = q(k)
          Ptot(i,k) = p(k)
        enddo
      enddo

c -----------------------------------------------------------------------------
c part iv.
c Calculation of convective (1/s) and diffusion fluxes (kg/s) for
c (M*dYdx) and d/dx(-D(i)*dY(i)/dx) terms.  Diffusion fluxes are calculated
c at the boundary and intermediate points.  Boundary conditions are set in
c external subroutine called in part ii (bcdef).
c -----------------------------------------------------------------------------
      do k = 1, nspecies+1
c calculation of convective terms
        Mx(1,k) =    Mtot(1,k) * Yx_l(k)
c if statement for upwind differencing
        do i = 2, npts-1
          if (Mtot(i,k) .lt. 0.0d0) then
            Mx(i,k) = Mtot(i,k) * (u(npde*i+k)-u(npde*(i-1)+k))/(x(i+1)-x(i))
          else
            Mx(i,k) = Mtot(i,k) * (u(npde*(i-1)+k)-u(npde*(i-2)+k))
     1                 /(x(i)-x(i-1))
          endif
        enddo
        Mx(npts,k) = Mtot(npts,k) * Yx_r(k)

c diffusion flux calculation by forward and backward differences for points
```

```
c not on boundaries
        do i = 2, (npts-1)
          flux_fd(i) = 0.5*(Rtot(i,k)+Rtot(i+1,k))*
     1                 (u(npde*i+k)-u(npde*(i-1)+k))/(x(i+1)-x(i))
          flux_bd(i) = 0.5*(Rtot(i,k)+Rtot(i-1,k))*
     1                 (u(npde*(i-1)+k)-u(npde*(i-2)+k))/(x(i)-x(i-1))
        enddo
c LEFT BOUNDARY diffusion fluxes.  Note: first spatial derivatives at
c the left and right boundaries, Yx_l(*) and Yx_r(*), are set by subroutine
c bcdef()
        flux_fd(1)      = Rtot(1,k) * Yx_l(k)
        flux_bd(npts)   = Rtot(npts,k) * Yx_r(k)

c calculation of diffusion terms
        Rxx(1,k) =    (flux_bd(2)-flux_fd(1))/((x(2)-x(1))/2.0)
        do i = 2, npts-1
           Rxx(i,k) = (flux_fd(i)-flux_bd(i))/((x(i+1)-x(i-1))/2.0)
        enddo
        Rxx(npts,k) = (flux_bd(npts)-flux_fd(npts-1))/((x(npts)-x(npts-1))/2.0)
      enddo

c ---------------------------------------------------------------------------
c part v.
c Calculation of all pde unsteady terms given coefficients from parts i, ii,
c iii, and iv.
c ---------------------------------------------------------------------------
c species and temperature at all points
      do i = 1, npts
        do k = 1, nspecies+1
          udot(npde*(i-1) + k) = (Mx(i,k) + Rxx(i,k) + Qtot(i,k)) / Ptot(i,k)
        enddo
c single equation(s) at all points: rp
        udot(npde*(i-1) + nspecies+1 + 1) = drpdt
      enddo
c measure of deviation from mass conservation due to numerical error
      errYt = 0.0d0
      do k = 1, nspecies+1
        udot(k) = Yt_l(k)
        errYt = errYt + Yt_l(k)
      enddo
c liquid data
      do i = 1, nptslor
        udot(neqng + npdel*(i-1) + 1) = 0.0d0
        udot(neqng + npdel*(i-1) + 2) = drpdt
      enddo
      do i = 1, nptsl
        udot(neqng + npdel*(i-1) + 1) = Tldot(i)
      enddo
c left Temp b.c. :
      udot(nspecies+1) = dTdt_l

c ---------------------------------------------------------------------------
c part vi.
c On screen information and diagnotics.
c ---------------------------------------------------------------------------
c on screen update at the end of an integrator time step
      if ((ts .ne. timeold) .or. (ts .eq. 0.0)) then
c time step information
        write(8,*) ts
        write(6,*) 'ts  = ', ts, '        rp = ', rp
        write(6,*) 'mp  = ', mp, '  Yt error = ', errYt
c gas velocity at wall and concentration information
        write(*,*) 'vrp = ', vrp,'    Y(1,fuel) = ', u(fuel)
```

```
        write(*,*) 'p    = ', pressure
        mold = rho(1)*vrp*rp**(corder)*((4*3.14159)**(corder/2))
        Qinold = Qin
        Qoutold = Qout
        vrpold = vrp
        Yold = u(fuel)
        Tempold = Temp(1)
        Ylf = Ylf + dYfdt*(ts-timeold)
        Ylw = Ylw + dYwdt*(ts-timeold)
        tcurr = ts
c rate of change of each species at the left boundary
        do k = 1, npde
c temperature data (temporary)
          if (k .eq. nspecies+1) then
            write (*,*) '* ', k, udot(k), u(k)
c other (species/temperature/rp) data
          else
            write (*,*) '  ', k, udot(k)
          endif
        enddo
c cad, P, T, and gamma data
        if (timeold .eq. 0)
     1 write (21,*) ('  CAD       ts(sec)          P(atm)        T(K)')
c for every new cad, write output
        if ((int(ts*avespeed/(2*8.8e-2)*360)
     1           .gt. int(timeold*avespeed/(2*8.8e-2)*360))
     2       .or. (timeold .eq. 0)) then
          write (21,1212) 375+ts*avespeed/(2*8.8e-2)*360, ts,
     1                    pressure/101332.0,
     2                    Temp(npts)
 1212        format (f8.1, e14.6, f12.3, f12.1)
c          pause
        endif
c flush output
        call flush(6)
        call flush(8)
        call flush(21)
      endif
c update last successful timestep
      if (ts .gt. timeold) timesucc = timeold
      timeold = ts

      return
      end

c end of subroutine func() ****************************************************
```

DEF.F

```
c Remaining simplifications:
c  1) No water condensation (imposed fractional gasification ratios)
c *************************************************************************
      subroutine bcdef (u, ts, Temp, wtm, rho, rp, Yt_l, dTldr_o, nptsl, Tr_l,
     1                  Yx_l, Yx_r, dTdt_l, drpdt, vrp, mp, Qin, Qout, vrpold)
c *************************************************************************
c *     This subroutine returns the terms defined by the interface of the     *
c *  problem.  These are the boundary conditions at the liquid/solid vs. gas  *
c *  interface.  Returned values include first spatial derivatives of         *
c *  species and first temporal derivatives of temperature at the interface,  *
c *  time derivative of droplet radius (if applicable) and the convective     *
c *  mass flux due to evaporation.  This subroutine can also be used to set   *
c *  boundary conditions for the solid vs. gas interface.                     *
c *                                                          03/08/1999     *
c *************************************************************************
c
c  Input variables:
c
c  u[neqn]    = solution vector
c               (neqn = (nspecies + 1 + nxeqn)*npts)
c  ts         = current time (s)
c  Temp[npts] = current temperature profile
c  wtm[npts]  = gas mean molar weight vector (kg/Kmole)
c  rho[npts]  = gas density vector (kg/m^3)
c  rp         = radius of liquid domain
c  dTldr_o      = derivative on liquid side of second point from interface
c  nptsl      = number of points in the liquid phase
c  vrpold     = eigenvalue of previous time step (used as beginning
c               guess for Netwon Iteration for vrp)
c
c  Output variables:
c
c  Yx_l[nsp+1]      = derivative for species and temperature
c                   at LEFT gas boundary ((K)/kg)
c  Tr_l               = derivative for temperature at RIGHT liquid boundary
c  Yx_r[nsp+1]      = derivative for species and temperature
c                   at RIGHT gas boudnary ((K)/kg)
c  dTdt_l     = time rate of change of temperature at LEFT gas boundary
c  drpdt        = time derivative of radius of droplet (m/s)
c  vrp          = velocity on gas side of interface (m/s)
c  mp         = convection coefficient (mass flux due to evaporation) (kg/s)
c  Qin        = heat transfer into liquid layer (out of domain)
c  Qout       = heat transfer out of liquid layer (into domain)
c
c  Notes:  o Sandia CHEMKIN and TRANSPORT routines are called to provide
c            thermodynamic and transport data for species.
c
c                                                        03/08/99
      implicit none
c argument variables:
      double precision      u(*), ts, Temp(*), wtm(*), rho(*), rp
      double precision      Yx_l(*), Yx_r(*), drpdt, vrp, mp,
     1                      Yt_l(*), Qin, Qout, qdot, fv
c common variables:
      integer           npdemx, nptsmx
      parameter               (npdemx = 80, nptsmx = 205)
      double precision      x(nptsmx), rx(nptsmx)
      double precision      pressure, dpdt, length, avespeed, tconst, pp, dppdt
      double precision      gasconst, timeold, timesucc, minrp, dTemp, Tr_l,
     1                      dXfdp, dXfdT, vrpold
      integer               npts, nspecies, nxeqn
      integer               turb, corder, chem, fuel
```

```
      common /ptinfo/       x, rx, npts, nspecies, nxeqn
      common /pressdat/     pressure, dpdt, length, avespeed, tconst
      common /misc/         gasconst, minrp, turb, corder, chem, fuel
      common /timedif/      timeold, timesucc

c CHEMKIN variables:
      integer           leniwk, lenrwk, lencwk
      parameter               (leniwk = 7500, lenrwk = 90000, lencwk = 500)
      integer               ickwrk(leniwk), imcwrk(leniwk)
      double precision      rckwrk(lenrwk), rmcwrk(lenrwk)
      character             cckwrk(lencwk)*(16)
      common /ckspi/        ickwrk, imcwrk
      common /ckspr/        rckwrk, rmcwrk
      common /ckspc/        cckwrk
c local variables:
      integer               npde, i, j, k, nptsl
      double precision      conmix, a_total(6), molarf(npdemx), y(npdemx),
     1                      diffmc(npdemx), Eqt(npdemx), wi(npdemx),
     2                      A(npdemx), B(npdemx), C(npdemx), D(npdemx),
c 1->3 for points 1->3; 4->6 for points (npts-2)->npts
     3                      R(6, npdemx), Q(6, npdemx), P(6, npdemx)
      double precision         sumwh, hml(npdemx), init_kin, init_tau, init_l,
     1                      kin, turb_diff, cpbar, diffmc_l(npdemx),
     2                      dTdr_l, dTdt_l, a_total_l, A1, B1, C1, pi, dTldr_o,
     3                      Xfdot, Ykdot(npdemx), Yfdot, Xf, sum1, sum2,
     4                      Yfdot2, Al, Al2, dVl, cp_l, lamliquid
      parameter            (pi = 3.14159)
c temporary variables:
      parameter (init_l = 3.0d-3)

c STODE variables:
      integer iownd, ialth, ipup, lmax, meo, nqnyh, nslp,
     1   icf, ierpj, iersl, jcur, jstart, kflag, l, meth, miter,
     2   maxord, maxcor, msbp, mxncf, n, nq, nst, nfe, nje, nqu
      double precision conit, crate, el, elco, hold, rmax, tesco,
     2   ccmax, el0, h, hmin, hmxi, hu, rc, tn, uround
      common /ls0001/ conit, crate, el(13), elco(13,12),
     1   hold, rmax, tesco(3,12),
     2   ccmax, el0, h, hmin, hmxi, hu, rc, tn, uround, iownd(14),
     3   ialth, ipup, lmax, meo, nqnyh, nslp,
     4   icf, ierpj, iersl, jcur, jstart, kflag, l, meth, miter,
     5   maxord, maxcor, msbp, mxncf, n, nq, nst, nfe, nje, nqu
c temporary variables:
      integer               ref
      double precision      cliquid, rholiquid, Lt, e(npdemx), Eq(npdemx),
     1                      ux(npdemx), sum,
     2                      uxnew(npdemx), wtk(npdemx)
c end of variable declarations

      npde = nspecies + 1 + nxeqn

c ---------------------------------------------------------------------------
c part i.
c Aquire thermodynamic and equilibrium data for species involved in
c evaporation/condensation onto liquid layer (often: fuel & oxidizer).
c ---------------------------------------------------------------------------
c get liquid properties:
c liquid-gas phase equilibrium must obey equation:
c ***   X = 10^(A1 - B1/(Tl + C)) / P   ***
c X: gas-phase mole fraction
c Tl: liquid temperature
c P: pressure
c A1, B1, C1: liquid properties
```

