This is to certify that the

thesis entitled

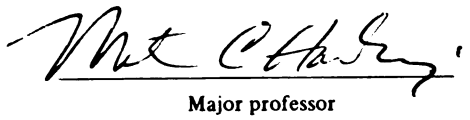
THREE DIMENSIONAL MODELLING AND SIMULATION OF THE CURING OF POLYMER COMPOSITES

presented by

Ananthapadmanaban Sundaram

has been accepted towards fulfillment
of the requirements for

M.S. degree in Chemical Engineering



Major professor

Martin C. Hawley

Date June 28, 1994

# THREE DIMENSIONAL MODELLING AND SIMULATION OF THE CURING OF POLYMER COMPOSITES

By

*Ananthapadmanaban Sundaram*

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Chemical Engineering

1994

# ABSTRACT

## THREE DIMENSIONAL MODELLING AND SIMULATION OF THE CURING OF POLYMER COMPOSITES

By

*Ananthapadmanaban Sundaram*

Curing of composite materials is an important process in enhancing the properties of the material for its final application. Modeling of the curing process is required to predict the variation of the different properties of the material and help control the process better. A three dimensional dynamic model of the curing process over materials of complex shapes has been developed using the Boundary Fitted Coordinate System for shape modeling. The developed model has been implemented in numerical simulation which can simulate the shape of the material as well as solve the thermal curing process model over the complex shapes. The simulation software has been developed using the C programming language and an output interface has been developed with the MATLAB(R) external library toolbox. The simulation results have been verified with different analytical models for certain cases and a qualitative treatment of different results in the three dimensional domain has also been presented.

# ACKNOWLEDGMENTS

I express my profound gratitude to my advisor Professor Martin Hawley for his guidance, encouragement and support throughout the course of this work. I have at different points of time sought the help of a number of people most notably Dr. James McDowell who has provided a number of suggestions that have gone into my work. Dr. Jianghua Wei provided me with an initial understanding of the problem and I am grateful to him for that. I wish to express my thanks to my colleagues in the research group, Valerie Adegbite and Dhulipala Ramakrishna as well as my friend Sanjay Mishra who have helped me greatly on and off the project. I am always indebted to my parents, who have always been my source of strength and inspiration in life.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF SYMBOLS

| Symbol(s) | Meaning |
|---|---|
| (x,y,z) | Physical coordinates |
| $(\xi, \eta, \zeta)$, (i,j,k), (u,v,w) | Natural coordinates |
| t | Time. |
| T | Temperature |
| $T_0$ | Ambient or surrounding temperature. |
| $T_i$ | Initial temperature. |
| X | Cure or extent of reaction |
| $P_m$ | Microwave power absorbed per unit volume |
| $r_c$ | Rate of cure reaction. |
| r | Radius. |
| $\alpha_{ij}$ | Element (ij) of shape factor matrix. |
| M | Shape transformation matrix. |
| $\beta_{ij}$ | Cofactor of element (ij) of M. |
| $L_h$ | Finite difference operator |
| $u_h$ | Vector of dependent variables. |
| $R_h^{2h}$ | Restriction operator. |
| $I_{2h}^h$ | Prolongation operator. |
| $\vec{q}$ | Heat flux |
| P,Q,R | Adaptive meshing functions |

| Symbol(s) | Meaning |
|---|---|
| J | Jacobian of transformation. |
| $\rho$ | Density of material. |
| $c_p$ | Specific heat. |
| $\kappa$ | Thermal conductivity |
| $h_c$ | Heat transfer coefficient |
| h | Grid size |
| n,m,l | Number of grid points along $\xi, \eta, \zeta$. |
| $\Gamma$ | Boundary in physical coordinate system |
| $\Omega$ | Boundary in natural coordinate system |
| $\Delta H_r$ | Heat of reaction. |
| $\Delta\xi, \Delta\eta, \Delta\zeta$ | Spatial steps in natural coordinates |

# CHAPTER 1

# Introduction

## 1.1 Motivation

Thermoset polymer composites are processed under a variety of different methods based on the nature of the final products, as well as the characteristics of the different components which make up the composite itself. The common basis for most of these methods involves driving the thermosetting or cross-linking reaction to completion, to obtain better physical and mechanical properties of the cured material. Different phenomena come into play during the curing process which critically depend upon the process conditions. The reaction is initiated by supplying heat to the material, in the form of thermal heat sources which propagate heat by use of temperature gradients, or by focussing heat into the material by use of other energy sources such as microwaves or radio waves.

Irrespective of the particular phenomena of heat transfer, the properties of the material being processed could be considered as varying with respect to the extent of the thermoset reaction, as well as the temperature. In practical industrial applications thick section composites need to be processed, which have varying temperature and cure gradients along different directions which change dynamically as the reaction proceeds to completion. Hence, the critical system parameters of, temperature and cure are dynamic functions in three di-

mensional space, and the domain of definition is the geometry of the material.

Any process model which describes these variables, does so, by relating the rate of change of these variables to the different physical phenomena at play in the process. These phenomena are in turn related to the different process parameters, depending upon the actual processing methods used. Hence, a generalized model provides the framework for predicting the critical system parameters of temperature and cure independent of the processing conditions as well as the geometry of the material. The specifics of these conditions are treated as various inputs, relevant to the process being considered. Two important aspects of the material affect the nature of the curing process. These are the geometry and the anisotropy of the system with respect to the properties of the material. This work essentially deals with the effects of shape and geometry only under isotropic conditions though the properties are allowed to vary as a function of the dependent variables under this framework.

Controlling the different processes, involves predicting the nature and magnitude of the temperature and cure profiles. The main motivation for this work is the automation of the control of curing processes which is able to take decisions based on generalized predictive methods at a higher level, and base the lower level reactive control on the actual specifics of the process being employed. The next major motivation for the effort, comes from the fact that a generalized model, and, a simulation based on it would essentially encompass all the different phenomena influencing the process, as well as incorporate into it the flexibility to include different sub-models. This would retain the specific information about the kinetics and the dependencies of the material properties on process parameters, as well as, the mode of application of heat such as thermal or microwave. Such a simulation would vastly enhance an automated control system which forms an essential unit of an expert system for the design and processing of polymer composites. It should be noted, that as different sub-models become available, like the model for flow, or the model for microwave

power absorption, these should be readily incorporated into the generalized model.

## 1.2 Background

### 1.2.1 Previous Modelling Efforts

The previous modelling and simulation efforts in this area, have been vastly based on one and two dimensional models (Bogetti & Gillespie [1]). These models though sufficient for the purposes of the application they were put to, are insufficient to characterize the complexities involved when the processing of thick section composites, of arbitrary shapes are considered . Essentially, by accounting only for one or two dimensions the predictions that are provided by these models do not include information of the gradients in the other direction(s) and hence also the interactions between the gradients in different directions. This phenomenon would be especially amplified in cases when, there is a preferential curing of the composite using different boundary conditions at the different surfaces. Also, in cases where hybrid sources, that is both thermal and microwave are used for the curing process, the different boundary conditions become especially critical though this case has not been handled by the current effort.

### 1.2.2 Fundamental Concepts

The basis of the process model are the energy and material balance equations in three dimensions, which are coupled in different ways depending upon the type of process employed for curing, as well as the nature and characteristics of the material being cured.

The basic approach for a modelling effort of this type consists of the following steps.

1. Modelling the domain of definition.

2. Model the process in the original domain.

3. Transformation of the process model equations into the modelled domain.

4. Solve the transformed process model.

5. Representation of the solution in the original domain.

Normal modelling efforts tend to keep the domain model the same as the physical one or, at least approximately so. This situation is not possible when the domain is complex with irregular boundaries. Extensive information about the exact shape of the domain is prohibitive on the time available, and the process models created in the original domain, cannot be easily solved accurately, in their original domain. Hence a method is required which marries the efficiency of the solution or simulation, with the nature of the problem being solved. The coordinates of the model domain should, in essence, reflect the natural dependencies of the process parameters in space.

A finite difference approach, tends to model the domain by assuming that the entire domain is composed of regular patterns of certain shapes such as rectangular, triangular, spherical etc, depending upon the accuracy of the representation using such a pattern, as well as the coordinate system used to represent the model equations. A finite element method on the other hand divides the domain into a number of elements that are polynomially dependent on regular coordinates. This gives the flexibility of handling complex shapes not suitably solved by the finite difference methods. But there is an increase in the computational complexity and once again, very accurate domain representation is necessary for the implementation of the method. The major drawback of the finite element methods is, there is an uniform use of the same type of elements over the entire domain. The use of varying elements is not popular and also requires extensive information from the user for an useful implementation.

The next major obstacle in using any of the above methods for the model, is the representation of boundary conditions especially the conditions involving the flow of heat, mass

etc. Typically the flow through any element, is determined by evaluating the normal to the element and projecting the flow parallel to it. For an arbitrary shaped body, this involves approximating the normal derivatives and when the dimensionality of the problem increases, the accuracy of such a representation critically depends on the exactness of the boundary coordinate information provided by the user. For processes whose parameters critically depend on the nature of such boundary flow conditions, like the curing process, a good process simulation should provide, a flexible and accurate way of representing these conditions numerically, without overwhelming the user with the tedious responsibility of coordinate generation for the complex domain.

Finally, from a framework aspect any simulation of a model, of the kind being solved in this effort, should include the necessary tools to extend the same to more general conditions. In summary, the programming framework should provide constructs, that would help use different sub-models of varying levels of complexity. For example, under the current implementation, the diffusional effects in the mass transfer are considered not limiting and the concentration profile is solely dependent on the reaction under progress. But an addition of the diffusion phenomena, could be done essentially with little work, without going through the entire process of formulation, representation and solution. Since there exists, a fundamental analogy between the various phenomena being solved, i.e mass, momentum and heat , modelling one with the necessary generality, would imply, that any of the others can be implemented in addition to, or independently, under the same programming framework. This is a very important aspect of a good simulation and the necessary framework has been provided in the current effort.

## 1.3 Objectives

The previous section brought out the main objectives of the modelling effort. The modelling goals are

1. Evolve a shape modelling framework to model the arbitrary nature of shapes encountered in a curing process.

2. The shape model of the given domain should aid in a simple representation of the boundary conditions over the surface of the domain, as these critically affect the accuracy of the simulation solution.

3. Provide a simulation framework using basic constructs, that help to incorporate various phenomena and sub-models.

4. Provide an accurate and at the same time, convenient representation of the process equations for computationally efficient simulation in the modelled domain.

5. Evolve a solution framework, that is both general, in the sense of using different mathematical tools, and accurate for the problems being solved.

Figure 1.1 indicates the overall framework for the simulation effort. All the different aspects represented in bold indicate the work covered under the current effort. Enhancements that could be added to the current simulation framework have also been indicated in the figure.