```
      call getlprop (Temp(1), pressure, cliquid, rholiquid, lamliquid, Lt,
     1               A1, B1, C1)
c (temporary) impose fractional gasification ratios
      do k = 1, nspecies+1
        e(k) = 0.0
      enddo
      e(fuel) = 1.0
c convert to molar fractions
      do k = 1, nspecies
        ux(k) = u(k)
      enddo
      call ck_y2x (ux(1), ickwrk, rckwrk, ux(1))
      call ck_mk  (ickwrk, rckwrk, wtk)

c -------------------------------------------------------------------------
c part ii.
c Diffusion data at points 2, and 3; diffusion, heat transfer, and reaction
c data for point 1.
c -------------------------------------------------------------------------
c thermo data for points 1, 2, 3, npts-2, npts-1, and npts:
      i = 1
      j = 1
      do 10 while (i .le. npts)
        if (turb .eq. 0) then
          turb_diff = 0.0d0
        else
          init_kin = 1.5*(0.75*avespeed)*(0.75*avespeed)
          init_tau = init_l/sqrt(init_kin)
          kin = abs(init_kin*(1+0.9*(tconst+ts)/init_tau))**(-1.11)
          turb_diff = 0.09*(rx(i)-rp)/(1+(rx(i)-rp)/init_l)*sqrt(kin)
        endif
        do k = 1, nspecies
          y(k) = u(npde*(i-1) + k)
        enddo
        call ck_species_source(pressure, Temp(i), y, ickwrk, rckwrk, wi)
        call ck_y2x(y, ickwrk, rckwrk, molarf)
        call ck_cpbl(Temp(i), molarf, ickwrk, rckwrk, cpbar)
        cpbar = cpbar/wtm(i)
        call ck_hml(Temp(i), ickwrk, rckwrk, hml)
        call ck_react_heat(wi, hml, rckwrk, sumwh)
        call mc_acon(Temp(i), molarf, rmcwrk, conmix)
        call mc_adif(pressure, Temp(i), molarf, rmcwrk, diffmc)
        a_total(j) = (turb_diff + conmix/(rho(i)*cpbar)) * rho(i)*cpbar
c saving data for first point (to use in liquid calculations)
        if (i .eq. 1) then
          do k = 1, nspecies+1
            diffmc_l(k) = diffmc(k)
          enddo
          a_total_l = a_total(j)
          cp_l = cpbar
        endif
c diffusion, chemical, and unsteady coefficient terms for temperature:
        do k = 1, nspecies
          R(j,k) = rx(i)**(2*corder)*rho(i)**2*(diffmc(k)+turb_diff)
          Q(j,k) = wi(k)*real(chem)/rho(i)
          P(j,k) = 1.0
        enddo
          R(j,nspecies+1) = rx(i)**(2*corder)*rho(i)*a_total(j)
          Q(j,nspecies+1) = (dpdt-real(chem)*sumwh)/rho(i)
          P(j,nspecies+1) = cpbar
        i = i + 1
        j = j + 1
        if (i .eq. 4) i = npts-2
```

```
   10 continue
c droplet geometry (volume and surface area)
      if (corder .eq. 2) then
        Al = (4.0*pi*rp**2)
        Al2 = (4.0*pi*(rp - rp/(nptsl-1))**2)
        dVl = 4.0/3.0*pi*(rp**3 - (rp - rp/(nptsl-1))**3)
      else
        Al = 1.0
        Al2 = 1.0
        dVl = (rp/(nptsl-1))
      endif

c call Newton iteration routine to get correct eigenvalue solution
      call getzero( vrp, vrpold, rx, x, qdot, dTdt_l, a_total, Temp,
     1                  rho, cp_l, Al, Al2, dVl, rholiquid, cliquid, lamliquid,
     2                  Yx_l, Yx_r, u, ux, e, rp, diffmc_l,
     3                  mp, wtk, R, Q, pressure, dpdt, Lt,
     4                  A1, B1, C1, dTdr_l, Tr_l, dTldr_o, nptsl,
     5                  npde, fuel, nspecies, corder, fv )

c update Qout term (out/into liquid layer at interface)
      Qin = ((4.0*pi)**(corder/2)*rp**(corder))*a_total(1)*dTdr_l
      Qout = ((4.0*pi)**(corder/2) * rp**(corder) * rho(1) * vrp) * Lt
c calculate Ydot term for first point
      do k = 1, nspecies
          Ykdot(k) = -mp*Yx_l(k)
     1               +(0.5*(R(1,k)+R(2,k))*
     2               (u(npde+k)-u(k))/(x(2)-x(1))
     3               - R(1,k)*
     4               Yx_l(k))
     5               / ((x(2)-x(1))/2.0) + Q(1,k)
      enddo

c calculate rate of change of "radius"
      drpdt = -vrp * rho(1) / (rholiquid)
c assign values to species boundary conditions on the gas phase
      do k = 1, nspecies+1
        Yt_l(k) = Ykdot(k)
      enddo

      return
      end
c end of subroutine bcdef() ************************************************


c ***************************************************************************
      subroutine liqdef (u, Tldot, rp, neqng, nptsl, npdel, dTdt_l, Tr_l,
     1                   corder, pressure)

c ***************************************************************************
c *     This subroutine applied energy conservation (can be modified to also  *
c *  include mass conservation) to liquid layer.                               *
c *                                                              03/08/1999    *
c ***************************************************************************
c
c  Input variables:
c
c  u            = solution vector
c  rp           = radius of liquid layer/droplet
c  neqng        = number of equations in the gas phase
c  nptsl        = number of points in the liquid phase
c  dTdt_l       = time rate of change of temperature at interface
c  Tr_l         = derivative of liquid-phase temperature at interface
```

```
c                 (not required if dTdt_1 given)
c  corder       = geomertric factor
c
c  Output variables:
c
c  Tldot        = time rate of change of temperature at position x in liquid
c
c                                                                    03/08/99
      implicit none
      double precision   u(*), Tldot(*), rp, dTdt_1, Tr_1
      integer            neqng, nptsl, npdel, i, corder
      double precision   Tl(200), c(200), rho(200), lam(200), dummy, R(200),
     1                   xl(200), pressure
c end of variable declaration

c get liquid properties and point distribution
      do i = 1, nptsl
        xl(i) = 1.0/(nptsl-1) * (i-1)
        Tl(i) = u(neqng + npdel*(i-1) + 1)
        call getlprop (Tl(i), pressure, c(i), rho(i), lam(i),
     1                 dummy, dummy, dummy, dummy)
      enddo
c get first point diffusion term
      R(1) = lam(2)*(Tl(2)-Tl(1))/(xl(2)*rp-xl(1)*rp)*4.0*3.14159*(rp*xl(2))**2
c get intermediate diffusion terms
      do i = 2, nptsl-1
        R(i) = ( 0.5*(lam(i+1)*xl(i+1)**corder+lam(i)*xl(i)**corder)
     1                 * (Tl(i+1)-Tl(i))/(xl(i+1)-xl(i))
     2         - 0.5*(lam(i)*xl(i)**corder+lam(i-1)*xl(i-1)**corder)
     3                 * (Tl(i)-Tl(i-1))/(xl(i)-xl(i-1)) )
     4       / ((xl(i+1) - xl(i-1)) / 2.0)
      enddo
c get last point diffusion terms
      R(nptsl) = ( (lam(nptsl)*xl(nptsl)**corder)
     1                 * Tr_1*rp
     2         - 0.5*(lam(nptsl-1)*xl(nptsl-1)**corder
     2     + lam(nptsl)*xl(nptsl)**corder)
     3                 * (Tl(nptsl)-Tl(nptsl-1))/(xl(nptsl)-xl(nptsl-1)) )
     4       / ((xl(nptsl) - xl(nptsl-1)) / 2.0)
c determine order and LEFT boundary condition
c droplet: slope = 0
c wall: fixed temperature
      if (corder .eq. 2) then
        Tldot(1) = 1.0/(rho(2)*c(2)*4.0/3.0*3.14159
     1                  * (xl(2)*rp)**3) * R(1)
      else
        Tldot(1) = 0.0d0
      endif
c calculate time rate of change of intermediate points
      do i = 2, nptsl
        Tldot(i) = 1.0/(rho(i)*c(i)*xl(i)**corder*rp**2) * R(i)
      enddo
c calcualte time rate of change of interface point
      Tldot(nptsl) = dTdt_1

      return
      end
c end of subroutine liqdef() ***********************************************


c ***************************************************************************
      subroutine pdedef    (y, ts, rxi, tempi, wtmi, rhoi, rp, mp, nspecies,
     1                      m, r, q, p)
```

```
c ************************************************************************
c *                              PDEDEF                                  *
c *     This subroutine defines the coefficients present in the gas phase *
c *  conservation equations.  Given inputs of the conditions at a specific *
c *  point, the routine returns the coefficients for all the species at that *
c *  point (see 'func' for definition of coefficients in the respective   *
c *  conservation equations).                                            *
c *                                                          03/08/1999  *
c ************************************************************************
c
c  Input variables:
c
c  y[nsp+1]   = array of species mass fractions at current location
c  ts         = current time (s)
c  rxi        = untransformed coordinate location (gridpoint i) (m)
c  tempi      = gas temperature at gridpoint i (K)
c  wtmi             = gas mean molar weight at gridpoint i (kg/Kmole)
c  rhoi             = gas density at gridpoint i (kg/m^3)
c  rp            = current droplet radius (m)
c  mp         = current mass flow due to evaporation (kg/s)
c  nspecies   = number of species in solution array
c
c  Output variables:
c
c  m[nsp+1]   = convection term coefficient at grodpoint i
c  r[nsp+1]   = diffusion coefficents (mass) at gridpoint i
c  q[nsp+1]   = chemical production rate coefficients at gridpoint i
c  p[nsp+1]   = unsteady term coefficients at gridpoint i
c                  m, r, p, q returned for all species and temperature
c
c  Notes:  o Sandia CHEMKIN and TRANSPORT routines are called to provide
c            thermodynamic and transport data for species.
c          o The diffusion coefficient does not include the derivative term;
c            in order to calculate this term (usually performed in 'func'),
c            the following operation is necessary:
c            R = d/dx (r dy/dx)
c          o The subroutine includes a simple k-e model for turbulence which
c            can be turned on or off.  At the present time this is done
c            by setting the value of global variable 'turb' to 0 for 'off' and
c            to 1 for 'on'.
c
c                                                              03/08/99
      implicit none
c argument variables:
      double precision      y(*), ts, rxi, tempi, wtmi, rhoi, rp, mp
      integer               nspecies
      double precision      m(*), p(*), q(*), r(*)
c common variables:
      double precision      pressure, dpdt, length, avespeed, tconst
      double precision      gasconst, minrp
      integer               turb, corder, chem, fuel
      common /pressdat/     pressure, dpdt, length, avespeed, tconst
      common /misc/         gasconst, minrp, turb, corder, chem, fuel
c CHEMKIN variables:
      integer           npdemx, leniwk, lenrwk, lencwk
      parameter              (npdemx = 80, leniwk = 7500, lenrwk = 90000,
     1                    lencwk = 500)
      double precision       diffmc(npdemx), hml(npdemx), wi(npdemx),
     1                    molarf(npdemx)
      integer               ickwrk(leniwk), imcwrk(leniwk)
      double precision      rckwrk(lenrwk), rmcwrk(lenrwk)
      character             cckwrk(lencwk)*(16)
      common /ckspi/        ickwrk, imcwrk
```

```
      common /ckspr/        rckwrk, rmcwrk
      common /ckspc/        cckwrk
c local variables:
      integer           k
      double precision        sumwh, conmix, init_kin, init_tau, init_l,
     1                  kin, turb_diff, a_total, cpbar
c temporary variables:
      parameter (init_l = 3.0d-3)
c end of variable declarations

c ---------------------------------------------------------------------------
c part i.
c Modified CHEMKIN and TRANSPORT subroutines called to return necessary
c thermodynamic and transport information.  Some units given before calls.
c ---------------------------------------------------------------------------
c units of wi (production rate of species): kg/(m^3*sec)
      call ck_species_source(pressure, tempi, y, ickwrk, rckwrk, wi)
      call ck_y2x(y, ickwrk, rckwrk, molarf)
      call ck_cpbl(tempi, molarf, ickwrk, rckwrk, cpbar)
c convert from unit J/(Kmole*K) to J/(kg*K)
      cpbar = cpbar/wtmi
c units of enthalpy: J/Kmole
      call ck_hml(tempi, ickwrk, rckwrk, hml)
c units of heat of reaction: J/m^3s
      call ck_react_heat(wi, hml, rckwrk, sumwh)
c temperature and species diffusion terms
      call mc_acon(tempi, molarf, rmcwrk, conmix)
      call mc_adif(pressure, tempi, molarf, rmcwrk, diffmc)

c ---------------------------------------------------------------------------
c part ii.
c Calculating turbulent transport coefficients with a simplifies k-e model
c (if turb = 0, laminar diffusion calculated).
c ---------------------------------------------------------------------------
      if (turb .eq. 0) then
        turb_diff = 0.0d0
      else
        init_kin = 1.5*(0.75*avespeed)*(0.75*avespeed)
        init_tau = init_l/sqrt(init_kin)
        kin = init_kin*(1+0.9*(tconst+ts)/init_tau)**(-1.11)
        turb_diff = 0.09*(rxi-rp)/(1+(rxi-rp)/init_l)*sqrt(kin)
      endif
      a_total = (turb_diff + conmix/(rhoi*cpbar)) * rhoi*cpbar

c ---------------------------------------------------------------------------
c part iii.
c Calculating PDE coefficients from obtained thermodynamic and transport
c information.
c ---------------------------------------------------------------------------

c OPTIONAL simplified case ********************************************
**************************************************************************
c        cpbar = 1744.0
c        conmix = 0.10335
c        do k = 1, nspecies
c          diffmc(k) = conmix/(rhoi*cpbar)
c          hml(k) = 0.0d0
c        enddo
c        hml(fuel) = 136.85e6
c        call ck_react_heat(wi, hml, rckwrk, sumwh)
c        a_total = conmix
**************************************************************************
```

```
c convective coefficient
      do k = 1, nspecies
        m(k) = -mp
      enddo
      m(nspecies+1) = -mp*cpbar

c diffusion coefficients
      do k = 1, nspecies
        r(k) = rxi**(2*corder)*rhoi**2*(diffmc(k)+turb_diff)
      enddo
      r(nspecies+1) = rxi**(2*corder)*rhoi*a_total

c chemical production rate coefficients
      do k = 1, nspecies
        q(k) = wi(k)*real(chem)/rhoi
      enddo
      q(nspecies+1) = (dpdt-sumwh*real(chem))/rhoi

c unsteady term coefficients
      do k = 1, nspecies
        p(k) = 1.0
      enddo
      p(nspecies+1) = cpbar

      return
      end
c end of subroutine pdedef() ***************************************************


c ***************************************************************************
      subroutine fofv( vrp, rx, x, qdot, dTdt_l, a_total, Temp,
     1                 rho, cp_l, Al, Al2, dVl, rholiquid, cliquid, lamliquid,
     2                 Yx_l, Yx_r, u, ux, e, rp, diffmc_l,
     3                 mp, wtk, R, Q, pressure, dpdt, Lt,
     4                 A1, B1, C1, dTdr_l, Tr_l, dTldr_o, nptsl,
     5                 npde, fuel, nspecies, corder, fv )
c ***************************************************************************
c *     This subroutine returs the value of the function which defines the    *
c *  solution of the eigenvalue problem.                                      *
c *                                                              03/08/1999   *
c ***************************************************************************
c
c  Output variables:
c
c  dTdr_l     = derivative of gas-phase temperature at interface
c  qdot             = total heat transfer per area at interface
c  Tr_l             = derivative of liquid-phase temperature at interface
c  dTdt_l     = time rate of change of temperature at interface
c  Yx_l[npde] = species derivatives at right gas-phase interface
c  Tx_r[npde] = species derivatives at left gas-phase interface
c  mp         = problem eigenvalue: mass flux at interface
c  fv         = value of problem function (zero when solution is found)
c
c
c                                                                 03/08/99
      implicit none
      integer           k
c required data for solution
      double precision  vrp, rx(*), x(*), qdot, dTdt_l, a_total(*), Temp(*),
     1                  rho(*), cp_l, Al, Al2, dVl, rholiquid, cliquid,
     1                  lamliquid,
     2                  Yx_l(*), Yx_r(*), u(*), ux(*), e(*), rp, diffmc_l(*),
     3                  mp, wtk(*), R(6,*), Q(6,*), pressure, dpdt, Lt,
```

```
     4                  A1, B1, C1, fv, dTdr_l, Tr_l, dTldr_o
      integer           npde, fuel, nspecies, corder, nptsl
c variables
      double precision  sum1, sum2, pp, dppdt, dTemp, dXfdT,
     1                  dXfdp, dTdr_l_l, dTdt_l2
      parameter        (dTemp = 1.0e-8)
c end of variable declaration

      dTdr_l = ( 0.5*(rx(1)**corder*a_total(1)+rx(2)**corder*a_total(2))
     1                      *(Temp(2)-Temp(1))/((rx(2)-rx(1)))
     2             / (rho(1)*cp_l*rp**corder*(rx(2)-rx(1))/2.0)
     3             + (Al*rho(1)*vrp*Lt + Al2*lamliquid*dTldr_o)
     4                /(dVl*rholiquid*cliquid) )
     5         / (vrp + a_total(1)/(rho(1)*cp_l*(rx(2)-rx(1))/2.0)
     6             + (Al*a_total(1)/(dVl*rholiquid*cliquid)))

      qdot = -(a_total(1)*dTdr_l - rho(1)*vrp*Lt)
      Tr_l = -qdot/lamliquid
      dTdt_l = -(qdot*Al + lamliquid*dTldr_o*Al2)
     1                          / (dVl * rholiquid * cliquid)

      dTdt_l2 = 1/(rp**corder*rho(1)*cp_l)*(0.5*(rx(2)**corder*a_total(2)
     1          + rx(1)**corder*a_total(1))
     1         *(Temp(2)-Temp(1))/(rx(2)-rx(1)) - rp**corder*a_total(1)
     2         *dTdr_l)/((rx(2)-rx(1))/2.0) - vrp*dTdr_l

c      write (*,*) dTdt_l, dTdt_l2

c test
c      dTdt_l = 0.0d0

c impose boundary conditions as functions of eigenvalue
      do k = 1, nspecies
        Yx_l(k) = vrp*(u(k)-e(k))/(rho(1)*rp**corder*diffmc_l(k))
        Yx_r(k) = 0.0d0
      enddo
c mass flow term for transformed equations
      mp = rp**corder * rho(1) * vrp * (1.0 + rho(1)/rholiquid)
      sum1 = 0.0d0
      sum2 = 0.0d0
      do k = 1, nspecies
        sum1 = sum1 + u(k)/wtk(k)
        sum2 = sum2 + 1.0/wtk(k)*
     1        (-mp*Yx_l(k)+(0.5*(R(1,k)+R(2,k))
     1                  *(u(npde+k)-u(k))/(x(2)-x(1))
     2                    - R(1,k)*Yx_l(k)) / ((x(2)-x(1))/2.0) + Q(1,k))
      enddo
c determine derivatives with respect to pressure and temperature
      pp = pressure * 7.4948 / 1000.0
      dppdt = dpdt  * 7.4948 / 1000.0
      dXfdT = (10.0**(A1-B1/(Temp(1)+dTemp-273.0+C1))/pp
     1          - 10.0**(A1-B1/(Temp(1)-dTemp-273.0+C1))/pp) / (2.0*dTemp)
      dXfdp = -10.0**(A1-B1/(Temp(1)-273.0+C1)) / pp**2
c calculate value of function
      k = fuel
      fv = 1.0/wtk(k)*(-mp*Yx_l(k) + (0.5*(R(1,k)+R(2,k))
     1                    *(u(npde+k)-u(k))/(x(2)-x(1))
     1                   - R(1,k)*Yx_l(k)) / ((x(2)-x(1))/2.0) + Q(1,k))
     2     -(dXfdp*dppdt + dXfdT*dTdt_l)*sum1
     3     -ux(k)*sum2

      return
      end
```

```
c end of subroutine fofv() ************************************************


c ***************************************************************************
      subroutine getzero ( vrp, vrpold, rx, x, qdot, dTdt_l, a_total, Temp,
     1                     rho, cp_l, Al, Al2, dVl, rholiquid, cliquid,
     1                     lamliquid,
     2                     Yx_l, Yx_r, u, ux, e, rp, diffmc_l,
     3                     mp, wtk, R, Q, pressure, dpdt, Lt,
     4                     A1, B1, C1, dTdr_l, Tr_l, dTldr_o, nptsl,
     5                     npde, fuel, nspecies, corder, fv )
c ***************************************************************************
c *     This subroutine executes the Newton interation to find the correct    *
c *  solution of the eigenvalue problem.                                      *
c *                                                              03/08/1999   *
c ***************************************************************************
c
c  Output variables:
c
c  dTdr_l       = derivative of gas-phase temperature at interface
c  qdot         = total heat transfer per area at interface
c  Tr_l         = derivative of liquid-phase temperature at interface
c  dTdt_l       = time rate of change of temperature at interface
c  Yx_l[npde]   = species derivatives at right gas-phase interface
c  Tx_r[npde]   = species derivatives at left gas-phase interface
c  mp           = problem eigenvalue: mass flux at interface
c  fv           = value of problem function (zero when solution is found)
c
c
c                                                              03/08/99
      implicit none
      integer           k, iteration, iterationlimit
      double precision  dfdv, dvrp, fv1, fv2, vrp1, vrp2, error
      parameter        (dvrp = 1.0e-10, error = 1.0e-10, iterationlimit = 40)
c data for slution
      double precision  vrp, rx(*), x(*), qdot, dTdt_l, a_total(*), Temp(*),
     1                  rho(*), cp_l, Al, Al2, dVl, rholiquid, cliquid,
     1                  lamliquid,
     2                  Yx_l(*), Yx_r(*), u(*), ux(*), e(*), rp, diffmc_l(*),
     3                  mp, wtk(*), R(6,*), Q(6,*), pressure, dpdt, Lt,
     4                  A1, B1, C1, fv, vrpold, dTdr_l, Tr_l, dTldr_o
      integer           npde, fuel, nspecies, corder, nptsl
c end of variable declaration

c use the following block to output the function to be solved
c      do vrp = -10, 10, .1
c        call fofv (vrp, rx, x, qdot, dTdt_l, a_total, Temp,
c     1                    rho, cp_l, Al, Al2, dVl, rholiquid, cliquid,
c     1                    lamliquid,
c     2                    Yx_l, Yx_r, u, ux, e, rp, diffmc_l,
c     3                    mp, wtk, R, Q, pressure, dpdt, Lt,
c     4                    A1, B1, C1, dTdr_l, dTldr_o, nptsl,
c     5                    npde, fuel, nspecies, corder, fv1)
c        write (*,*) vrp, fv1
c      enddo
      vrp = vrpold
c get derivative at vrp
 10   vrp1 = vrp
      call fofv (vrp1, rx, x, qdot, dTdt_l, a_total, Temp,
     1                    rho, cp_l, Al, Al2, dVl, rholiquid, cliquid,
     1                    lamliquid,
     2                    Yx_l, Yx_r, u, ux, e, rp, diffmc_l,
     3                    mp, wtk, R, Q, pressure, dpdt, Lt,
```