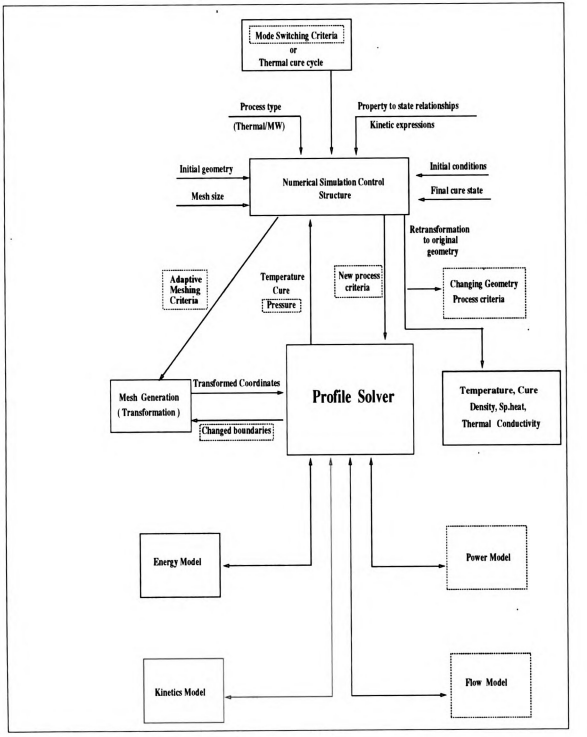



Figure 1.1. **Overall framework for cure process simulation**

# CHAPTER 2

# Shape and Process Modelling

## 2.1 The Process Model

Consider, a material of arbitrary shape undergoing the process of curing. Heat is supplied to the body in some manner and this causes the startup of the curing reaction within the body. The reaction releases exothermic heat and this further causes changes in the curing pattern within the body. Hence, the entire process of curing is a dynamic one and since, the heat propagation mechanism also involves conduction the curing pattern is also dependent on the location within the material, or in other words the spatial coordinates. There is no flow within the system and shape of the material is constant throughout the process. The basic equation governing this process is the energy balance equation, which accounts for the heat flux entering and leaving a differential element of the body over time. Under the above conditions and assumptions, this equation is written in standard notation as

$$\rho c_p \frac{\partial T}{\partial t} = -\vec{\nabla} \cdot \vec{q} + Q \tag{2.1}$$

In the above energy balance $\rho$ is the density of the material, $c_p$ is the specific heat, T is the temperature at any point within the material at a given time, t is the time coordinate, q is the

heat flux across any differential surface element inside the material, Q is the net rate of heat generation due to different phenomena. The first term in the above equation involves the conduction of heat. The material properties in the above equation can reflect the changes in the curing process by being dependent on the state of the material i.e., its temperature and extent of cross-linking. A Fourier type relation for the heat flux is assumed which relates the energy balance in heat flux terms to the temperature of the system under isotropic conditions. This is then given as

$$\vec{q} = -\kappa \vec{\nabla} T \tag{2.2}$$

In the above expression the term k is the thermal conductivity of the material and can be a function of the temperature and other process parameters. In the energy balance, Equation 2.1, the net heat generation term, Q specifically depends on the mode of supply of heat to the body. In the case of thermal cure, the application of the heat from the external source to the body is by convection and hence the term Q contains only the exothermic heat due to the cross-linking reaction. In the case of a microwave(MW) application of heat, this term contains two parts. The reaction heat part would be included because the reaction is still present. However, it has to be noted that the form and functionality of the reaction term may be different in the case of microwave curing for the same material, when compared with thermal curing. It is known that excitation of dipoles causes the evolution of heat during MW curing. This might influence the reaction mechanism itself by affecting the reaction centers at a molecular level leading to a change in the rate equation for the reaction in the MW case. dditionally, there is another term in Q which accounts for the energy absorbed due to the microwave. Unlike thermal means, which supply heat directly only to the bounding surfaces of the material by convection, the microwave radiation focusses heat into the body. The interactions of the dipoles of the body with the electromagnetic waves causes successive alignment and relaxation of these dipoles leading to an absorption and release of energy. This energy being absorbed from the MW, depends on the dielectric properties of

the material which are once again functions of the temperature, and the degree of cure of the material. Hence, the microwave power absorption by the body might be envisioned to be a function of the above mentioned parameters. Expanding the gradient operator in Equation 2.2, substituting into Equation 2.1 and rewriting the heat generation term in terms of its two constituents i.e reaction and MW parts we get the following equation in Cartesian coordinates.

$$\begin{aligned}\rho(T,X)c_p(T,X)\frac{\partial T}{\partial t} &= \frac{\partial}{\partial x}\left(\kappa\frac{\partial T}{\partial x}\right)+\frac{\partial}{\partial y}\left(\kappa\frac{\partial T}{\partial y}\right)+\frac{\partial}{\partial z}\left(\kappa\frac{\partial T}{\partial z}\right)\\ &\quad +\rho(T,X)r_c\,(-\Delta H)+P_m(T,X,x,y,z,t)\end{aligned}\tag{2.3}$$

In the above equation $r_c$ is the rate of the thermosetting reaction, expressed as time$^{-1}$, $\Delta$H is the heat of reaction and is negative for exothermic reaction, $P_m$ is the power being absorbed from the microwave radiation per unit area and X is the degree or extent of cure. In the case of conventional thermal curing the $P_m$ is zero and this will be assumed through the rest of the Thesis. The reaction rate needs to be accounted, in terms of the different process variables and this is given by an ordinary differential equation as follows.

$$\frac{dX}{dt} = r_c(X,T)\tag{2.4}$$

The above equation neglects diffusion, and hence explicit spatial dependency of the extent of reaction. The term $r_c$(X,T) is the rate equation of the thermosetting reaction available as a function of temperature and cure. The coupled Equations 2.3 and 2.4 form the mathematical process model for curing. The above equations are part of the boundary value problem for T and X. The boundary conditions for this problem come from the convective type boundary. The heat exchange at the bounding surfaces of the body being cured can be modelled as a

convection process. This leads to the following boundary condition,

$$\vec{n} \cdot \vec{q} = h\left(T - T_s\left(t\right)\right) \text{ on } \Gamma \tag{2.5}$$

where $\Gamma$ is the bounding surface of the arbitrary shaped body, $\vec{n}$ is the outward normal on the surface $\Gamma$, h is the heat transfer coefficient on the surface of the body, $T_s$ is the ambient temperature. In deriving the above boundary condition, the effect of radiation has been assumed to be negligible. Equations 2.3, 2.4 and 2.5 complete the process model for the curing process under the assumptions and conditions explained in this section.

## 2.2 The Boundary Fitted Coordinate System

Consider the mathematical model for the curing process as presented in the previous section. The system of equations is a second order partial differential equation(PDE) coupled to an ordinary differential equation(ODE). The domain of definition for this system is the geometry of the body. Consider now, the boundary conditions to this system of differential equations. They are defined over the entire bounding surface of the body. For a closed simply connected body, this would be the entire outer surface of the body. Since the body is arbitrary in shape, the surface $\Gamma$ is also a complex three dimensional surface. Equation 2.5 which is the boundary condition contains a term $\vec{n}$ which is the unit outward normal from the surface of the body. For an arbitrary shaped surface, the orientation of this normal vector is also arbitrary. Solution to this system of equations is essentially numerical, due to the nonlinearities involved and lack of functional representations for some of the parameters involved. Thus, the solution strategy would basically involve, the discretisation of the domain including the boundary into a number of elements. In such a case, the boundary normal has to be approximated by using the nearest neighbor elements. Such a procedure has to be carried over all the elements on the boundary. This often involves tedious computation

which is not very accurate. Besides, the boundary surface also needs to be approximated, usually by use of polynomial fits for the evaluation of the spatial derivatives involved in the boundary condition. These polynomial approximations do not work very well in a number of cases, due to the fact that these approximations themselves have to vary over the surface. In other words, different order polynomials may be required at different regions of the boundary for a good approximation, and the decision as to which to use, is not arbitrary. The Introduction chapter explained the different pitfalls of the finite difference and finite element formulations, as far as the shape modelling was concerned. Both these formulations use some, or all of the techniques explained above, for domain and boundary approximation. There is a lot of user effort involved in the decision making for the shape modelling aspect of the problem solving. The criteria used to make the decisions involved in finite element and difference approximations of arbitrary shapes is a complex numerical exercise and no generic technique exists to evolve them. This is by far the most unattractive aspect in using either of these methods for complex shape modelling. Therefore, it is necessary that the complex geometry be represented in a regular manner by a suitable transformation, for the task of solving the process model. The tradeoff in such a case is the additional work, involved in re-formulating the process model in the new domain which introduces additional terms in the model. Consider now, the problem of transforming a complex geometry of three dimensions into a regular shape, say a cubical domain. The aim of this representation is to provide a mapping from each point on the surface of the irregular boundary, to a point on the surface of the regular transformed domain. The advantage of the transformation besides the fact that the domain becomes regular is that there is no necessity for normal evaluation on the boundaries of the regular domain. The entire domain being cubic, the normals are along the positive and negative transformed coordinate directions. Thus, given the surface coordinates of the original domain for the process model, a transformed domain is to be generated by some means, which also imposes, continuity along the coordinate directions

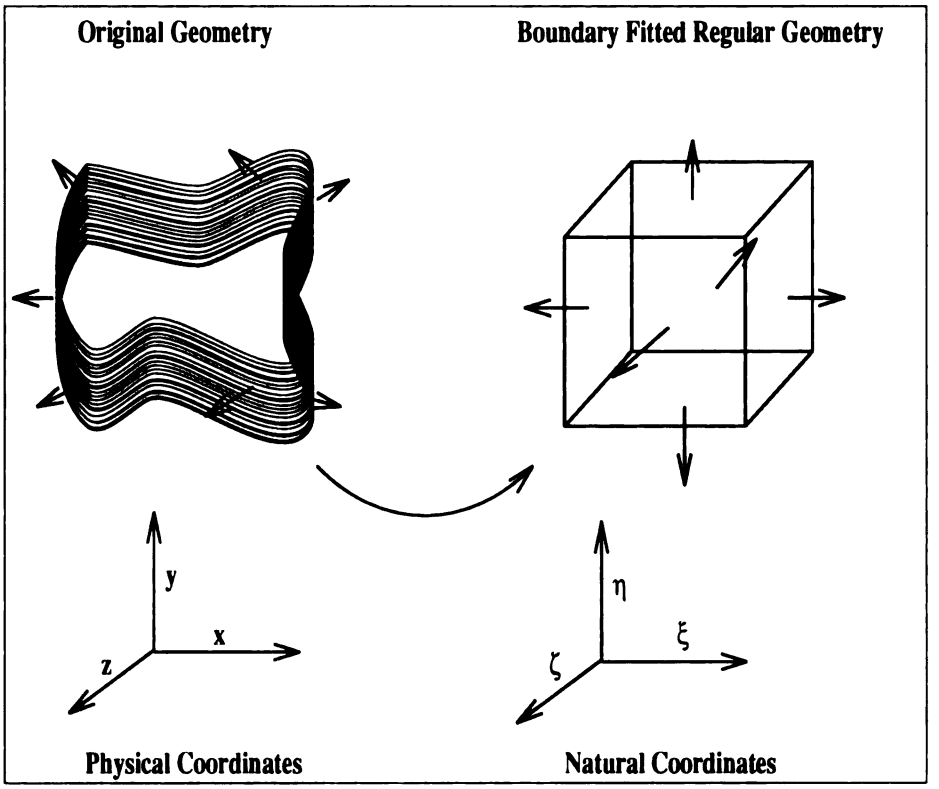



Figure 2.1. **Boundary Conforming Transformation**

within the interior. This problem is basically a Boundary Value problem to be solved for the transformed coordinates, also called the **Natural Coordinates**, given the surface coordinate values of the natural coordinates as a discrete function of the original or **Physical Coordinates** of the system. The resulting solution of natural coordinates as a function of physical coordinates defined over the entire domain is called the **Boundary Fitted Coordinate System**(BFCS). Figure 2.1 gives the pictorial representation of the method. In the analysis to be developed in the rest of the Thesis, the physical coordinates are denoted as (x,y,z) and the natural coordinates are denoted as ($\xi$,$\eta$,$\zeta$).

The boundary fitted coordinate system technique attempts to generate a set of natural coordinates $\xi$(x,y,z), $\eta$(x,y,z), $\zeta$(x,y,z) under certain constraints. As already mentioned, this

problem is a boundary value problem and different methods exist which can be applied to obtain a solution. The most common methods which are employed for this purpose are

1. Algebraic grid generation methods.

2. Elliptic grid generation methods.

Algebraic generation methods use algebraic interpolation techniques along different directions, using boundary information, to obtain a natural coordinate mapping for the given domain. Elliptic generation methods make use of elliptic equations of the natural coordinates, with the physical coordinates as the independent variables to solve for the same problem. But, the elliptic methods normally require an initial approximate solution which is then refined by use of a generating system of elliptic differential equations. That initial guess, is provided by the algebraic generation methods. The next two sections develop the details of the two methods in light of their implementation in the present work.