```
     4                    A1, B1, C1, dTdr_l, Tr_l, dTldr_o, nptsl,
     5                    npde, fuel, nspecies, corder, fv1)
      vrp2 = vrp1 + dvrp
      call fofv (vrp2, rx, x, qdot, dTdt_l, a_total, Temp,
     1                    rho, cp_l, Al, Al2, dVl, rholiquid, cliquid,
     1                    lamliquid,
     2                    Yx_l, Yx_r, u, ux, e, rp, diffmc_l,
     3                    mp, wtk, R, Q, pressure, dpdt, Lt,
     4                    A1, B1, C1, dTdr_l, Tr_l, dTldr_o, nptsl,
     5                    npde, fuel, nspecies, corder, fv2)
      dfdv = (fv2 - fv1) / dvrp
      vrp = vrp - fv1/dfdv
c quit if minimim error reached
      if (abs(vrp - vrp1) .lt. error) then
        return
      endif
c quit if interating too many times, otherwise repeat loop
      if (iteration .lt. iterationlimit) goto 10
c quit if absolute error too small
      if (abs(fv) .le. error) return
c failure due to too many iterations
      write (*,*) 'Failed to find root!'
      stop

      end
c end of subroutine getzero() ********************************************
```

PROP.F

```
c ***************************************************************************
      subroutine getlprop (Temp, press, cliquid, rholiquid, lamliquid, Lht,
     1                     A1, B1, C1)
c ***************************************************************************
c *     This subroutine must be provided by the user and give the properties  *
c *  of the liquid that must be used.  See below for the description of the   *
c *  form of required data.  All are possible functions of temperature.       *
c *                                                                 03/08/1999    *
c ***************************************************************************
c
c  Input variables:
c
c  Temp             = temperature of liquid (K)
c
c  Output variables:
c
c  cliquid    = liquid specific heat (J/kg-K)
c  rholiquid = liquid density (kg/m^3)c
c  lamliquid = liquid heat conductivity (W/m-K)
c  Lht        = liquid latent heat (J/kg)
c  A1, B1, C1= Antoine equation parameters (for fuel), must conform to:
c              Pf = 10^(A1 - B1/(Tl + C))
c              where Pf is the partial pressure of fuel in the gas phase
c
c                                                                  03/08/99
      implicit none
      double precision       Temp, cliquid, rholiquid, lamliquid,
     1                       Lht, A1, B1, C1, press
      double precision       th
c end of variable declaration

c ************************* DEFAULT: METHANOL ********************************
c Methanol liquid properties:
c density:
      th = 1.0 - Temp/512.64
      rholiquid = 1.0e3*32.042*(0.0224941*th**(1.0/3.0)
     1                 - 0.0281691*th**(2.0/3.0)
     2                     + 0.107894*th
     3                         - 0.156349*th**(4.0/3.0)
     4                            + 0.0803262*th**(5.0/3.0) + 1.0/118.0)
c specific heat:
      cliquid = 8.314*(9.83728
     1               - 4.87999 * (Temp/512.64)
     2                   - 5.08323 * (Temp/512.64)**2
     3                      + 22.7383 * (Temp/512.64)**3) * 1000/32.024
c heat conductivity:
      lamliquid = 0.2
c      lamliquid = 86.0060 + 1.13222*(Temp/512.64) - 0.73678*(Temp/512.64)**
c Methanol phase equilibrium constants:
      if (press .lt. 337250.0) then
        A1 = 7.87863
        B1 = 1473.11
        C1 = 230.0
      else
        A1 = 8.07246
        B1 = 1574.99
        C1 = 238.86
      endif

c latent heat:
c      Lht = 35.354*((513.0 - Temp)/(513.0 - 337.8))**0.38
c      Lht = Lht * 1000.0 * 1000.0 / 32.04
```

```
      th = (512.64-Temp)/(512.64-175.59)
      Lht = (41846.0*th + (th**0.700886-th)*
     1       (147232.0-75568.4*th+44997.3*th**2.0))
     2     / 32.042 * 1000.0

      return
      end
c end of getlprop () ********************************************************


c ****************************************************************************
      subroutine equil (u, nspecies, npde, A1, B1, C1)
c ****************************************************************************
c *     This subroutine determines the equilibrium concentration given A1,    *
c *  B1, and C1.  It is called at the beginning of the program run to          *
c *  insure that the equilibrium concentration exists at the liquid/gas        *
c *  interface before beginning calculations that only depend on               *
c *  time integration.  It may be modified if necessary (if other than         *
c *  Antoine equation is to be used).                                          *
c *  Note: the correct gas-phase equilibrium is determined by the temperature *
c *  at the interface.                                                         *
c *                                                          03/08/1999    *
c ****************************************************************************
c
c  Input variables:
c
c  u         = for temperature at interface
c  A1, B1, C1= Antoine constants
c  npde             = number of total pdes
c  nspecies  = number of species
c
c  Output variables:
c
c  u            = solution vector with correct initial conditions at interface
c
c                                                                 03/08/99
      implicit none
c input variables/parameters
      double precision      u(*)
      integer               nspecies, npde, npdemx, nptsmx, ref, k
      parameter            (npdemx = 80, nptsmx = 205)
      double precision      ux(npdemx), sum, wtk(npdemx), Eq(npdemx),
     1                      uxnew(npdemx), Temp(nptsmx), y(npdemx), wtm,
     2                      A1, B1, C1
c CHEMKIN variables
      integer               leniwk, lenrwk, lencwk
      parameter            (leniwk = 7500, lenrwk = 90000, lencwk = 500)
      integer               ickwrk(leniwk), imcwrk(leniwk)
      double precision      rckwrk(lenrwk), rmcwrk(lenrwk)
      character             cckwrk(lencwk)*(16)
      common /ckspi/        ickwrk, imcwrk
      common /ckspr/        rckwrk, rmcwrk
      common /ckspc/        cckwrk
c local variables
      double precision      pressure, dpdt, length, avespeed, tconst, pp
      double precision      gasconst, timeold, timesucc, minrp
      integer               turb, corder, chem, fuel
      common /pressdat/     pressure, dpdt, length, avespeed, tconst
      common /misc/         gasconst, minrp, turb, corder, chem, fuel
c end of variable declaration

c get necessary data: mole fractions, temperature, and pressrue at interface
      do k = 1, nspecies
```

```
          y(k) = u(k)
        enddo
        call ck_mmwy(y, ickwrk, rckwrk, wtm)
        Temp(1) = u(nspecies+1)
        Temp(2) = u(npde + nspecies+1)
        pp = pressure * 7.4948 / 1000
c gas-liquid phase equilibrium mole fraction of fuel at interface
        uxnew(fuel) = 10**(A1-B1/(Temp(1)-273.0+C1))/pp
c convert to others to molar fractions
        do k = 1, nspecies
          ux(k) = u(k)
c determine reference species that can be used to preserve proportional
c distribution of remaining species
          if ((u(k) .gt. 1.0e-3) .and. (k .ne. fuel)) ref = k
        enddo
        call ck_y2x (ux(1), ickwrk, rckwrk, ux(1))
        call ck_mk  (ickwrk, rckwrk, wtk)
        sum = 0.0
        do k = 1, nspecies
          if ((k .ne. ref) .and. (k .ne. fuel)) sum = sum + ux(k)/ux(ref)
        enddo
        do k = 1, nspecies
          if (k .ne. fuel) uxnew(k) = ux(k)/ux(ref)*(1-uxnew(fuel))/(1+sum)
        enddo
        sum = 0.0d0
        do k = 1, nspecies
          sum = sum + uxnew(k)*wtk(k)
        enddo
        do k = 1, nspecies
          Eq(k) = uxnew(k)*wtk(k)/sum
        enddo
c assign values to solution vector
        do k = 1, nspecies
          u(k) = Eq(k)
        enddo

        return
        end
c end of equil () ************************************************************


c ****************************************************************************
      subroutine liquidpr (u, rp, Tl, neqng, npdel, nptsl, corder, Tw)
c ****************************************************************************
c *     This subroutine determines the initial temperature profiles in the    *
c *  liquid phase.  Default are linear for wall condition (with wall temp     *
c *  at 361K) and zero slope for droplet, with the initial, gas-specified     *
c *  temperature at the interface for both cases.  This subroutine may also   *
c *  be modified to satisfy other conditions.                                 *
c *                                                          03/08/1999    *
c ****************************************************************************
c
c  Input variables:
c
c  rp         = radius of layer/liquid
c  Tl         = temperature at interface
c  neqng      = number of equations in the gas phase
c  npdel      = number of equations in the liquid phase
c  nptsl      = number of points in the liquid phase
c  coder      = geometric factor
c
c  Output variables:
c
```

```
c  u            = solution vector with correct initial conditions for liquid
c
c                                                                    03/08/99
      implicit none
c local variables
      double precision     u(*), T(200), Tw, xl, rp, r, Tl
      integer              neqng, npdel, nptsl, corder, i
c endo of variable declarations

c initial liquid profiles:
c DEFAULT:
c corder = 0: Twall = 360 (default), linear profile to Tinterface
c corder = 2: Teverywhere = Tinterface
c equispaced in (r/rp)
      do i = 1, nptsl
        r = rp/(nptsl-1) * (i-1)
        if (corder .eq. 2) then
          T(i) = Tl
        else
          T(i) = (Tl - Tw) / rp * r + Tw
        endif
      enddo
c assign to solution vector
      do i = 1, nptsl
        u(neqng + npdel*(i-1) + 1) = T(i)
        u(neqng + npdel*(i-1) + 2) = rp
      enddo

      return
      end
c end of liquidpr () ************************************************************
```

SUBS.F

```
c=======================================================================
      subroutine remesh ( npts, npde, xj, y, xjnew, nptsnew,
     +                    a, b, mindis, maxdiff, different )
c=======================================================================
c
c       remesh() sub-equidistributes a function weighted by
c     differences in the components and gradients of components
c     at adjcent mesh points.
c
c     Inputs:  npts, xj, y, nptsnew, a, b, mindis, maxdiff
c     Outputs: xjnew, nptsnew, different
c
c     The desired result is to have a function where the constant Kc is
c     such that 1.5 int w1 = int wo.  w1 = Mesh function with Kc <> 0, wo =
c     Mesh function with Kc = 0.  This supposedly gives very good results:
c     Larrouturou, Bernard, "Numerical Modeling in Combustion", Edited by
c     T. J. Chung, Taylor & Francis, 1993, p. 183.  Our gradients here seem
c     larger, however, and a factor of 1.1 works better.
c=======================================================================

      implicit            none
      integer             nptsmx
      parameter           (nptsmx = 205)
      integer             i, j, k, npts, nptsnew, npde, nptsold
      integer             curPoints, outPoints
      double precision    slope, target, mindisTol, factor
      double precision    xj(*), y(*), Fmesh(nptsmx), Fmesh_o(nptsmx),
     +                    xjnew(*), a(*), b(*), h, Kc,
     +                    into, int1, one, min, max,
     +                    mindis, maxdiff,
     +                    xmeshold(nptsmx), Fmeshold(nptsmx)
      logical             different
      common /Fmeshdata/  Fmeshold, xmeshold, nptsold
      parameter           (mindisTol = 1.0E-09)
      parameter           (factor = 1.05)

      write (*,*) 'Entering remesh with target: ', nptsnew, ' points.'

      one = 1.0

c---- Calculate the mesh function ---------------------------------------
c----  (larger values of factor result in more even point distributions)

      Kc = 0.0
      call getFmesh(npts, npde, xj, y, Fmesh_o, a, b, Kc)

      into = 0.0
      do i = 2, npts
        into = into + 0.5*abs(Fmesh_o(i)+Fmesh_o(i-1))*(xj(i)-xj(i-1))
      enddo

      Kc = 1.0e-30
      int1 = into
      do while ( (factor*int1 .gt. into) .and. (Kc .lt. 1.0e11) )
         call getFmesh(npts, npde, xj, y, Fmesh, a, b, Kc)
         int1 = 0.0
         do i = 2, npts
            int1 = int1 + 0.5*abs(Fmesh(i)+Fmesh(i-1))*(xj(i)-xj(i-1))
         enddo
         Kc = Kc * 1.5
      enddo
```

```
c       Kc = 1.0e4
c       call getFmesh(npts, npde, xj, y, Fmesh, a, b, Kc)

c---- mesh setup and minimum checking algorithm ---------------------------
       curPoints = nptsnew
       outPoints = 0

       do while ( (outPoints .lt. nptsnew) .and. (curPoints .lt. 200 ) )
          xjnew(1) = xj(1)

          i = 1
          do j = 1, curPoints-2
             target = (one*j)/(one*(curPoints-1))

             do while ( (Fmesh(i+1) .lt. target) .and. (i .lt. npts) )
                i = i+1
             enddo

             slope = (Fmesh(i+1) - Fmesh(i)) / (xj(i+1) - xj(i))

             if ( slope .ne. 0.0 ) then
                xjnew(j+1) = xj(i) + (target - Fmesh(i))/slope
             else
                xjnew(j+1) = xj(i+1)
             endif

c---- eliminate mesh spacings below mindis
             if ( (xjnew(j+1) - xjnew(j)) .lt. mindis ) then
                xjnew(j+1) = xjnew(j) + mindis
             endif
          enddo

          xjnew(curPoints) = xj(npts)

          outPoints = curPoints
          do i = 1, curPoints-1
             j = i+1
             k = i
             do while ( xjnew(j) .lt. (xjnew(k) + mindis - mindisTol) )
                k = k-1
                outPoints = outPoints-1
             enddo
             xjnew(k+1) = xjnew(j)
          enddo

          curPoints = curPoints + (nptsnew - outPoints)

       enddo

       nptsnew = outPoints

c---- determine minimum and maximum spacings in output mesh ---------------
       min = xjnew(2) - xjnew(1)
       max = min
       do i = 2, nptsnew
          h = xjnew(i) - xjnew(i-1)
          if ( h .lt. min ) min = h
          if ( h .gt. max ) max = h
       enddo
       write (6,*) 'Minimum spacing: ', min
       write (6,*) 'Maximum spacing: ', max

c---- check to see if mesh has changed significantly from last time -------
```

```
      call compare(maxdiff, Fmesh, xj, npts,
     +             Fmeshold, xmeshold, nptsold, different)

      if (different) then
         do i = 1, npts
            Fmeshold(i) = Fmesh(i)
            xmeshold(i) = xj(i)
         enddo
      endif
c-----------------------------------------------------------------------

      write (*,*) 'Exit remesh - new points: ', nptsnew

      return
      end
c=======================================================================

c=======================================================================
      subroutine getFmesh(npts, npde, xj, u, Fmesh, a, b, Kc)
c=======================================================================
c       getFmesh produces a mesh function Fmesh, given the grid point
c     locations xj(npts), the species profiles u(npts, npde), weight
c     arrays a(npde) and b(npde) for the first and second derivatives
c     (respectively) of each species, and the "flatness factor" Kc.
c     The formula for Fmesh(i), where 1 <= i <= npts, is as follows:
c
c      Fmesh(i) = (1/Fmesh(npde)) * SUM[k = 1..npde]{
c                  a[k]*Integral[x = 0..xj(i)](dy(k)/dx + Kc)dx /
c                       Integral[x = 0..xj(npts)](dy(k)/dx + Kc)dx +
c                  b[k]*Integral[x = 0..xj(i)](d^2y(k)/dx^2 + Kc)dx
c                       Integral[x = 0..xj(npts)](d^2y(k)/dx^2 + Kc)dx }
c
c       As can be seen from inspection of the above formula, Kc controls
c     the smoothness of the above function.  If Kc is very large, Fmesh
c     will be a straight line from 0 to 1, which will produce a mesh with
c     equally spaced gridpoints.  If Kc is zero, the meshpoints will be
c     placed _only_ in areas where the sum of the first and second species
c     derivatives is nonzero.
c=======================================================================
      implicit         none
      integer          i, k, npts, npde
      integer             nptsmx
      parameter          (nptsmx = 205)
      double precision xj(*), y(nptsmx), yx(nptsmx), yxx(nptsmx), yxint, yxxint,
     +                 dx, Fmesh(*), a(*), b(*), yxintx, yxxintx, Kc,
     +                 u(*)

      do i = 1, npts
        Fmesh(i) = 0.0
      enddo

      do k = 1, npde

c---- put current species profile into y(npts)
         do i = 1, npts
            y(i) = u(npde*(i-1) + k)
         enddo

c---- get first and second derivatives
         call deriv ( xj, y, yx, npts, 1, 3 )
         call deriv ( xj, y, yxx, npts, 2, 3 )

c---- integrate first and second derivatives over entire domain
```

```
          yxint = 0.0
          yxxint = 0.0

          do i = 2, npts
            dx = xj(i) - xj(i-1)
            yxint = yxint + 0.5*(abs(yx(i)+yx(i-1)) + Kc)*dx
            yxxint = yxxint + 0.5*(abs(yxx(i)+yxx(i-1)) + Kc)*dx
          enddo

c---- add this species' contribution into Fmesh()

          Fmesh(1) = Fmesh(1) + 0.0

          yxintx = 0.0
          yxxintx = 0.0

          do i = 2, npts
             dx = xj(i) - xj(i-1)
             yxintx = yxintx + 0.5*(abs(yx(i)+yx(i-1)) + Kc)*dx
             yxxintx = yxxintx + 0.5*(abs(yxx(i)+yxx(i-1)) + Kc)*dx

             if ( yxint .ne. 0.0 ) then
                Fmesh(i) = Fmesh(i) + a(k)*((yxintx)/(yxint))
             endif
             if ( yxxint .ne. 0.0 ) then
                Fmesh(i) = Fmesh(i) + b(k)*((yxxintx)/(yxxint))
             endif
          enddo

       enddo

c---- normalize Fmesh()
       do i = 1, npts
         Fmesh(i) = Fmesh(i) / Fmesh(npts)
       enddo

       return
       end
c=======================================================================

c=======================================================================
       subroutine interpall (npts, nptsnew, npde, xj, xjnew, u)
c=======================================================================
c        interpall() returns u interpolated onto xjnew, and
c      changes npts and u to conform to the new grid
c=======================================================================
       implicit none
       integer                 npdemx, nptsmx, neqnmx
       parameter              (npdemx = 80, nptsmx = 205, neqnmx = npdemx*nptsmx)
       integer            i, j, k, npts, npde, nptsnew, nxeqn
       double precision   xj(*), xjnew(*), u(*), unew(neqnmx)


c---- interpolate species data
       do k = 1, npde

          unew(k) = u(k)
          i = 1
          j = 1
 20       continue
           if (xjnew(j) .lt. xj(i)) then
             unew(npde*(j-1) + k) = u(npde*(i-2) + k) +
     +                            ( u(npde*(i-1) + k) -
```

```
     +                              u(npde*(i-2) + k)) *
     +                         (xjnew(j)-xj(i-1)) / (xj(i)-xj(i-1) )
            j = j + 1
          else
            i = i + 1
          endif
        if (j .lt. (nptsnew)) goto 20

        unew(npde*(nptsnew-1) + k) = u(npde*(npts-1) + k)

       enddo



       do i = 1, nptsnew
         xj(i) = xjnew(i)

         do k = 1, npde
           u(npde*(i-1) + k) = unew(npde*(i-1) + k)
         enddo

       enddo

       npts = nptsnew

       return
       end
c=======================================================================

c=======================================================================
       subroutine compare (maxdiff, Fmesh, xj, npts,
      +                    Fmeshold, xmeshold, nptsold, different)
c=======================================================================
c       compare() checks for differences between two mesh functions.
c     If they differ, different is returned as .true.
c=======================================================================
       implicit         none
       integer          npts, nptsold, i
       doubleprecision  maxdiff, Fmesh(*), xj(*), Fmeshold(*),
      +                 xmeshold(*), x, y1, y2, max
       logical          different

       different = .false.

       max = 0.0d0

       do i = 1, npts
         x = xj(i)
         y1 = Fmesh(i)
         call interp1(npts, x, y2, xmeshold, Fmeshold)
         if ( abs(y1-y2) .gt. max ) then
           max = abs(y1-y2)
         endif
       enddo

       write (*,*) 'Maximum mesh function difference: ', max

       if ( max .ge. maxdiff ) different = .true.

       return
       end
c=======================================================================
```

```
c=======================================================================
      subroutine getavg( npts, y, avg )
c=======================================================================
      double precision  y(*), avg, max, i, min
      integer           npts

      max = y(1)
      min = y(1)
      do i = 1, npts
        if (y(i) .gt. max) max = y(i)
        if (y(i) .lt. min) min = y(i)
      enddo
      avg = (max + min) / 2.0

      return
      end
c=======================================================================

c ======================================================================
      subroutine deriv( x, y, Dy, npts, order, method )
c ======================================================================
c     Calculates derivatives of the function y(x), which is specified by
c     the input arrays x(npts) and y(npts).
c     The result is stored in the output array Dy(npts).
c     The input variable order specifies the derivative order (1 or 2)
c     The input variable method specifies the differencing method to use:
c
c         method value             differencing method
c         ------------             -------------------
c              1                   forward difference
c              2                   backward difference
c              3                   centered difference
c
c ======================================================================

      implicit         none
      integer          npts, order, method, i
      double precision x(npts), y(npts), Dy(npts)

c ----- check that arrays are sufficiently large -------------------------
      if ( npts .lt. order+2 ) then
         write (*,*) 'deriv was given arrays that were too short!'
         return
      endif

c ----- calculate the requested derivatives ------------------------------
      if ( order .eq. 1 ) then
         if ( method .eq. 1 ) then

            do i = 1, npts-1
               Dy(i) = (y(i+1) - y(i))/(x(i+1) - x(i))
            enddo

            Dy(npts) = Dy(npts-1)

         elseif ( method .eq. 2 ) then

            do i = 2, npts
               Dy(i) = (y(i) - y(i-1))/(x(i) - x(i-1))
            enddo

            Dy(1) = Dy(2)
```

```
         elseif ( method .eq. 3 ) then

            do i = 2, npts-1
               Dy(i) = (y(i+1) - y(i-1))/(x(i+1) - x(i-1))
            enddo

            Dy(1) = (y(2) - y(1))/(x(2) - x(1))