### 2.2.1 Algebraic Generation Systems

The shape modelling problem requires the generation of a regular rectangular grid, given the correspondence of the grid points on the bounding surfaces of the regular transformed domain, to the original boundary information on the irregular domain. Consider Figure 2.2 which shows the pictorial representation of the problem on a three dimensional surface. The natural coordinates assume integral values from (0,0,0) to (n,m,l) and let (i,j,k) be the coordinates of any point on the natural coordinate space spanned by ($\xi$,$\eta$,$\zeta$). The grid points are shown at their corresponding positions in the physical coordinate space. The figure shows a surface over which $\zeta$ is constant but not necessarily a bounding surface. The intermediate points in the figure are the interior grid points on the regular domain. Corresponding to every value (i,j,k) on the regular grid, there exists a value (x(i,j,k),y(i,j,k),z(i,j,k)). The

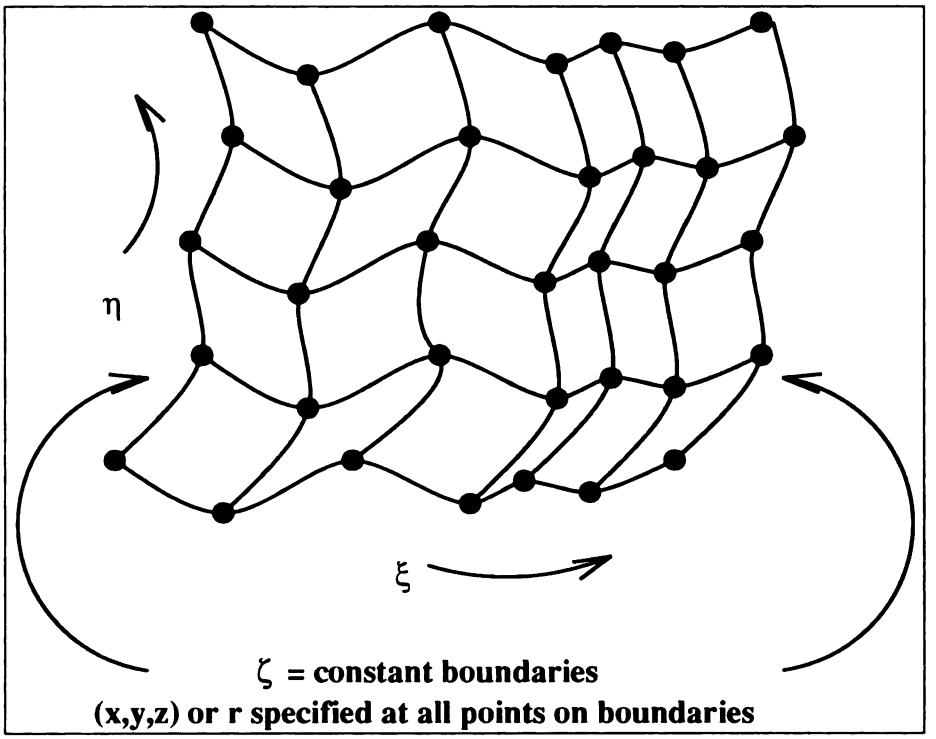



Figure 2.2. **Interpolation problem in two dimensions**

problem is to find the triplets (x,y,z) for every value of (i,j,k), given the values at all the extreme points. Since each of the physical coordinates x, y or z, are exclusive functions of (i,j,k) the problem is nothing, but one of three dimensional interpolation. The method used to perform this three dimensional interpolation in this simulation is called **Transfinite Interpolation**. The basis of this method is independent interpolation along three directions. Consider the three dimensional grid shown in the Figure 2.3. Let $\phi_\eta$ interpolate the vector $\vec{r}(\xi,\eta,\zeta)$ along the $\eta$ direction. For the purpose of this interpolation, any standard technique such as cubic splines, B-spline, Hermite interpolation can be used. Transfinite interpolation performs the following operations

$$\vec{E}_1 = \phi_\xi(\vec{r}) \tag{2.6}$$

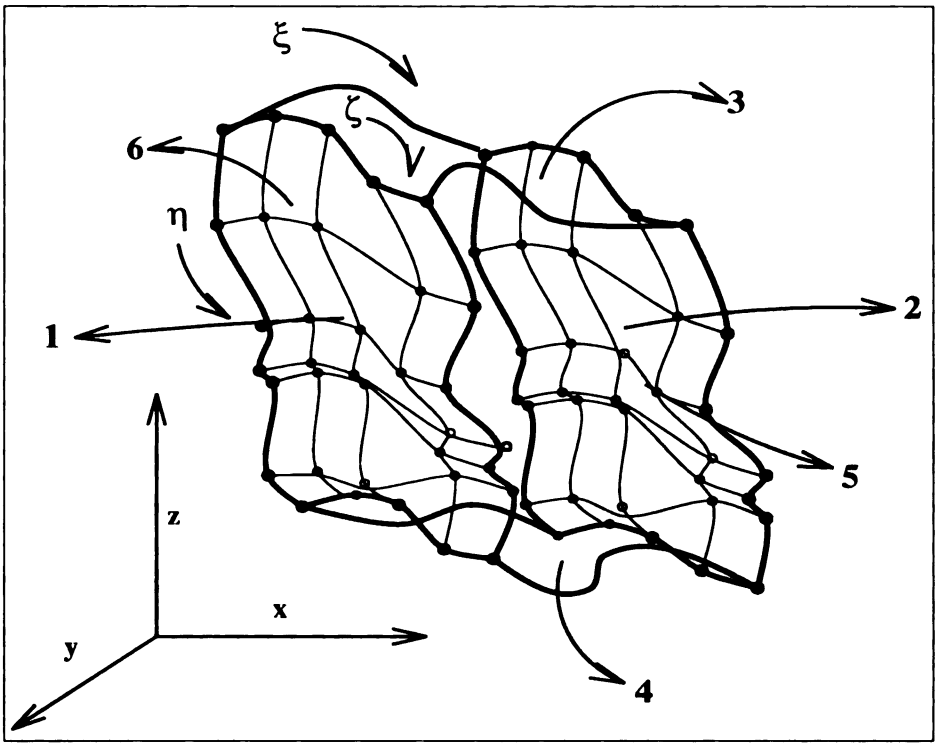



Figure 2.3. **Natural to Physical Coordinate Line Mapping**

$$\vec{E}_2 = \phi_\eta(\vec{r} - \vec{E}_1) \tag{2.7}$$

$$\vec{E}_3 = \phi_\zeta(\vec{r} - \vec{E}_1 - \vec{E}_2) \tag{2.8}$$

$$\vec{r}(\xi, \eta, \zeta) = \vec{E}_1 + \vec{E}_2 + \vec{E}_3 \tag{2.9}$$

In the steps above, $\vec{r}$ refers to a vector in the physical coordinate system with components x, y, z respectively along the three principal directions. The Figure 2.3 indicates the different bounding surfaces on which $\xi$, $\eta$ and $\zeta$ are constant, respectively. Step 2.7 performs interpolation along the $\xi$ direction across the $\eta\zeta$ surfaces. This step yields the value of the dependent vector $\vec{r}$ along the $\xi$ coordinate lines on all surfaces including the boundaries indicated as 1 and 2 in the figure. The next step 2.8 calculates the error $(\vec{r} - \vec{E}_1)$ in the one

dimensional interpolation on the boundaries 1 and 2 and interpolates this error along the $\eta$ direction. It is to be noted, that the sum of the vectors $\vec{E}_1$ and $\vec{E}_2$ exactly matches the boundary coordinate information on both $\xi$=constant and $\eta$=constant surfaces. The next step interpolates the total error ($\vec{r} - \vec{E}_1 - \vec{E}_2$) due to the two previous interpolations, along the $\zeta$ coordinate lines, using the values of this error vector on the boundaries 3 & 4 where $\zeta$ is constant. As before, the sum of all the three vectors $\vec{E}_1$, $\vec{E}_2$, $\vec{E}_3$ exactly matches the supplied coordinate information on all the bounding surfaces. Thus the transfinite interpolation, uses standard one dimensional interpolation techniques to perform three dimensional grid generation. The advantage of this technique is that, one can use any known one dimensional interpolation technique depending upon the nature of the shape and the constraints on the continuity involved. The code developed for algebraic grid generation in this effort makes use of linear interpolation as only two points, one on each opposing boundary is specified for every coordinate line. Sometimes, even in a three dimensional algebraic grid generation only the edge and corner point coordinate information is available. In such cases a two dimensional transfinite interpolation is performed on all the six bounding surfaces first and then a three dimensional transfinite interpolation is performed using the different steps as explained above. For the general case of three dimensional problems algebraic grid generation does not provide a good mapping, in the sense that sometimes the interior point mappings are not accurate and might appear outside the boundaries. Thus, the algebraic grid generation is not a good stand alone technique for mapping physical coordinates to natural coordinates. In the present formulation an elliptic generating system which makes use of an algebraic grid as an initial guess and refines the solution using continuity and boundary constraints on the natural coordinates has been used to perform the grid generation.

### 2.2.2 Elliptic Generation Systems

Elliptic generation systems are natural coordinate systems generated by solving for a set (as many as the dimensions involved) of coupled elliptic partial differential equations, constructed as a boundary value problem, using the boundary physical coordinate information as the constraining boundary conditions. The idea behind the elliptic generation systems is to evolve natural coordinate surfaces along which one coordinate remains constant, which is similar to a problem of generating stream functions. In this case however the boundary conditions on these natural coordinates or streamlines are the boundary coordinate information of the object. Elliptic generation systems are guaranteed to provide one to one mapping when the generating equations are Laplacian (Mastin & Thompson [2]). The fundamental basis of the Laplace generating equations is the Eulerian equation for the error (Thompson et. al. [3]) given by the following expression.

$$I = \int_x \int_y \int_z \sum_{\psi_i = \xi, \eta, \zeta} \left( \nabla \vec{\psi}_i \right)^2 dV \tag{2.10}$$

where $\psi_i$ are the natural coordinates. This error is minimized under the following conditions for the natural coordinates given as the Laplacian system of equations,

$$\begin{aligned} \nabla^2 \xi &= 0 \\ \nabla^2 \eta &= 0 \\ \nabla^2 \zeta &= 0 \end{aligned} \tag{2.11}$$

The boundary conditions to the above system of equations are as mentioned earlier, the known surface coordinate information of the domain. These are given as

$$\xi = \xi_\Gamma(x, y, z) \text{ on } \Gamma$$

$$\begin{aligned} \eta &= \eta_\Gamma(x, y, z) \text{ on } \Gamma \\ \zeta &= \zeta_\Gamma(x, y, z) \text{ on } \Gamma \end{aligned} \tag{2.12}$$

It is to be noted that, the functionalities above are normally available only as discrete values at different points. Thus, the accuracy of surface coordinate information determines the spacings of the grid on the transformed domain. The system of Equations 2.11 with the boundary conditions 2.12 form the elliptic generation system of equations. The characteristic of the Laplacian system of equations is that, the generated solution has continuous second derivatives and smooth coordinate lines.

The system of equations as defined above is defined on the original domain of the body. Since the body is of arbitrary geometry, once again there is a problem of irregular coordinate lines and surfaces. Thus, the equations formulated to avoid the above problem, themselves have to be solved in a similar domain. This is overcome by reformulating the above equations with a change in the dependent variables. As already stated equations 2.11 and 2.12 provide the solutions $\xi$(x,y,z), $\eta$(x,y,z), $\zeta$(x,y,z). By switching the dependent and independent variables the equations can be transformed to be solved for x($\xi$,$\eta$,$\zeta$), y($\xi$,$\eta$,$\zeta$), z($\xi$,$\eta$,$\zeta$). This is the same as reformulating the problem with a change of the coordinate system from the covariant base vectors to contravariant base vectors. However the two coordinate base vectors need not be parallel as it happens in orthogonal coordinate systems. The advantage of this transformation is the fact that the new set of equations are defined on a regular rectangular geometry, and hence standard numerical techniques can be used to solve these differential equations, without the need for approximate interpolation or fitting. The necessary analysis that has to be used to obtain this new set of equations for the three dimensional case can be found in [2] . Basically, this involves determining the matrix of the transformation, in terms of the derivatives of the natural coordinates with respect to each of the physical

coordinates. Upon such a transformation the following set of differential equations result.