            Dy(npts) = (y(npts) - y(npts-1))/(x(npts) - x(npts-1))

         endif
      elseif ( order .eq. 2 ) then
         if ( method .eq. 1 ) then

            Dy(1) = ( (y(3)-y(2))/(x(3)-x(2)) - (y(2)-y(1))/(x(2)-x(1)) )
     1              / (x(2) - x(1))

         elseif ( method .eq. 2 ) then

         elseif ( method .eq. 3 ) then

            Dy(1) = ( (y(3)-y(2))/(x(3)-x(2)) - (y(2)-y(1))/(x(2)-x(1)) )
     1              / (x(2) - x(1))

            do i = 2, npts-1
               Dy(i) = 2*( (y(i+1)-y(i))/(x(i+1)-x(i)) -
     1                 (y(i)-y(i-1))/(x(i)-x(i-1)) ) /(x(i+1) - x(i-1))
            enddo

            Dy(npts) = ( (y(npts)-y(npts-1))/(x(npts)-x(npts-1)) -
     1                   (y(npts-1)-y(npts-2))/(x(npts-1)-x(npts-2)) ) /
     2                  (x(npts) - x(npts-1))

         endif
      endif

      return
      end

c ==== end of deriv() ================================================

c=======================================================================
      subroutine interp1 (npts, x, y, xj, yj)
c=======================================================================
      integer           npts, i
      double precision  x, y, xj(*), yj(*)

      i = 1
   10 if (i .eq. npts) goto 20
      i = i + 1
      if (xj(i) .lt. x) goto 10

      y = yj(i-1) + (yj(i) - yj(i-1))
     +              / (xj(i) - xj(i-1))
     +              * (x - xj(i-1))

   20 continue

      return
      end
c=======================================================================
```

INTERP.F

```
      subroutine read_initial( linit, lout, cckwrk, nspecies, npts, x, u,
     1                         rp, corder, Tw, turb, chem, ignite, nptsl, fuel, rtol )
c=======================================================================
c      This subroutine reads in an initial condition file and stores
c      the appropriate data in arrays for the start of integration.
c
c      (describe initial file format)
c
c      written by: Chris O'Brien
c      version: 1.00
c      date: 13 January 1998
c=======================================================================

c==== variable definitions ============================================

c     REMOVE THE IMPLICIT STATEMENT WHEN THE WHOLE THING WORKS?????
      implicit double precision (A-H,O-Z), integer (I-N)


c==== common variables (from CHEMKIN linking file data) ===============

      COMMON /CKSTRT/ NMM , NKK , NII , MXSP, MXTB, MXTP, NCP , NCP1,
     1                NCP2, NCP2T,NPAR, NLAR, NFAR, NLAN, NFAL, NREV,
     2                NTHB, NRLT, NWL,  IcMM, IcKK, IcNC, IcPH, IcCH,
     3                IcNT, IcNU, IcNK, IcNS, IcNR, IcLT, IcRL, IcRV,
     4                IcWL, IcFL, IcFO, IcKF, IcTB, IcKN, IcKT, NcAW,
     5                NcWT, NcTT, NcAA, NcCO, NcRV, NcLT, NcRL, NcFL,
     6                NcKT, NcWL, NcRU, NcRC, NcPA, NcKF, NcKR, NcK1,
     7                NcK2, NcK3, NcK4, NcI1, NcI2, NcI3, NcI4

      COMMON /INITDAT/   tmax, tinc

      integer            linit, lout, nspecies, npts,
     1                   corder, turb, chem, nptsl, fuel
      character          cckwrk(*)*(*)
      character          line*3000, sub(300)*300, upcase*50
      logical            exists
      double precision     x(*), u(NKK+1,*), rp, ignite, Tw, rtol
      double precision   x_cur( 200 ), y_cur( 200 ), dx
      integer            species( 200 ), temp_col, index

      integer            cadInit
      double precision   speed(1),cad(720), pressure(720), length(720)
      common /cadpldat/  speed, cad, pressure, length, cadInit

      parameter            (leniwk = 7500, lenrwk = 90000, lencwk = 500 )
      integer              ickwrk(leniwk), imcwrk(leniwk)
      double precision     rckwrk(lenrwk), rmcwrk(lenrwk)

      common /ckspi/ ickwrk,imcwrk
      common /ckspr/ rckwrk, rmcwrk

c==== common variables for remesh =====================================

      integer              npdemx, nptsmx
      parameter           (npdemx = 80, nptsmx = 205)
      integer              nptsnew, nptstotry, remeshflag, ormesh, nptsold
      logical              different
      double precision     S(nptsmx), xmeshold(nptsmx),
     1                     xjnew(nptsmx), ynew(nptsmx), maxdiff, mindis, avg,
     2                     meshTime, avgarray(npdemx), maxavg, mindistance,
     3                     Fmeshold(nptsmx)
```

```
      common /remesh1/      nptsnew, nptstotry, remeshflag, ormesh, different
      common /remesh3/      S, xjnew, ynew, maxdiff, mindistance, meshTime


c==== variables for boundary condition settings ============================

      integer               bc_species, bc_side, bc_type
      double precision      bc_val
      double precision      bc_flux( 2, 80 )
      integer               which_bc( 2, 80 )

      common /boundary/     bc_flux, which_bc

c==== variables for error-function profile fits ============================

      integer             erfPts
      double precision    erfWidth, erfHeight, erfDx, erfX

c==== initialize variables =================================================

      dx = 0.0
      erfPts = 0
      erfWidth = 0.0
      erfHeight = 0.0
      erfDx = 0.0
      erfX = 0.0
      rp = 0.0
      corder = 0
      ignite = 0.0d0

c---- the elements of which_bc are initialized to -1 to indicate
c----  that no boundary condition has been set.  See set_boundary() in main.f
      do i = 1, NKK+1
         which_bc(1,i) = -1
         which_bc(2,i) = -1
      enddo

c==== list species included in mechanism to output file ====================

      write ( lout, '(A)' ) ' --- Species in chemical mechanism --- '
      do i = 1, NKK
         write ( lout, '(A,I3,A,A)' ) 'Species ',i,': ',cckwrk(IcKK+i-1)
      enddo
      write ( lout, '(A,I4,//)' ) 'total species: ', NKK

c==== parse the first line of the input file ================================
c
c     The first input line should contain the words x, temp, and the names
c      of all species that will be specified on the lines below.  Any species
c      that is not named on this line will be initialized to 0 at all points
c
c============================================================================

10    continue

      read ( linit, '(A)', end = 6000 ) line
      ilen = ipplen( line )
      if ( ilen .eq. 0 ) goto 10
      exists = .true.

      call ckisub( line(:ilen), sub, nsub )

      if ( upcase(sub(1),1) .eq. 'X' ) then
```

```
         continue
      else
         stop 'First column in initial file should be "x"'
      endif

      do i = 2, nsub
         if ( upcase(sub(i),4) .eq. 'TEMP' ) then
            temp_col = i
            goto 20
         endif

         do j = 1, NKK
            lsub = ILASCH(sub(i))-IFIRCH(sub(i))+1
            ifir = IFIRCH(cckwrk(IcKK+j-1))
            ilas = ILASCH(cckwrk(IcKK+j-1))

            if(upcase(sub(i),lsub) .EQ. cckwrk(IcKK+j-1)(ifir:ilas)) then
               species(j) = i
               goto 20
            endif
         enddo

         write(*,*) 'NOTE: unknown species found in initial file:'
         write(*,'(A)') sub(i)

20       continue
      enddo

c==== read the rest of the input file, using first line as template =========

      index = 1
      dx = 0.0

30    continue

      read ( linit, '(A)', end = 6000 ) line
      ilen = ipplen( line )
      if ( ilen .eq. 0 ) goto 30

      call ckisub( line(:ilen), sub, nsub )

c==== first check for keyword lines

c---- Initial profile specification keywords -------------------------------
c---- DX (double x) keyword: constant meshpoint spacing at x until next
c----  x position is given in input file
      if ( upcase(sub(1),2) .eq. 'DX' ) then
         if ( index .eq. 1 ) then
            write(*,*) 'DX statement before first species line is ignored.'
            goto 30
         else
            call ipparr( sub(2), 1, 1, dx, nval, ier, lout )
            if (ier .eq. 2) then
               write(*,'(A,I5)') 'Bad input in initial file, line ', index
               stop
            endif
c            write(*,'(A,1PE12.4)' ) 'dx: ', dx
            goto 30
         endif

c---- ERF (double erfWidth) (int erfPoints) keyword: fit an error function
c----  profile of width erfWidth and number of points erfPoints between the
c----  last specified set of concentrations and the next one.
```

```
      else if ( upcase(sub(1),3) .eq. 'ERF' ) then
         if ( index .eq. 1 ) then
            write(*,*) 'ERF statement before first species line is ignored.'
            goto 30
         else
            call ipparr( sub(2), 1, 1, erfWidth, nval, ier, lout )
            if (ier .eq. 2) then
               write( *,* ) 'Bad input in initial file, line ', index
               write( *,* ) ' ERF command will be ignored. '
               erfPts = 0
            else
               write ( lout, * ) 'error function width: ', erfWidth, ' m'
            endif

            call ippari( sub(3), 0, 1, erfPts , nval, ier, lout )
            if (ier .eq. 2) then
               write( *,* ) 'Bad input in initial file, line ', index
               write( *,* ) ' ERF command will be ignored. '
               erfPts = 0
            else
               write ( lout, * ) 'error function points: ', erfPts
            endif
         endif
         goto 30

c---- Time Stepping Parameter Keywords ----------------------------------------
c---- TMAX: total time for this run
      else if( upcase(sub(1),4) .EQ. 'TMAX' ) then
         call ipparr( sub(2),1,1,tmax,nval,ier,LOUT )
         write( lout, '(A,1PE12.4,A)' ) 'maximum time:  ', tmax, ' seconds.'
         goto 30

c---- TINC: timestep for output
      else if( upcase(sub(1),4) .EQ. 'TINC' ) then
         call IPPARR(SUB(2),1,1,tinc,nval,ier,LOUT)
         write( lout, '(A,1PE12.4,A)' ) 'time interval: ', tinc, ' seconds.'
         goto 30

c---- rp: initial liquid radius (default; rp = 0.0)
      else if( upcase(sub(1),2) .EQ. 'RP' ) then
         call IPPARR(SUB(2),1,1,rp,nval,ier,LOUT)
         write( lout, '(A,1PE12.4,A)' ) 'initial rp:    ', rp, ' m.'
         goto 30

c---- rtol: real tolerace (default; rtol = 1.0e-4)
      else if( upcase(sub(1),4) .EQ. 'RTOL' ) then
         call IPPARR(SUB(2),1,1,rtol,nval,ier,LOUT)
         write( lout, '(A,1PE12.4,A)' ) 'rtol:    ', rtol
         goto 30

c---- nptsl: initial number of points in the liquid phase
c    (default set in main program)
      else if( upcase(sub(1),5) .EQ. 'NPTSL' ) then
         call IPPARI(SUB(2),0,1,nptsl,nval,ier,LOUT)
         write( lout, '(A,I10)' ) 'initial nptsl:          ', nptsl
         goto 30

c---- fuel: initial number of points in the liquid phase
c    (default set in main program)
      else if( upcase(sub(1),4) .EQ. 'FUEL' ) then
         call IPPARI(SUB(2),0,1,fuel,nval,ier,LOUT)
         write( lout, '(A,I10)' ) 'fuel index:             ', fuel
         goto 30
```

```
c---- corder: order of coordinate system:
c                corder = 0 : flat coordinates (flat wall) (default)
c                corder = 1 : cylindrical coordinates
c                corder = 2 : spherical coordinates (droplet)
      else if( upcase(sub(1),6) .EQ. 'CORDER' ) then
         call IPPARI(SUB(2),0,1,corder,nval,ier,LOUT)
         write( lout, '(A,I5)' ) 'coordinate system order: ', corder
         goto 30

c---- Tw (double wall temp): wall temperature for flat case
      else if( upcase(sub(1),2) .EQ. 'TW' ) then
         call IPPARR(SUB(2),1,1,Tw,nval,ier,LOUT)
         write( lout, '(A,F5,A)' ) 'Wall Temperature:             ',
     1         Tw, ' K.'
         goto 30

c---- turb: flag for turbulence:
c                turb = 0 : turbulence off
c                turb = 1 : turbulence on (default)
      else if( upcase(sub(1),4) .EQ. 'TURB' ) then
         call IPPARI(SUB(2),0,1,turb,nval,ier,LOUT)
         write( lout, '(A,I5)' ) 'turbulence flag:         ', turb
         goto 30

c---- chem: flag for chemistry:
c                chem = 0 : chemistry off
c                chem = 1 : chemistry on (default)
      else if( upcase(sub(1),4) .EQ. 'CHEM' ) then
         call IPPARI(SUB(2),0,1,chem,nval,ier,LOUT)
         write( lout, '(A,I5)' ) 'chemistry flag:          ', chem
         goto 30

c---- ignite:   time for ignition (default; ignite = 0.0, no ignition)
      else if( upcase(sub(1),6) .EQ. 'IGNITE' ) then
         call IPPARR(SUB(2),1,1,ignite,nval,ier,LOUT)
         write( lout, '(A,1PE12.4,A)' ) 'ignite at:        ', ignite, ' s.'
         goto 30


c---- Remeshing Algorithm Keywords ----------------------------------------
c---- REMESH ("on" or "off"): turns adaptive mesh on or off for this run
      else if ( upcase(sub(1),6) .EQ. 'REMESH' ) then
         if ( upcase(sub(2),2) .EQ. 'ON' ) then
            remeshflag = 1
            write( lout, '(A)' ) 'Remesh on.'
         else
            remeshflag = 0
            write( lout, '(A)' ) 'Remesh off.'
         endif
         goto 30

c---- MINDIST (double mindistance): specifies mindistance as the smallest
c----  allowed spacing between two meshpoints
      else if( upcase(sub(1),7) .EQ. 'MINDIST' ) then
         call IPPARR(SUB(2),1,1,mindistance,nval,ier,LOUT)
         write( lout, '(A,1PE12.4,A)' ) 'Minimum separation of mesh points: ',
     1         mindistance, ' meters.'
         goto 30

c---- MAXDIFF (double maxdiff): specifies maxdiff as the allowable difference
```

```
c----  between mesh functions - if the difference exceeds maxdiff,
c----  the remeshing algorithm will be called.
      else if( upcase(sub(1),7) .EQ. 'MAXDIFF' ) then
         call IPPARR(SUB(2),1,1,maxdiff,nval,ier,LOUT)
         write( lout, '(A,1PE12.4)' ) 'Maximum difference for remesh: ',
     1         maxdiff
         goto 30

c---- MESHPOINTS (int nptstotry): gives the target number of mesh points.
c----  The number of points produced will be either nptstotry or
c----  (x_maximum/mindist), whichever is smaller.
      else if( upcase(sub(1),10) .EQ. 'MESHPOINTS' ) then
         call IPPARI(SUB(2),0,1,nptstotry,nval,ier,LOUT)
         write( lout, '(A,I5)' ) 'Target number of points for remesh: ',
     1         nptstotry
         goto 30

c---- Initial Crank Angle Degree Keyword ---------------------------------
c---- CAD (int cadInit): cadInit is the crank angle degree in the cad-pl
c----  profile that corresponds to the initial time for this run.
      else if( upcase(sub(1),3) .EQ. 'CAD' ) then
         call IPPARI(SUB(2),0,1,cadInit,nval,ier,LOUT)
         write( lout, '(A,I5)' ) 'Starting crank angle degree: ',
     1         cadInit
         goto 30

c---- Boundary Condition Specification Keywords ---------------------------
c---- BC(char(*)species)("left"/"right")("conc"/"flux"/"evap")(double bcVal):
c----  Sets the boundary condition for a species.  The first argument is
c----  the species name; the second specifies whether the condition is for
c----  the right or left boundary; the third determines whether the
c----  concentration or flux of the species is fixed, or for the "evap" input,
c----  the flux of the species is proportional to heat flux; and the fourth
c----  argument specifies the mole fraction for a concentration b.c.,
c----  the molar flux for a flux b.c., or the constant K in the relationship
c----- (species flux) = K(heat flux) for the evaporation ("evap") b.c.
c----  EXAMPLE:  BC C3H8 left flux 0.10
c----            sets the flux of C3H8 at the left boundary to 0.10 mol/m^2.s

      else if ( upcase( sub(1), 2 ) .eq. 'BC' ) then
         bc_species = -1

         if ( upcase(sub(2),4) .eq. 'TEMP' ) then
            bc_species = NKK + 1
         endif

         do j = 1, NKK
            lsub = ILASCH(sub(2))-IFIRCH(sub(2))+1
            ifir = IFIRCH(cckwrk(IcKK+j-1))
            ilas = ILASCH(cckwrk(IcKK+j-1))

            if(upcase(sub(2),lsub) .EQ. cckwrk(IcKK+j-1)(ifir:ilas)) then
               bc_species = j
            endif
         enddo

         if ( bc_species .gt. -1 ) then
            if ( upcase( sub(3), 4) .eq. 'LEFT' ) then
               bc_side = 1
            else
               bc_side = 2
            endif
```

```
            if ( upcase( sub(4), 4) .eq. 'CONC' ) then
               bc_type = 0
            else if ( upcase( sub(4), 4) .eq. 'FLUX' ) then
               bc_type = 1
            else if ( upcase( sub(4), 4) .eq. 'EVAP' ) then
               bc_type = 2
            endif

            call ipparr ( sub(5), 1, 1, bc_val, nval, ier, lout )

            which_bc( bc_side, bc_species ) = bc_type
            if ( bc_type .eq. 0 ) then
c------------- change this implementation
            else
              bc_flux( bc_side, bc_species ) = bc_val
            endif
         else
            write ( lout, * ) 'Bad species name in BC keyword line - ignored'
            write ( *, * ) 'Bad species name in BC keyword line - ignored'
         endif

         goto 30

c---- END Keyword: ends the initial condition file --------------------------
      else if( upcase(sub(1),3) .eq. 'END') then
          write(lout,'(A,//)') '---------- End of initial file ------------'
          goto 6000
      endif

c==== DEFAULT: Species/Temperature Profile Line ============================
c====  If the line is not a keyword line, read in x, temp, and species =====
c====  All species profile keywords are implemented here as well ===========

      call ipparr( sub(1), 1, 1, x(index), nval, ier, lout )
      if (ier .eq. 2) then
         write(*,'(A,I5)') 'Bad input in initial file, line ', index
         stop
      endif

      if ( dx .gt. 0.0 ) then
c      implementation for "DX" keyword
         do while ( x(index-1)+dx .lt. x(index) )
            x(index+1) = x(index)
            x(index) = x(index-1) + dx
            do i = 1, NKK+1
               u(i, index) = u(i, index-1)
            enddo
            index = index + 1
         enddo
         dx = 0.0
      endif

      do i = 1, NKK
         x_cur(i) = 0.0
         y_cur(i) = 0.0
      enddo

      do i = 2, nsub
         if ( i .eq. temp_col ) then
            call ipparr( sub(i), 1, 1, u(NKK+1,index), nval, ier, lout )
            if (ier .eq. 2) then
               write(*,'(A,I5)') 'Bad input in initial file, line ', index
               stop
```

```
            endif
         endif

         do j = 1, NKK
            if ( species(j) .eq. i ) then
               call ipparr( sub(i), 1, 1, x_cur(j), nval, ier, lout )
               if (ier .eq. 2) then
                  write(*,'(A,I5)') 'Bad input in initial file, line ', index
                  stop
               endif
               goto 40
            endif
         enddo

40       continue
      enddo

      call ck_x2y( x_cur, ickwrk, rckwrk, y_cur )

      do i = 1, NKK
         u(i, index) = y_cur(i)
      enddo

c ---- Insert error-function profile if specified by user
      if ( erfPts .gt. 0 ) then

c        write (*,*) 'Fitting error function'

         erfDx = erfWidth / (1.0 * erfPts)
         erfWidth = erfWidth * 0.4

c        write (*,*) 'dx: ', erfDx

         erfX = x( index-1 )
         x( index + erfPts ) = x( index )
         do i = 1, NKK+1
            u( i, index + erfPts ) = u( i, index )
         enddo

         do i = index, index + erfPts - 1
            x( i ) = x( i-1 ) + erfDx

c           write (*,*) 'Point ', i-index+1, ': ', x(i), 'm'

            do j = 1, NKK+1
               erfHeight = u(j,index+erfPts) - u(j,index-1)

               u(j,i) = u(j,i-1) + erfHeight *
     1           (erfDx/erfWidth/1.772454) *
     2           (exp(-((x(i)-erfX)/erfWidth)*((x(i)-erfX)/erfWidth)) +
     3            exp(-((x(i-1)-erfX)/erfWidth)*((x(i-1)-erfX)/erfWidth) ) )

c              if ( j .eq. NKK+1 ) then
c                 write (*,*) 'T: ', u(j,i)
c              endif
            enddo
         enddo

         index = index + erfPts
         erfPts = 0
      endif
```

```
c=====DIAGNOSTIC OUTPUT========
c      write(*,'(A,I4)') 'step ', index
c      write(*,'(A,1PE12.4)') 'temp ', u(NKK+1,index)
c      do i = 1, NKK
c         write(*,'(1PE12.4,1PE12.4)') x_cur(i),u(i,index)
c      enddo
c=====DIAGNOSTIC OUTPUT========

      index = index+1
      goto 30

6000  if ( .not. exists ) stop 'The initial file is empty!'
      close(linit)

      nspecies = NKK
      npts = index-1

c====DIAGNOSTIC OUTPUT========
c      do i = 1, npts
c         write(*,'(1PE12.4,1PE12.4)') x(i),u(nspecies+1,i)
c      enddo
c====DIAGNOSTIC OUTPUT========

      return
      end

c==== End of subroutine read_initial ========================================

      subroutine write_species( ldata, cckwrk )
c=============================================================================
c      This subroutine writes the species names to a binary data file,
c      each preceeded by its length
c=============================================================================

c==== variable definitions ==================================================

      implicit double precision (A-H,O-Z), integer (I-N)

c==== common variables (from CHEMKIN linking file data) =====================

      COMMON /CKSTRT/ NMM , NKK , NII , MXSP, MXTB, MXTP, NCP , NCP1,
     1                NCP2, NCP2T,NPAR, NLAR, NFAR, NLAN, NFAL, NREV,
     2                NTHB, NRLT, NWL,  IcMM, IcKK, IcNC, IcPH, IcCH,
     3                IcNT, IcNU, IcNK, IcNS, IcNR, IcLT, IcRL, IcRV,
     4                IcWL, IcFL, IcFO, IcKF, IcTB, IcKN, IcKT, NcAW,
     5                NcWT, NcTT, NcAA, NcCO, NcRV, NcLT, NcRL, NcFL,
     6                NcKT, NcWL, NcRU, NcRC, NcPA, NcKF, NcKR, NcK1,
     7                NcK2, NcK3, NcK4, NcI1, NcI2, NcI3, NcI4

      integer            ldata, i, j ifir, ilas, len
      character          cckwrk(*)*(*)

c==== list species included in mechanism to output file =====================

      do i = 1, NKK
         ifir = IFIRCH(cckwrk(IcKK+i-1))
         ilas = ILASCH(cckwrk(IcKK+i-1))
         len = ilas-ifir+1
         write (ldata) len
         write (ldata) ( cckwrk(IcKK+i-1)(ifir:ilas) )
      enddo

      return
```

```
      end
c==== End of subroutine write_species ==================================