$$\begin{aligned}
\alpha_{11}x_{\xi\xi} + 2\alpha_{12}x_{\xi\eta} + 2\alpha_{13}x_{\xi\zeta} + 2\alpha_{23}x_{\eta\zeta} + \alpha_{33}x_{\zeta\zeta} &= 0 \\
\alpha_{11}y_{\xi\xi} + 2\alpha_{12}y_{\xi\eta} + 2\alpha_{13}y_{\xi\zeta} + 2\alpha_{23}y_{\eta\zeta} + \alpha_{33}y_{\zeta\zeta} &= 0 \\
\alpha_{11}z_{\xi\xi} + 2\alpha_{12}z_{\xi\eta} + 2\alpha_{13}z_{\xi\zeta} + 2\alpha_{23}z_{\eta\zeta} + \alpha_{33}z_{\zeta\zeta} &= 0
\end{aligned} \tag{2.13}$$

where the $\alpha_{ij}$ are the shape factors of the transformation and are defined as

$$\alpha_{ij} = \sum_{m=1}^{3} \beta_{mi}\beta_{mj};\ \text{i=1,2,3;}\ \text{j=1,2,3} \tag{2.14}$$

and $\beta_{jk}$ is the cofactor of the element in the position (j,k) of the matrix M, which is defined by the expression

$$M = \begin{pmatrix} x_\xi & x_\eta & x_\zeta \\ y_\xi & y_\eta & y_\zeta \\ z_\xi & z_\eta & z_\zeta \end{pmatrix} \tag{2.15}$$

The boundary conditions 2.12 are transformed by the change of dependent variables as,

$$\begin{aligned}
x &= x_\Omega(\xi,\eta,\zeta) \text{ on } \Omega \\
y &= y_\Omega(\xi,\eta,\zeta) \text{ on } \Omega \\
z &= z_\Omega(\xi,\eta,\zeta) \text{ on } \Omega
\end{aligned} \tag{2.16}$$

where $\Omega$ is the regular boundary(six bounding surfaces) of the regular domain as indicated in Figure 2.1. The coefficients in the differential Equations 2.13 are variable as they are functions of the dependent variables, but the nature of the equations is still elliptic. That is, the transformation does not change the nature of the original formulation. Any numerical method used to solve elliptic equations with nonlinear coefficients is applicable for this

system of equations. The solution to the above equations yields the values of x, y, z at the different grid points on the regular or natural coordinate system. But the original process model is defined over the irregular domain. The mathematical formulation of the process model needs to be transformed into the regular domain before it is solved. The transformation of the different terms in the process model 2.3 and the process boundary conditions (Equation 2.5) depends on the different derivatives involved. The next section details these transformations and presents the process model in the natural coordinates.

## 2.3 Process Model Transformation

The process model presented in Section 2.1 is defined over the physical coordinate system. The next section presented the transformation of the domain from physical to natural coordinates. Correspondingly, the process model should also be represented in its equivalent form in the natural coordinate system using the solution to the elliptic generating system (Equations 2.13 and 2.16). The boundary fitted coordinate generation technique actually obtains the natural coordinates as functions of the physical coordinates using the contravariant (normal to coordinate surfaces) base vectors of the system. The curvilinear coordinate lines of the three dimensional system are space curves formed by the intersection of surfaces on which one of the coordinates is constant. Thus, along coordinate lines only one coordinate varies and the others are constant. The various operators such as the gradient, the Laplacian, divergence etc can now be redefined in terms of the base vectors. These base vectors are in turn, related to the physical coordinates through the shape transformation matrix M and the shape factors $\alpha$s which were defined in he previous section. Table 2.1 gives the transformation relations for the different operators involved in the original process model in terms of the covariant and contravariant base vectors as well as the relations of these base vectors to the physical coordinates. In the table the variable $\xi^i$ can be $\xi$, $\eta$ or $\zeta$ when i = 1, 2 or 3. All

Table 2.1. **Transformation Relations**

| *Name* | *Symbol* | *Transformation/Definition* |
|---|---|---|
| Jacobian | J | det—M— |
| Covariant base vector | $\vec{a_i}$ | $x_{\xi^i}\vec{e_1} + y_{\xi^i}\vec{e_2} + z_{\xi^i}\vec{e_3}$ |
| Contravariant base vector | $\vec{a^i}$ | $\frac{1}{J}(\beta_{i1}\vec{e_1} + \beta_{i2}\vec{e_2} + \beta_{i3}\vec{e_3})$ |
| Gradient | $\vec{\nabla}A$ | $\sum_{i=1}^{3} \vec{a^i} A_{\xi^i}$ |
| Divergence(A) | $\vec{\nabla} \cdot \vec{A}$ | $\sum_{i=1}^{3} \vec{a^i} \cdot \vec{A_{\xi^i}}$ |
| Laplacian | $\nabla^2 A$ | $\frac{1}{J^2}\sum_{i=1}^{3}\sum_{j=1}^{3} \alpha_{ij} A_{\xi^i\xi^j} + \sum_{k=1}^{3} (\nabla^2 \xi^j) A_{\xi^j}$ |
| Time Derivative | $\frac{\partial A}{\partial t}$ | $\frac{\partial A}{\partial t}$ |

the other variables have already been defined. The exact details of these transformations and their derivations can be found in [3]. The above transformations can be used in Equations 2.3 and 2.5 to obtain the differential equations and boundary conditions in the natural coordinates given by Equations 2.17 and 2.18 below. These equations are more complex than the original model in the sense that additional terms as well as nonlinearities have been introduced. But the nature of the differential equation is still the same and the boundary conditions do not contain anything more than the first derivatives. Besides, these equations are defined over a rectangular grid and hence any method employed to solve parabolic

$$\begin{aligned}\rho(T,X)c_p(T,X)\frac{\partial T}{\partial t} &= \frac{\alpha_{11}(\kappa T_\xi)_\xi + \alpha_{22}(\kappa T_\eta)_\eta + \alpha_{33}(\kappa T_\zeta)_\zeta}{J^2} \\ &+ \frac{\alpha_{12}\left\{(\kappa T_\xi)_\eta + (\kappa T_\eta)_\xi\right\}}{J^2} \\ &+ \frac{\alpha_{13}\left\{(\kappa T_\xi)_\zeta + (\kappa T_\zeta)_\xi\right\}}{J^2} \\ &+ \frac{\alpha_{23}\left\{(\kappa T_\eta)_\zeta + (\kappa T_\zeta)_\eta\right\}}{J^2} \\ &+ \rho(T,X)r(T,X)(-\Delta H) + P_m(T,X) \qquad (2.17)\end{aligned}$$

$$
\begin{aligned}
h_c T + \gamma_1 \frac{\kappa}{J\sqrt{\alpha_{11}}} \sum_{j=1}^{3} \alpha_{1j} T_{\xi^j} &= h_c T_s(t);\ \gamma_1 = \pm 1 \text{ when } \xi = 0 \text{ or n} \\
h_c T + \gamma_2 \frac{\kappa}{J\sqrt{\alpha_{22}}} \sum_{j=1}^{3} \alpha_{2j} T_{\xi^j} &= h_c T_s(t);\ \gamma_2 = \pm 1 \text{ when } \eta = 0 \text{ or m} \\
h_c T + \gamma_2 \frac{\kappa}{J\sqrt{\alpha_{33}}} \sum_{j=1}^{3} \alpha_{3j} T_{\xi^j} &= h_c T_s(t);\ \gamma_3 = \pm 1 \text{ when } \zeta = 0 \text{ or l}
\end{aligned} \tag{2.18}
$$

PDEs over regular domains can be used to solve the transformed set of equations. Once a solution is obtained, it can be mapped back to the original domain using the solution of the shape model. In the above development, it has been assumed that the shape model is time invariant i.e. the shape of the body does not change with respect to time. When flow problems need to be solved this assumption is not valid anymore and the boundaries themselves change with time depending upon the flow parameters such as pressure and viscosity of the system. In such a case, a shape model has to be solved at every step in time with the boundary coordinate information calculated from the changing geometry of the body being cured. In this case, the time derivatives of the process dependent variables need to be transformed in the formulation of the process model in natural coordinates. This relation is given as

$$
\left(\frac{\partial A}{\partial t}\right)_{\vec{x}} = \left(\frac{\partial A}{\partial t}\right)_{\vec{\xi}} - \vec{\nabla} A \cdot \left(\frac{\partial \vec{x}}{\partial t}\right)_{\vec{\xi}} \tag{2.19}
$$

In the above equation $\vec{x}$ is a vector in the physical coordinate system and $\vec{\xi}$ is the corresponding vector in natural coordinate system. The subscripts attached to the time derivatives indicate the variable being held constant in the partial differential equation. The time derivative on the right hand side is at a fixed position in the transformed space i.e, at a given grid point. Similarly the time derivative on the left hand side is at a fixed position in the physical space i.e., the time derivative that appears in the equations of motion. So, in the case of changing geometry the problem can essentially be solved on a fixed grid though the corresponding physical coordinates change over time.

# CHAPTER 3

# Numerical Solution

## 3.1 Finite Difference Formulations

The last chapter dealt with the different differential formulations involved in the shape modelling and process simulation aspects of the given problem. Different methods are available for the effective solution to these problems. However the geometry that is of concern in the transformed problem is a regular (cubical) one and hence the finite difference representation affords a simple and effective means of solving the equations over this domain. Several different kinds of formulations exist in determining these finite difference representations to be used. Since a finite difference representation essentially divides up the domain into discrete points along the different directions, two different methods exist, depending on the points at which the derivatives are evaluated. In a **Cell Centered** approach the variable values are assigned to the center of the cells while the derivative values are evaluated on the edges using the cell centered values. In the **Vertex Centered** approach, the derivative evaluation is performed on the vertices of the cell which are the grid points for this formulation. Figure 3.1 shows a two dimensional representation of the vertex and cell centered approaches. The next step in the finite difference formulation is, the approximation of the differentials using differences. This is done by using a local Taylor's series expansion about the grid

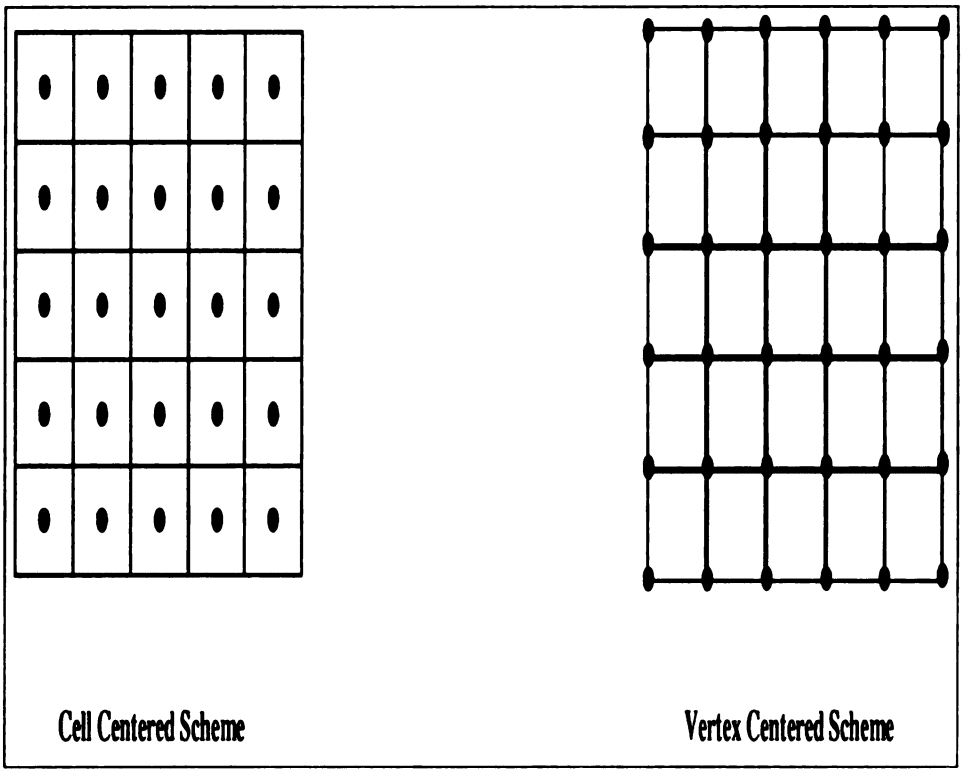



Figure 3.1. **Types of finite difference formulations**

point under consideration. When the expansions involve only the neighbors on one side of the grid point, one sided differences result. Besides, these Taylor's expansions are only of first order and hence they have an error term associated with them. But different one sided differences can be added together to eliminate some of the terms in the error to get second or higher order accurate differences. These are the central difference representations which almost always involve the values of the dependent variables on either side of the grid point at which the derivative is being determined. Thus, depending upon the centering of the grid points and the order of the difference formulations, the derivatives are approximated in different manners. The differential equations in the current effort have been approximated using a second order, vertex centered approach. The derivatives at the interior points of the

domain have been approximated using the following difference formulations.