C-----------------------------------------------------------------------C
      Subroutine READ_CADPL(LCADPL, LOUT)
C
C  BEGIN PROLOGUE
C
C  Read cad-pressure-length input files and store information into arrays
C
C     Input:  LCADPL, and LOUT are indexes of input/output files
C
C
C  END PROLOGUE
C*****precision > double
      IMPLICIT DOUBLE PRECISION (A-H,O-Z), INTEGER (I-N)

      CHARACTER LINE*80, SUB(80)*80, UPCASE*10
      LOGICAL exist
      double precision length(720)
      integer cadInit
      DIMENSION speed(1),cad(720), pressure(720)
      COMMON /CADPLDAT/ speed, cad, pressure, length, cadInit

CC
CC READ CAD-PRESSURE-LENGTH INPUT DATA
CC
      exist = .FALSE.
   10 continue
      read(LCADPL,'(A)',END=5000) LINE
      ILEN = IPPLEN(LINE)
      if( ILEN .EQ. 0) GOTO 10
      call CKISUB(LINE(:ILEN), SUB, NSUB)
C
C  Read engine speed
C
      if(UPCASE(SUB(1),5) .NE. 'SPEED') stop 'Please put engine speed(RPM)
     : in the first line of input file (cad-pl)'
      call IPPARR(SUB(2),1,1,speed,nfound,ier,LOUT)
      if(ier .eq. 2) stop 'Engine speed is missing in file cad-pl'
      write(LOUT,9000)
      write(LOUT,9010) speed
      exist = .TRUE.
C
C  Read crankangle(degree)-pressure(n/m^2)-length(m) data
C
      write(LOUT,9020)
      do i = 1 , 720
   20 continue
      read(LCADPL,'(A)',END=5000) LINE
      ILEN = IPPLEN(LINE)
      if(ILEN .EQ. 0) GOTO 20

      call CKISUB(LINE(:ILEN), SUB, NSUB)
      call IPPARR(SUB(1),1,1,cad(i),nval,ier,LOUT)
      call IPPARR(SUB(2),1,1,pressure(i),nval,ier,LOUT)
      call IPPARR(SUB(3),1,1,length(i),nval,ier,LOUT)
      write(LOUT,9030)cad(i),pressure(i),length(i)
      if(NSUB .NE. 3) stop 'Something is missing in cad-pl file'
      enddo

 5000 if(.NOT. exist) stop 'Check cad-pl file, it is empty !'
```

```
      write(LOUT, '(A,//)') '----------- End of cad-pl file -------------'
      close(LCADPL)


 9000 format('* Length : distance between cylinder head and piston *')
 9010 format(' Engine Speed :', 1PE12.4)
 9020 format('__Crankangle___Pressure___Length___')
 9030 format(3(1PE12.4))
      end

C-----------------------------------------------------------------------C
      SUBROUTINE CKISUB (LINE, SUB, NSUB)
C
C     Generates an array of CHAR*(*) substrings from a CHAR*(*) string,
C     using blanks or tabs as delimiters
C
C     Input:  LINE  - a CHAR*(*) line
C     Output: SUB   - a CHAR*(*) array of substrings
C             NSUB  - number of substrings found
C     A '!' will comment out a line, or remainder of the line.
C                                     F. Rupley, Div. 8245, 5/15/86
C-----------------------------------------------------------------------C
C*****precision > double
       IMPLICIT DOUBLE PRECISION (A-H,O-Z), INTEGER (I-N)
C*****END precision > double
C*****precision > single
C      IMPLICIT REAL (A-H,O-Z), INTEGER (I-N)
C*****END precision > single
C
      CHARACTER*(*) SUB(*), LINE
      NSUB = 0
C
      DO 5 N = 1, LEN(LINE)
        IF (ICHAR(LINE(N:N)) .EQ. 9) LINE(N:N) = ' '
    5 CONTINUE
C
      IF (IPPLEN(LINE) .LE. 0) RETURN
C
      ILEN = ILASCH(LINE)
C
      NSTART = IFIRCH(LINE)
   10 CONTINUE
      ISTART = NSTART
      NSUB = NSUB + 1
      SUB(NSUB) = ' '
C
      DO 100 I = ISTART, ILEN
         ILAST = INDEX(LINE(ISTART:),' ') - 1
         IF (ILAST .GT. 0) THEN
            ILAST = ISTART + ILAST - 1
         ELSE
            ILAST = ILEN
         ENDIF
         SUB(NSUB) = LINE(ISTART:ILAST)
         IF (ILAST .EQ. ILEN) RETURN
C
         NSTART = ILAST + IFIRCH(LINE(ILAST+1:))
C
C        Does SUB have any slashes?
C
         I1 = INDEX(SUB(NSUB),'/')
         IF (I1 .LE. 0) THEN
```

```
            IF (LINE(NSTART:NSTART) .NE. '/') GO TO 10
            NEND = NSTART + INDEX(LINE(NSTART+1:),'/')
            IND = INDEX(SUB(NSUB),' ')
            SUB(NSUB)(IND:) = LINE(NSTART:NEND)
            IF (NEND .EQ. ILEN) RETURN
            NSTART = NEND + IFIRCH(LINE(NEND+1:))
            GO TO 10
         ENDIF
C
C        Does SUB have 2 slashes?
C
         I2 = INDEX(SUB(NSUB)(I1+1:),'/')
         IF (I2 .GT. 0) GO TO 10
C
         NEND = NSTART + INDEX(LINE(NSTART+1:),'/')
         IND = INDEX(SUB(NSUB),' ') + 1
         SUB(NSUB)(IND:) = LINE(NSTART:NEND)
         IF (NEND .EQ. ILEN) RETURN
         NSTART = NEND + IFIRCH(LINE(NEND+1:))
         GO TO 10
  100 CONTINUE
      RETURN
      END


C-------------------------------------------------------------------------------
C
      CHARACTER*(*) FUNCTION UPCASE(ISTR, ILEN)
C
C  START PROLOGUE
C
C  FUNCTION UPCASE(ISTR, ILEN)
C    return an uppercase character string according to the assigned length
C
C  INPUT
C     ISTR      -  Input character string
C                  data type - character array
C     ILEN      -  length of the character string returned
C                  data type - integer
C
C  OUTPUT
C     UPCASE    -  Returned uppercase character string
C                  data type - character array
C
C  END PROLOGUE
C
      CHARACTER ISTR*(*), LCASE(26)*1, UCASE(26)*1
      DATA LCASE /'a','b','c','d','e','f','g','h','i','j','k','l','m',
     1            'n','o','p','q','r','s','t','u','v','w','x','y','z'/,
     2     UCASE /'A','B','C','D','E','F','G','H','I','J','K','L','M',
     3            'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'/
C
      UPCASE = ' '
      UPCASE = ISTR(:ILEN)
      JJ = MIN (LEN(UPCASE), LEN(ISTR), ILEN)
      DO 10 J = 1, JJ
         DO 10 N = 1,26
            IF (ISTR(J:J) .EQ. LCASE(N)) UPCASE(J:J) = UCASE(N)
   10 CONTINUE
      RETURN
      END
```

Bibliography

[1] Hochgreb, S. "Combustion-Related Emissions from Spark-Ignition Engines". in *Handbook of Air Pollution from Internal Combustion Engines*, Sher, E., Ed., Academic Press, 1998.

[2] Oliveira, I. and Hochgreb, S. "Effect of Operating Conditions and Fuel Type on Crevice HC Emissions: Model Results and Comparison with Experiments". In preparation.

[3] Wu, K. and Hochgreb, S. "Numerical Simulation of Post-Flame Oxidation of Hydrocarbons in Spark-Ignition Engines". *SAE Transaction Paper* 970886, 1997.

[4] The Numerical Algorithms Group, Limited. *The NAG Fortran Libraries, Mark 16*, 1993.

[5] Hindmarsh A. "OdePack, A Systematic Collection of ODE Solvers". *Scientific Computing*, Stepleman, R., S. (ed.), North-Holland, Amsterdam, pp. 55-64, 1983.

[6] Marchese, A., J. and Dryer, F., L. "The Effect of Liquid Mass Transport on the Combustion and Extinction of Bicomponent Droplets of Methanol and Water". *Combustion and Flame*, 105: 104-122, 1996.

[7] Cho, S., Y., Yetter, R., A., Dryer, F., L. "A Computer Model for One-Dimensional Mass and Energy Transport in and around Chemically Reacting Particles, Including Complex Gas-Phase Chemistry, Multicomponent Molecular Diffusion, Surface Evaporation, and Heterogenous Reaction". *Journal of Computational Physics*, 102: 160-179, 1992.

[8] Larrotourou, B. "Adaptive Numerical Simulation of Premixed Flame Propagation". in *Numerical Modeling in Combustion*, Taylor and Francis, 1993.

[9] Poulos, S., G. and Heywood, J. B. "The Effect of Chamber Geometry on Spark Ignition Engine Combustions". *SAE Paper* 830334, 1983.

[10] Kee, R. J., Rupley, F., M., and Miller, J. A. "CHEMKIN-II: A FORTRAN Chemical Kinetics Package for the Analysis of Gas-Phase Chemical Kintics". *Sandia Report*, SAND89-8009, 1989.

[11] Kee, R. J., Warnatz, J., and Miller, J. A. "A FORTRAN Computer Code Package for the Evaluation of Gas-Phase Viscosities, Conductivities, and Diffusion Coefficients". *Sandia Report*, SAND83-8209, 1988.

[12] Hirata, M., Ohe, S., and Nagahama, K. *Computer-Aided Data Book of Vapor-Liquid Equilibria.* Kodansha Sci., 1975.

[13] Press, W., H., Teukolsky, S., A., Vetterling, W., T., and Flannery, B., P. *Numerical Recipes in Fortran 77, The Art of Scientific Computing.* Cambridge University Press, 1992.

[14] Williams, F., A. *Combustion Theory.* Second Edition, Addison-Wesley, 1985.

[15] Cho, S. Y., Choi, M. Y., and Dryer, F., L. *Twenty-Third Symposium (International) on Combustion*, The Combustion Institute, Pittsburgh, 1990, p. 1611.

[16] Chen, Y.-J. "Reduced Reaction Mechanisms for Methanol-Air Diffusion Flames". *Combustion Science and Technology*, Vol 78, pp. 127-145, 1991.

[17] Smith, B. D., and Srivastava, R. *"Thermodynamic Data for Pure Compounds, Part B: Halogenated Hydrocarbons and Alcohols".* Elsevier, Amsterdam, 1986.

[18] Fermi, E. *"Thermodynamics".* Reprinted by Dover Publications, New York, 1956.

THESIS PROCESSING SLIP

FIXED FIELD: ill. _____ name _____

index _____ biblio _____

▶ COPIES: (Archives) Aero Dewey (Eng) Hum
Lindgren Music Rotch Science

TITLE VARIES: ▶ ☐ _____

NAME VARIES: ▶ ☑ Borges

IMPRINT: (COPYRIGHT) _____

▶ COLLATION: 181 P

▶ ADD: DEGREE: _____ ▶ DEPT.: _____

SUPERVISORS: _____

NOTES:

cat'r: date:

page: J 140

▶ DEPT: M.E.

▶ YEAR: 1999 ▶ DEGREE: S.M.

▶ NAME: OLIVEIRA, Ivan B.