$$\begin{aligned} f_\xi|_{i,j,k} &= \frac{f_{i+1,j,k} - f_{i-1,j,k}}{2\Delta\xi} \\ f_\eta|_{i,j,k} &= \frac{f_{i,j+1,k} - f_{i,j-1,k}}{2\Delta\eta} \\ f_\zeta|_{i,j,k} &= \frac{f_{i,j,k+1} - f_{i,j,k-1}}{2\Delta\zeta} \end{aligned} \tag{3.1}$$

where $f_\xi$, $f_\eta$, $f_\zeta$ are the $\xi$, $\eta$, $\zeta$ derivatives of a three dimensional function f, (i,j,k) is the point at which the derivative is evaluated and $\Delta\xi$, $\Delta\eta$, $\Delta\zeta$ are the grid spacings along each of those directions. The double derivatives are approximated as,

$$\begin{aligned} f_{\xi\xi}|_{i,j,k} &= \frac{f_{i+1,j,k} - 2f_{i,j,k} + f_{i-1,j,k}}{\Delta\xi^2} \\ f_{\eta\eta}|_{i,j,k} &= \frac{f_{i,j+1,k} - 2f_{i,j,k} + f_{i,j-1,k}}{\Delta\eta^2} \\ f_{\zeta\zeta}|_{i,j,k} &= \frac{f_{i,j,k+1} - 2f_{i,j,k} + f_{i,j,k-1}}{\Delta\zeta^2} \end{aligned} \tag{3.2}$$

and the cross derivatives are approximated as follows.

$$\begin{aligned} f_{\xi\eta}|_{i,j,k} &= \frac{f_{i+1,j+1,k} - f_{i+1,j-1,k} - f_{i-1,j+1,k} + f_{i-1,j-1,k}}{4\Delta\xi\Delta\eta} \\ f_{\eta\zeta}|_{i,j,k} &= \frac{f_{i,j+1,k+1} - f_{i,j-1,k+1} - f_{i,j+1,k-1} + f_{i,j-1,k-1}}{4\Delta\eta\Delta\zeta} \\ f_{\zeta\xi}|_{i,j,k} &= \frac{f_{i+1,j,k+1} - f_{i-1,j,k+1} - f_{i+1,j,k-1} + f_{i-1,j,k-1}}{4\Delta\zeta\Delta\xi} \end{aligned} \tag{3.3}$$

Whenever the second derivatives contain a variable coefficient for the first derivative a local average is used to evaluate the coefficient while the above formulation is used to determine the second derivative of the dependent variable at that grid point. Since the partial differential equation also involves a time derivative all the second space derivatives used in the difference model, are averaged over time according to the **Crank-Nicolson** scheme. This

completes the finite difference development for the differential equations at all the interior points.

The boundary conditions of the differential problem needs to be transformed to its finite difference approximation. These conditions may or may not involve derivatives depending upon, whether they are Dirichlet or mixed type conditions. The finite difference formulations for these two conditions are different depending upon the purpose of this evaluation. In other words, if the boundary condition itself does not involve derivatives but the derivatives on the boundary are required for other evaluations, then a simple one sided second order accurate difference is used. The boundary point formulation in such a case for the present problem is as follows.

$$f_\xi = \frac{3f_{n,j,k} - 4f_{n-1,j,k} + f_{n-2,j,k}}{2\Delta\xi};\ \text{i = n boundary} \tag{3.4}$$

$$f_\xi = \frac{-3f_{0,j,k} + 4f_{1,j,k} - f_{2,j,k}}{2\Delta\xi};\ \text{i = 0 boundary} \tag{3.5}$$

Similar equations can be written for the other two derivatives on the boundary. In the case of mixed boundary conditions the boundary conditions themselves involve first derivatives to be evaluated on the boundary. In such cases a fictitious surface of points surrounding the boundary surface on the outside is assumed. The first derivatives to be evaluated at the boundary points are then approximated by normal central differences involving the single interior point neighbor and the fictitious point on the surface outside the boundary, perpendicular to the direction of evaluation as in Equation 3.1. The boundary conditions can then be manipulated to get the value of the dependent variable at the fictitious point as a function of the variable values on the boundary and interior points. Now, the differential equations are written for the boundary point in terms of their finite difference formulations. The approximations used above for the second derivatives would now involve function evaluation at points outside the boundary, which are fictitious. The values of the dependent variable

at these points have already been established as functions of variable values at the interior and boundary points, and these functionalities are appropriately used to eliminate the fictitious point evaluations. Thus, the final set of boundary difference equations obtained in this manner can be added to the existing set of interior point difference equations to solve for the dependent variable values over the entire grid. For, a three dimensional problem involving various cross derivatives, this process of elimination of fictitious points by substitution is a very tedious and time consuming process with lots of opportunities for mistakes, when done by hand. But the basis of any variable elimination process is a simple concept of coefficient collection. A brief description of how this could be achieved in the coding without cumbersome reformulation is given in the Simulation chapter of the Thesis.

## 3.2 The Multi-Grid Method

### 3.2.1 Introduction

Solution to a differential equation, involves a number of steps each of which derives from different areas of mathematics. But essentially all differential equations when solved numerically are reduced at some stage to an algebraic approximation either by using finite difference or variational formulations. This reduces the differential equations and the boundary conditions, into a set of simultaneous linear or nonlinear algebraic equations. The classic techniques that are used to solve these problems involve matrix reduction or inversion, or iterative solution by repeated correction of an initial guess in the case of nonlinear equations. Techniques such as multidimensional Newton-Raphson methods are also popular, but are normally used in a modified form to overcome computational burden involving calculation of derivatives and huge matrix inversions. All the above methods belong to a class of operations called **Relaxations**. All the methods start with an initial solution or guess and relax the guess over multiple passes or iterations over the given set of equations. This can

also be seen as relaxation of an initial error over different iterations. Figure 3.2 shows the

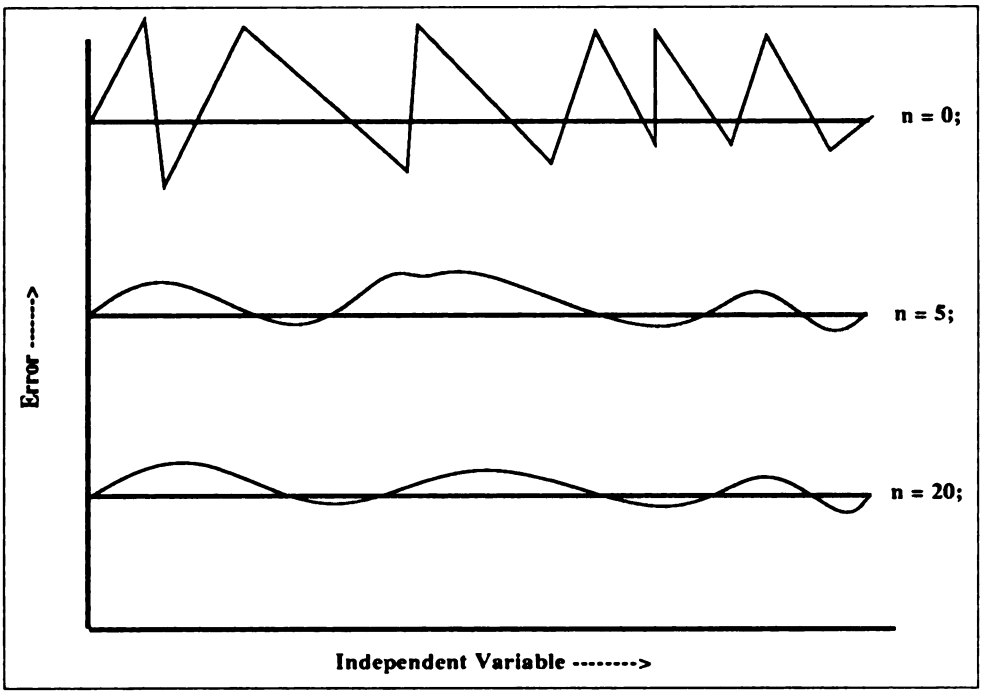



Figure 3.2. **Performance of relaxation methods**

performance of a typical relaxation method as a function of the number of iterations employed. The figure indicates two essential features. The first one is that, initially the error is highly oscillatory in nature when seen as a function of the independent variable. But as the number of iterations are increased, it becomes smoother and the average amplitude of the oscillations is reduced. Secondly, after a certain number of iterations further relaxations produce negligible change in reducing the remaining components of the error. The explanation for this kind of performance is that, the relaxation methods operate efficiently on the highly oscillatory components of the error to reduce them rapidly, until only the smooth components are left. Their performance degrades when they operate on the smoother com-

ponents of the error. Different methods have been adopted in the past that reduce this effect of relaxations by modifications to the actual relaxation algorithm. These include using iterative parameters which act as functions of the number of relaxations and approximating the solution as weighted sums of solutions over successive relaxations. Multi-grid methods are conceptually based on the reduction of the smooth components of the error by amplifying their oscillatory nature. Briggs [4], Hackbusch [5], McCormick([6] and [7]) contain different parts of the details for the development and implementation of multi-grid methods for different problems. These references develop the methodology for one and two dimensional problems. The extension to three dimensions is fairly straightforward, except some coding detail which is dealt with, in the next chapter. Figure 3.3 shows the effect of reducing

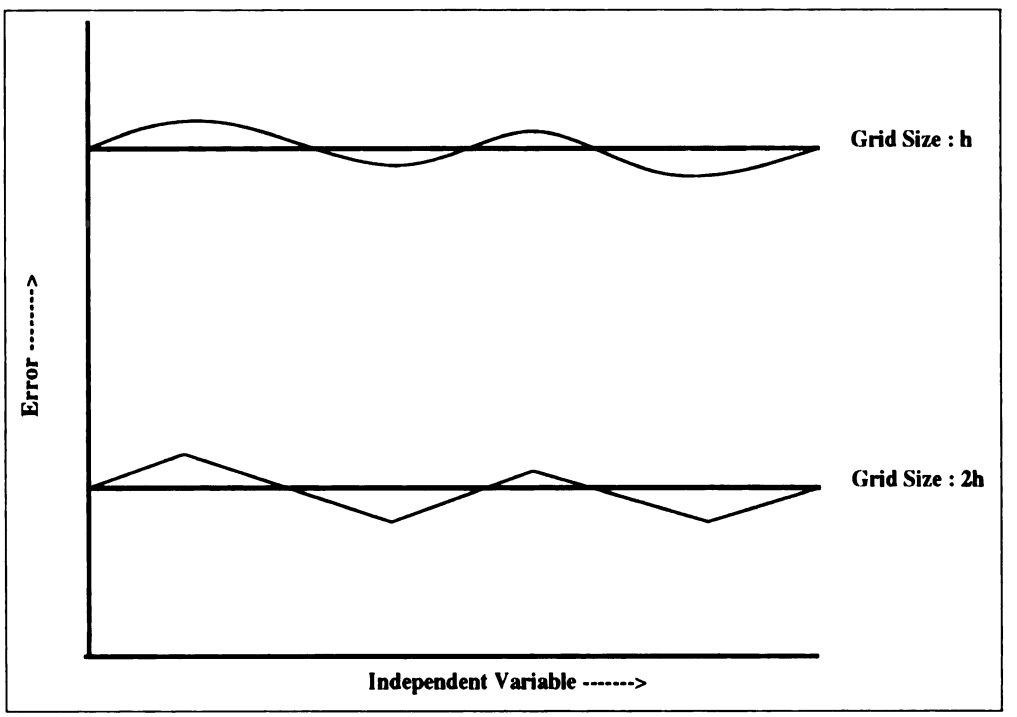



Figure 3.3. **Effect of relaxing over different grid sizes**

the number of grid points, in the solution to a set of difference equations. This reduces the number of equations to be solved, for an approximate solution of the original problem. But more importantly, the representation shows clearly that the error which appeared smooth on the finer grid (containing more grid points) is now more oscillatory or contains oscillatory components in the coarser grid (containing lesser number of grid points). In effect, we have amplified the oscillations of the smooth components of the error by using a two grid representation for the original problem. Hence, any relaxation method initially used to relax the error on the fine grid could be used to do the same on the coarser grid. Multi-grid methods do this by representing the residue or the difference between the exact and approximate solutions on the coarser grid and use the coarse grid relaxation to correct the smoothened solution on the fine grid.

Let the finite difference representation of a differential equation do so on a grid of size 'h'. This is the finest grid over which the problem is being solved. The problem including the boundary conditions can be represented in the operator notation as,

$$L_h\left(u_h\right) = f_h \tag{3.6}$$

where the subscript h refers to the grid size. The operator L can be linear or nonlinear in the dependent variable vector u. The term $f_h$ is the finite difference approximation to the source terms or forcing terms (terms which are not functions of the dependent variable). A relaxation of the above set of equations attempts to provide successive approximations to the variables $u_h$. Since the solution is only an approximation there is always a residual error which is given as,

$$\epsilon_h = f_h - L_h\left(u_h\right)$$

An approximation to this residual is obtained by relaxing the residual problem on a coarse grid say 2h (larger grid size and hence lesser number of grid points), and using that solution

to correct the solution to the problem on grid h. The multi-grid algorithm is nothing, but an extension of the two grid algorithm to the coarsest grid. That is, the multi-grid algorithm successively solves the residual problem on coarser and coarser grids until the problem is relaxed on a grid containing only a single interior point to a good enough accuracy. But there still needs to be established a way to go from the solution on one grid to that of another grid. The operators used for this purpose are discussed in the following section.

## 3.3 Operators

The multi-grid method works with three different kinds of generic operators, other than the operators specific to the problem being solved. Two of these operators are used to obtain coarse grid solutions from the dependent variable values on a fine grid and vice-versa. The operator which calculates the coarse grid solution from the fine grid solution, is called the **Restriction** operator. The other operator which performs the reverse of the preceding operation is called the **Prolongation** operator (also called Interpolation in some texts). The third operator is the relaxation operator, which can be any one of a number of iterative methods available for difference equations. The exact construction adopted for any of these operators is in general dependent on the problem being solved i.e the order of the differential equation, the nature of non-linearities etc. The trans-grid operators i.e prolongation and restriction operators are also additionally dependent on the difference formulation used for the solution. Different operators are used depending upon whether the finite difference formulation is based on a vertex centered or cell centered approach. Figure 3.4 shows the two-dimensional coarsening and refining of the grid as well as the solution using the restriction and prolongation operators respectively. Hackbusch [5] provides a very detailed discussion on the different kinds of restriction and prolongation operators in two dimensions. The development to three dimensions is fairly straightforward. Since the problems being solved

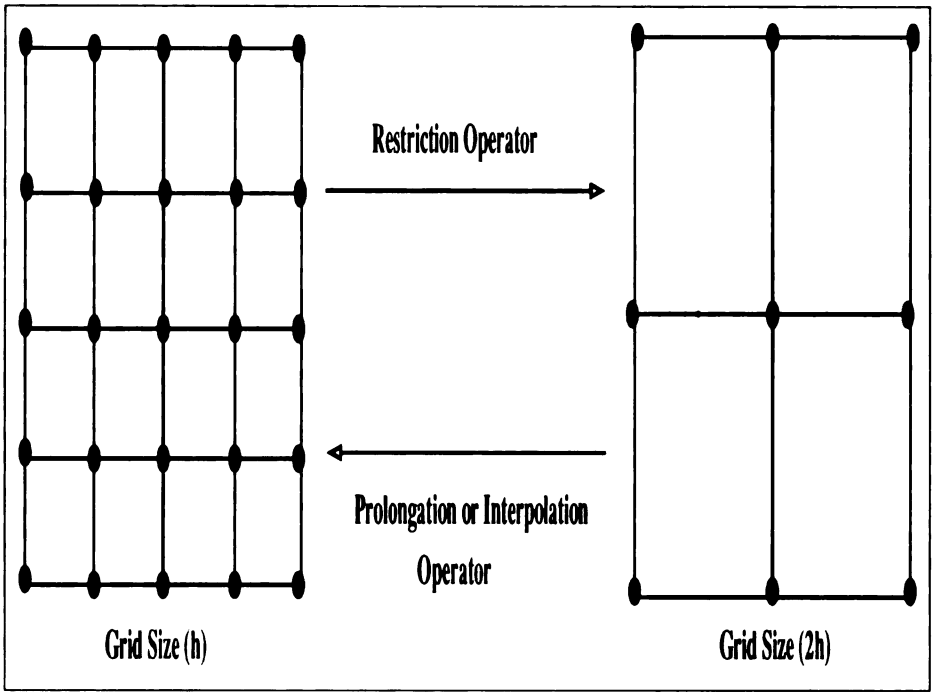



Figure 3.4. **Restriction and Prolongation Operators**

are all second order partial differential equations, the recommended prolongation operator is linear interpolation and the restriction operator is its inverse. For the problem at hand, the following prolongation operator was used.

$$
\begin{aligned}
u^h_{2i,2j,2k} &= u^{2h}_{i,j,k} \;\forall\; 0 \le i,j,k \le n,m,l \\
u^h_{2i+1,2j,2k} &= 0.50 * \left(u^{2h}_{i,j,k} + u^{2h}_{i+1,j,k}\right) \;\forall\; (0 < i < n) \\
u^h_{2i,2j+1,2k} &= 0.50 * \left(u^{2h}_{i,j,k} + u^{2h}_{i,j+1,k}\right) \;\forall\; (0 < j < m) \\
u^h_{2i,2j,2k+1} &= 0.50 \left(u^{2h}_{i,j,k} + u^{2h}_{i,j,k+1}\right);\; 0 < k < l \\
u^h_{2i,2j+1,2k+1} &= 0.25 \left(u^{2h}_{i,j,k} + u^{2h}_{i,j+1,k} + u^{2h}_{i,j,k+1} + u^h_{i,j+1,k+1}\right) \;\forall\; (0 < j,k < m,l) \\
u^h_{2i+1,2j,2k+1} &= 0.25 \left(u^{2h}_{i,j,k} + u^{2h}_{i+1,j,k} + u^{2h}_{i,j,k+1} + u^h_{i+1,j,k+1}\right) \;\forall\; (0 < i,k < n,l) \\
u^h_{2i+1,2j+1,2k} &= 0.25 \left(u^{2h}_{i,j,k} + u^{2h}_{i+1,j,k} + u^{2h}_{i,j+1,k} + u^h_{i+1,j+1,k}\right) \;\forall\; (0 < i,j < n,m)
\end{aligned}
$$

$$u^h_{2i+1,2j+1,2k+1} = 0.125\left(u^{2h}_{i,j,k} + u^{2h}_{i+1,j,k} + u^{2h}_{i,j+1,k} + u^h_{i,j,k+1} + u^{2h}_{i+1,j+1,k} + u^{2h}_{i+1,j,k+1}\right)$$
$$+0.125\left(u^{2h}_{i,j+1,k+1} + u^{2h}_{i+1,j+1,k+1}\right) \ \forall\,(0 < i,j,k < n,m,l) \qquad (3.7)$$

where i,j,k are the grid coordinates of any point in space, u is the dependent variable vector of discretized values, h and 2h are the fine and coarse grids used and n,m,l are the number of grid points used along the three principal coordinate directions. The corresponding restriction operator for the current formulation is given as follows.

$$\begin{aligned}
u^{2h}_{i,j,k} &= u^h_{2i,2j,2k};\ i = 0,n \ or\ j = 0,m \ or\ k = 0,l \\
f_1 &= u^h_{2i,2j,2k} \\
f_2 &= u^h_{2i-1,2j,2k} + u^h_{2i+1,2j,2k} + u^h_{2i,2j-1,2k} \\
&\quad + u^h_{2i,2j+1,2k} + u^h_{2i,2j,2k-1} + u^h_{2i,2j,2k+1} \\
f_3 &= u^h_{2i-1,2j-1,2k} + u^h_{2i-1,2j+1,2k} + u^h_{2i+1,2j-1,2k} \\
&\quad + u^h_{2i+1,2j+1,2k} + u^h_{2i-1,2j,2k-1} + u^h_{2i-1,2j,2k+1} \\
&\quad + u^h_{2i+1,2j,2k-1} + u^h_{2i+1,2j,2k+1} + u^h_{2i,2j-1,2k-1} \\
&\quad + u^h_{2i,2j-1,2k+1} + u^h_{2i,2j+1,2k-1} + u^h_{2i,2j+1,2k+1} \\
f_4 &= u^h_{2i-1,2j-1,2k-1} + u^h_{2i-1,2j-1,2k+1} + u^h_{2i-1,2j+1,2k-1} \\
&\quad + u^h_{2i-1,2j+1,2k+1} + u^h_{2i+1,2j-1,2k-1} + u^h_{2i+1,2j-1,2k+1} \\
&\quad + u^h_{2i+1,2j+1,2k-1} + u^h_{2i+1,2j+1,2k+1}
\end{aligned} \qquad (3.8)$$

$$u^{2h}_{i,j,k} = (8f_1 + 4f_2 + 2f_3 + f_4)/64 \qquad (3.9)$$

Finally, the choice of a relaxation operator has to be made before providing the formulation for the multi-grid method for a set of nonlinear equations. Any standard matrix relaxation algorithm is a candidate. A number of methods exist including the Jacobi iterations, the alternating direction implicit method and the Gauss-Siedel method. All these methods have

their distinct features and advantages . But, in conjunction with the multi-grid algorithm it has already been noted that the number of relaxations that are performed on a particular grid is small (about five or six). This implies that the solution algorithm is not significantly dependent on the relaxation procedure for its accuracy or computational efficiency. The stability of the multi-grid method is a characteristic that is dependent on the choice of the relaxation, but all the ones mentioned above are stable for similar kind of problems and hence this is also not a constraining factor on the choice of the relaxation method. The Gauss-Siedel iterative scheme is one of the most popular schemes used as a relaxation operator and the same has been used in the multi-grid implementation for this simulation. Additionally this method can be combined with a local Newton-Raphson(NR) operator to handle highly non-linear equations ([8]). The NR method has not been specifically implemented for the problems solved in this effort, but a tested routine has been included to do this if the need arises at a later date. The Gauss-Siedel method in its implementation provides a number of choices depending on the ordering of the points relaxed. In other words each of these methods relax over a different ordering of the points at each pass. A Red-Black(odd-even) ordering of points was chosen in the implementation. This essentially means, that the odd numbered grid points along each direction are relaxed first followed by the even numbered grid points. This method provides a very simple reduction to the algorithm when applied for a linearized (all coefficients are locally linearized) coefficient finite difference formulation. Consider the second order accurate representation that was proposed in Section 3.1. When derivatives are approximated using that formulation, it is clear that at any point the difference equation is entirely in terms of the nearest neighbors assuming all the coefficients and non-linear terms have been linearized in terms of the nearest neighbors. Thus at any point the dependent variable value at that grid point can be obtained as a function of its nearest neighbors. This implies that all the dependent variables at odd numbered grid points depend on the variable values at even numbered grid points and vice-versa. Hence a complete re-

laxation sweep could be performed only over one set of points at a time. This completes the definition and the choice of the different operators which form the backbone of the multi-grid algorithm. Thus, a formal definition could now be made for this algorithm and the variation of the method that has been used to solve the current problem at this stage. The exact methodology is established in the following section.

## 3.4 The Multi-Grid Algorithm

In the previous section the conceptual basis for the multi-grid method was given and a qualitative structure was established. This section formalizes and details the algorithms that are used in the implementation. The development is standard and could be found in different references already mentioned in the previous section. Each of these texts provides its own structure and modularity to these algorithms. But the algorithm that is to be given has been derived from these various texts by piecing together different structures as required.

Multi-Grid algorithms are defined differently for two different types of problems, namely linear and nonlinear difference equations. The essential difference is in obtaining the forcing vector for the coarse grid from the approximate solution on the fine grid. Due to the non-linearities that are present in the formulated problem the nonlinear multi-grid algorithm has been implemented. It was mentioned in the previous section that the coarsest grid correction is obtained for a grid with a single interior point. In the case of Dirichlet boundary condition, this implies that the value of the dependent variable at that interior point could be solved for exactly if the equation is linear. If the equation is nonlinear the interior point solution on the coarsest grid can be obtained only approximately by successive iteration. Hence the algorithm which handles a completely nonlinear problem is called the **Fully Approximate Storage** (FAS) algorithm (Brandt [9]).

Consider the difference equation system of the form given in Equation 3.6 including

the discretized boundary conditions. The multi-grid method begins with an initial guess or approximation for the dependent variable on the finest grid. The corresponding error in this approximation on the finest grid is given by Equation 3.2.1. A few sweeps of an appropriate relaxation scheme such as the Gauss-Siedel iteration would reduce the error to the smooth components. This relaxation operator is denoted by $S_h^{(\nu)}$(u,f), where $\nu$ refers to the number of sweeps that are performed on the given grid h. Now consider the immediate coarser grid of size 2h. The above error could be approximated on this grid by using a restriction operator as defined above. The restriction operator which restricts a variable vector on grid size h to a grid of size 2h can be denoted as $R_h^{2h}$. This provides a first approximation to the error on the grid of size 2h. This can be written as

$$e_{2h} = R_h^{2h}(e_h) \tag{3.10}$$

Thus the forcing term on the coarser grid is given by the approximation

$$f_{2h} = L_{2h}(u_{2h}) - e_{2h} \tag{3.11}$$

The operator $L_{2h}$ used above refers to the set of difference equations formulated on the grid of size 2h and not an operator obtained by restriction or other means. This implies that the problem on the coarse grid is now defined as that of finding a solution for the system of equations.

$$L_{2h}(u_{2h}) = f_{2h} \tag{3.12}$$

where $f_{2h}$ is as defined by Equation 3.11. There is only one thing that is to be defined and that is the initial guess for $u_{2h}$ and this given by a simple relaxation of the smoothed solution on grid h.

$$u_{2h}|_{initial} = R_h^{2h}(u_h) \tag{3.13}$$

Now the coarse grid approximation is smoothened by performing relaxations on the coarse grid to obtain the smoothened solution $u_{2h}$. The fine grid correction is then done as

$$u_h = u_h + I_{2h}^h \left( u_{2h} - R_h^{2h}(u_h) \right) \tag{3.14}$$

In the above equation $I_{2h}^h$ refers to the interpolation or prolongation operator that carries the solution from a coarse grid to the fine grid. The above development provides the basic formal structure for a two grid method or algorithm. The same could be extended by obtaining corrections on the grid size 2h by solving for a similarly constructed problem on a grid of size 4h and so on, till the coarsest grid containing a single grid point is reached. Here, one can iteratively solve for the value of the dependent variable as mentioned earlier. This is the essential multi-grid method. Figure 3.5 shows the pictorial representation of how this algorithm works, that is, one passes from a fine grid to a coarse grid smoothening the initial approximations by relaxation, till the coarsest grid is reached and moves from the coarsest to the finest grid correcting the finer grid solutions along the way. This is called a Fully Multi-Grid V (FMV) algorithm indicative of the V shape pass involved. The FMV algorithm is formally defined as n Table 3.1. The FAS algorithm which was mentioned prior to this development makes use of the FMV cycle at each grid. This means that, both pre-smoothing and correction using the coarser grids are performed on each of the grids, before moving on to the next grid. The graphical representation of this scheme is given in Figure 3.5. It is also called the F-cycle algorithm. In the FAS algorithm one can also control the number of FMV cycles to be performed and hence change the nature of the FAS cycle from the one shown in Figure 3.5. In a general sense, any multi-grid operation is an offshoot or variation of the methods explained in this section, but they operate on different kinds of variables depending upon the nature of the problem being solved and the convenience of representation as dictated by its complexity. Besides, use of the right kind of representation distributes the

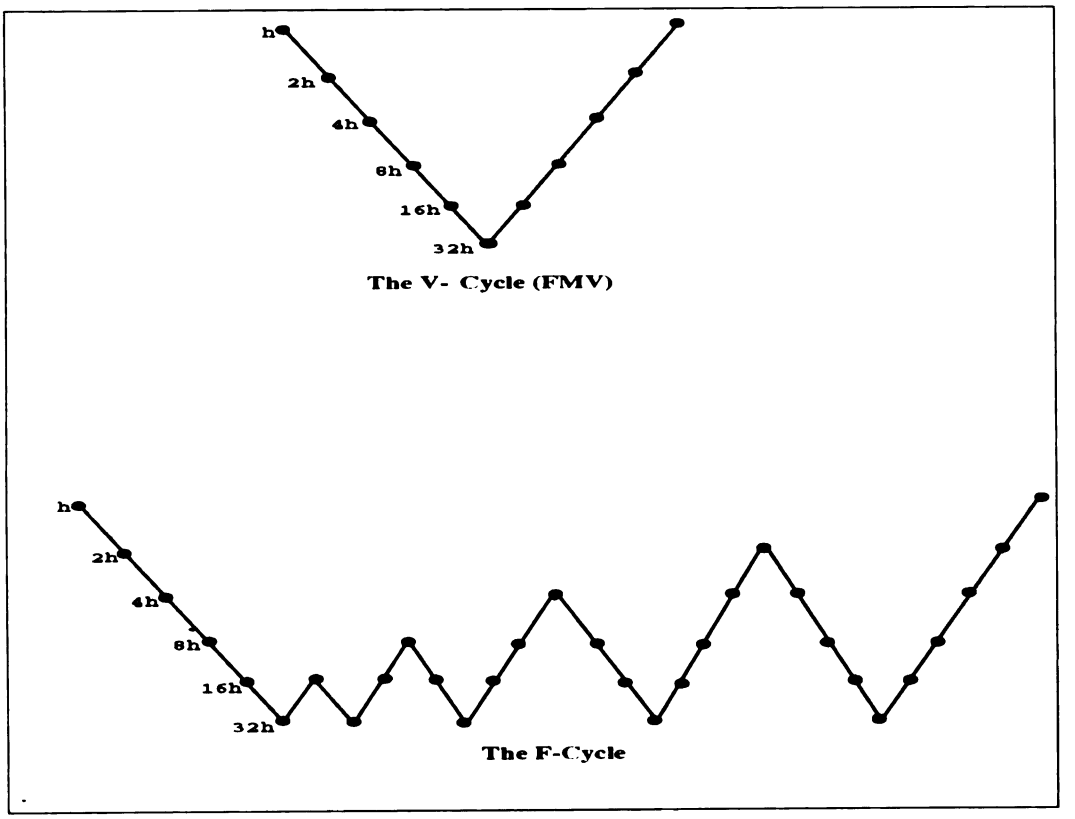



Figure 3.5. **The V and F cycles for multi-grid methods**

Table 3.1. **The FMV algorithm**

**FMV**$(h,u_h,f_h)$
if h is coarsest then $u_h=S_h^{\nu_0}(u_h,f_h)$
**else**
**begin**
$\hat{u}_h=S_h^{\nu}(u_h,f_h)$
$v_{2h}=R_h^{2h}(u_h)$
$f_{2h}=L_{2h}(u_h)-R_h^{2h}(L_h(u_h)-f_h)$
**FMV**$(2h,u_{2h},f_{2h})$
$u_h=u_h+I_{2h}^h(v_{2h}-\hat{u}_h)$
**end**

memory efficiently during simulation and helps to achieve a high degree of computational speed. The next section deals with the different kinds of variable representations used in the simulation and the manipulations that are performed on them to achieve the above goals.

# CHAPTER 4

# Simulation

## 4.1 Programming Fundamentals

The entire code for the simulation has been written in ANSI C with the supporting programs for output graphical generation using MATLAB(R) script programs, which are essentially programs with MATLAB(R) based commands. The development in the previous two chapters indicate that, at the outset, any code for the simulation should contain two relatively independent parts, namely

1. Mesh generation

2. Variable profile simulation (Temperature, Cure etc)

Though conceptually, the two parts are different in the sense of accomplishing different aspects of the simulation, both of them involve solving for differential equations of the same kind. The basic difference is that, the mesh generation part aims to solve for a time invariant set of partial differential equation and the profile simulation part involves solving for time dependent set of partial differential equations. The domains of definition for the differential equations are the same but the nature of boundary conditions may differ. For example, the mesh generation part involves solving Dirichlet boundary conditions as explained in

Chapter 2, while the profile simulation can involve solution to either Dirichlet or mixed type boundary conditions. Hence, the basic set of routines that have to be constructed to solve the difference equations in both these apparently different parts perform the same functions using different operators and subject to different boundary conditions.

Thus, any code written for this purpose, should have the necessary set of generality in its basic set of routines to allow for usage for the two parts as mentioned above. The mesh generation part consists of the following elements

1. Surface approximation

2. Initial volume generation using algebraic methods.

3. Mesh refinement using elliptic generation

The surface approximation involves the generation of a surface mesh using algebraic methods, using the edge and or surface coordinate information provided by the user. The elliptic generation method was explained in detail in Chapter 2 . The finite difference representation of the various different terms involved was also presented there. The multi-grid method is used to solve the finite difference representations of the differential equations in both the above parts. This critically determines the various data structures for the representations to be used for the variables to be used in the code. The next section discusses in detail the basic structures used and their relevance to the understanding and implementation of the various methods discussed so far. The Appendices contain the different routines with the notes on the variables used and the details of the exact implementation of the different algorithms.

### 4.1.1 Data Structures

The basic variables to be solved for, in the simulation are the temperature and cure. Besides, the mesh generation step involves determining the physical coordinates (i.e Cartesian) as a

function of the natural coordinates. All these variables are hence functions of the natural coordinates, and hence three dimensional in nature at any point of time. The time dependency of the temperature and cure variables is taken into account by replacing the current set of variables, for their values at the previous time step but always retaining values one time step old. This is required, because the finite difference representation for the second derivatives in the process differential equations are done using the Crank-Nicholson technique which involves values of the variables at the previous time step. Hence, the basic data structure unit of this code is a three dimensional matrix represented as U(i,j,k) where U is any three dimensional variable under consideration and i,j,k is the grid position in the natural coordinate system.

In addition, as explained in Chapter 3 the multi-grid algorithm also imposes some new representation constraints on the code. Besides being three dimensional in nature, the variables to be solved for are also functions of the grid size. The multi-grid algorithm as defined earlier requires that all the variables be calculated not only on the finest grid but also on each of the grids up to the coarsest grid containing just one interior point and all the rest boundary points. The logical solution to this problem is to use a series or a linked list of structures, each of the structures corresponding to a particular grid size and containing information about the grid parameters, the values of the required variables as a function of three dimensional space within that particular grid size. The structures are linked in the direction of increasing grid size, to facilitate recursive computation of coarser grid variables, in terms of the values of the same variables on the finer grid, initially before relaxation is performed on the grid. Figure 4.1 shows the pictorial representation of a linked-list structure used for the purpose of multi-grid solution of the finite difference equations. Each of the blocks in the figure could be one of the two array structures that are described below. There are two of these structures that are used, one for the mesh generation part, and one for the profile simulation part. The mesh generation code uses the structure in Table 4.1. In

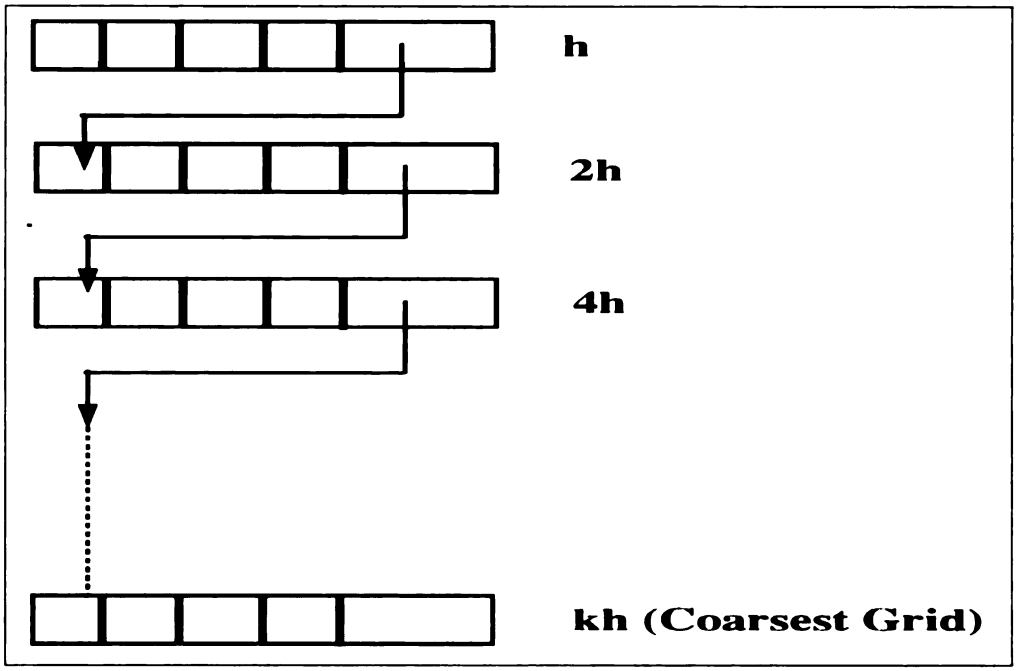



Figure 4.1. **Linked-list structure for Multi-grid method**

the above structure '***u' is the C notation, for a three dimensional variable. The source terms are calculated for the coarse grid from the variable values and the source term values for the fine grid as explained in Chapter 3.

Once the mesh generation is complete, it provides information on the functional dependency of the physical coordinates on the natural coordinates. In addition it also provides the local Jacobian values at each grid point as well as the shape transformation matrices i.e the $\alpha_{kl}$ at every grid point. Hence a new structure is defined which serves as the conduit passing information between the mesh generation part and the profile simulation part. The **point** structure accomplishes this objective. In the C language notation this is defined as in Table 4.2. In the above definition, the variable definition **D2_array** defines the two dimensional matrix $\alpha$ at each grid point which contains the shape factor information.

The profile simulation part performs the multi-grid algorithm at each time step from an

Table 4.1. **Data structure for mesh generation**

```
structure varray_xyz
{
int n,m,l; /* Number of grid points along the 3 directions */
double ***x,***y,***z; /* The physical coordinates
                          as three dimensional matrices */
double ***Sx,***Sy,***Sz; /* Source terms in the
                             difference equations, as three
                             dimensional matrices */
struct varray_xyz *next; /* Pointer to the coarser grid */
}
```

Table 4.2. **Conduit structure between mesh generation and profile simulation**

```
structure point
{
double ***x,***y,***z; /* Physical coordinate
                          information at each grid point */
struct D2_array ***a; /* Shape factor information at
                         each grid point */
double ***J; /* Local Jacobian evaluated
                at each grid point */
}
```

initial to a final point, but it uses a slightly different structure. Essentially, the shape information available from mesh generation via the **point** structure has to be made available for each of the different grids from the finest to the coarsest. Besides, the profile simulation part involves solving for difference equations in its relaxation part that use time averaged operators and hence information on the previous time values of the variables should also be available. In addition, the process conditions such as heat transfer coefficients have to be made available at each surface point and for each grid size. All these constraints motivates the use of the structure defined as in Table 4.3 for the profile simulation part. The structures

Table 4.3. **Data structure for profile simulation**

```
structure varray
{
int n,m,l; /* Number of grid points */
struct point *grid_pt; /* Shape information */
double ***multp; /* Time derivative coefficients */
double ***Sf; /* Source term for difference equation */
double ***T,***X; /*Temperature and Cure
                    variables at current time */
double ***Tg,***Xg; /* Temperature and Cure
                      variables at previous time step */
double ***conv; /* Heat transfer coefficient */
struct varray *next; /* Pointer to next level
                      coarser grid */
}
```

seem to carry a lot of computational burden but actually only the temperature and the cure information are updated during every relaxation sweep. All the other information are either constant throughout an entire step or through all the relaxation passes for a given grid. Besides the memory allocation is done dynamically that is the memory is allocated only when required and freed when a particular structure is not required anymore. It was stated earlier that the number of relaxation sweeps on a given grid is very small and this greatly reduces the computational time over the traditional single grid methods.

## 4.2 Graphical Output Interface

Mesh generation provides a model of the original domain on which the equations are to be solved. The accuracy of this representation depends on a number of factors including the nature of the shape and the user supplied surface coordinate information. Thus, the user has to be sure that there exists a one to one correspondence between the modelled or transformed

domain and the original shape. The boundary fitted coordinate system generated using the elliptic system of Laplacians ensures this, but the accuracy is not guaranteed. Hence, a visual interface has to be provided wherein the user can view the actual shape of the object being modelled, in correspondence to the transformed coordinates. The user should also be able to view different sections to decide if a more accurate surface representation is required for the shape at hand. In addition, when the simulation moves into actually generating the variable profiles there is a necessity to observe the development of these profiles both with respect to time and space. This provides the user with the problem to shape correspondence of the process in a qualitative sense in the least. The simulation though complete for certain purposes is in its developmental stages, from the point of view of the different processes being modelled, as well as the sophistication of the numerical grid generation procedures which can be enhanced. There needs to be a justifiable tradeoff between the nature of problems being solved, the computational efficiency of the grid generation procedure in terms of time and cost and the accuracy of the results required. Such information can be provided only based on extensive testing of the simulation for different problems and verification with experimental results. A graphical output interface achieves the goal of providing the user information of the magnitude and nature of variables at every stage of the simulation. Such an interface has been established using the MATLAB(R) external interface library which provides tools for simultaneous display of variables as the solution evolves. The representation that MATLAB(R) uses for its variables is different from the representation that is being used within the C code for the simulation, the nature of which are dictated by the methods being used and the demands of generality on the simulation. The next section discusses some details regarding the representational differences and the coding that has been included to translate between the two.

### 4.2.1 The MATLAB External Library

MATLAB(R) is a mathematical, control and graphics tool which provides a lot of interactive tools that can act as interfaces between the user and a C or FORTRAN code that performs a complex numerical task. Besides, MATLAB(R) also provides for its own language which operates on these structural units. In addition, it has a vast compilation of routines which perform predefined functions which include manipulation of two and three dimensional plots as well as contour maps. Hence MATLAB(R) can perform tasks based on commands from external C code through its external library as well as operate using programs written in its own language. The most efficient use of the tools available from this software is made, when there exists a partition between the nature of the conduit created between the C code and the MATLAB(R) workspace, and the kind of tasks controlled by programs written in the MATLAB(R) language. This partition is the partition between information passing and information processing. In other words the best arrangement is when MATLAB(R) receives only the data passed to it from the code for display and all the structural manipulation and control of the graphical procedures such as plotting and sectioning occurs from the MATLAB(R) language based programs. In any simulation, the graphical interface is just a window through which the user is able to perceive at all times the status of the simulation in an easily understandable form. In a simulation of a complex nature, the interface should provide only the minimum demands on the computer time and memory. This has to be borne in mind when evolving criteria for implementation. To centralize the control the, C code was initially given total control over the MATLAB(R) workspace by explicitly specifying the different operations as well as performing different matrix manipulation operations. It was found that from a speed point of view, the partitioning of the tasks as information passing and processing, between the code and the MATLAB(R) script files is much faster than a centralized control of all operations . This is advised for all future implementations using this workspace.

Programs based on the MATLAB(R) language are called script files. These script files are functions which take in arguments which are allowed MATLAB(R) structures, and perform both numerical and graphical operations on them. The basic information containing unit of the MATLAB(R) language is a matrix (utmost two dimensional). All the variables that MATLAB(R) handles are matrices or vectors including string variables. On the other hand the basic structural unit for all the key variables used in the simulation code are three dimensional matrices. The passage from one representation to another is established from within the C code itself. That is, whenever numerical information needs to be passed to the MATLAB(R) workspace the C code manipulates the structure into the appropriate MATLAB(R) format (a three dimensional matrix is converted to a vector in this case). Once the information is passed all further operations to be performed on them is done using the MATLAB(R) script files.

Once the information is made available on the MATLAB(R) workspace, it needs to be manipulated effectively to obtain the relevant graphical output. Since originally three dimensional matrices are converted into vectors, simple script programs were written to provide independent access along the different coordinate directions as required . Once this is established, further manipulation of the operators is straightforward and are contained in additional script files written for this purpose. For a detailed discussion of the various commands involved and the exact implementation of the MATLAB(R) external library the appropriate manuals [10] & [11] are referred. The Appendices contain details of the different script files used and the tasks being performed by them. The next chapter contains different plots generated by the MATLAB interface . In addition, the interface is also capable of generating color maps indicate of temperature profiles and also allows the user who is conversant with MATLAB to manipulate shapes and figures.

# CHAPTER 5

# Results and Discussion

## 5.1 Shape Simulation

The very first step in the profile prediction is the simulation of the shape of the domain. The boundary fitted coordinate system was implemented using the multi-grid method for the finite difference solution. As mentioned in Chapter 2, the shape simulation consists of two parts, the algebraic grid generation and subsequent refinement using the elliptic system of equations. The algebraic generation is often very approximate and serves only as a first guess for the solution to the elliptic generation system. Thus the capability of the shape simulation is critically dependent on the elliptic generating system.

In the current effort this capability was tested by providing a bad initial guess deliberately over the entire domain with correct coordinate information being provided only on the boundaries. The elliptic generating system was made to take over from that point onwards. Figure 5.1 shows the results of such a shape generation procedure starting from a bad initial guess. The elliptic generation system performs very well as shown in the figure. It refines the solution at the interior points so that the interior is boundary conforming and also generates a one to one mapping between natural and physical coordinate systems in the process. Figures 5.2 and 5.3 show the section wise comparison between the surfaces of

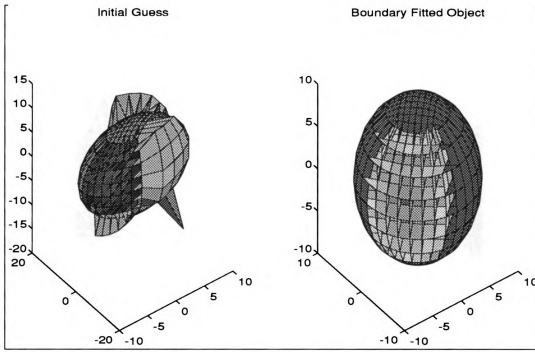



Figure 5.1. **Performance of BFCS method on bad initial guess**

the initial guess and the corresponding surfaces on the boundary fitted object. In these figures u,v,w indicate the $\xi$ , $\eta$ and $\zeta$ coordinates respectively. These are plots generated by the user interface created using the MATLAB(R) external library. It is evident that the Laplacian system of generating equations provides a smooth solution to the shape simulation in terms of coordinate line continuity in the interior. However, the surface coordinates remain at their specified values. The results of the shape simulation include not only the coordinate values but also the shape factors ($\alpha_{ij}$s) and the Jacobian (J) values at each of the grid points in the natural coordinate system. These values are then used for solving the process model in the natural coordinate system.

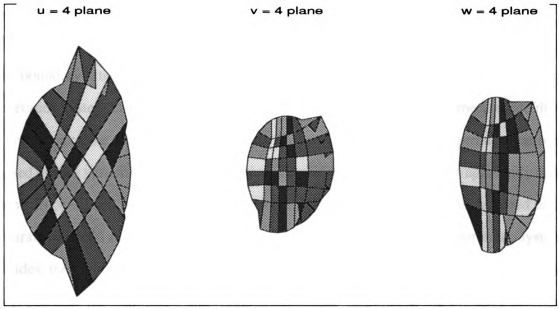



Figure 5.2. **Sections of initial guess**

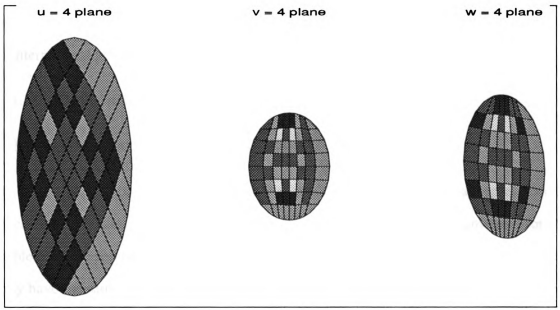



Figure 5.3. **Sections of boundary fitted object**

## 5.2 Profile Simulation

### 5.2.1 Analytical Verification

The boundary fitted coordinate system guarantees a one to one mapping under the elliptic generation system of Equation 2.13. But, the solution to this system is numerical and there is limitation on the accuracy of the problem. The original shape modelling problem has been solved using a system of non-linear elliptic system of equations and though apparently there seems to be a good solution to the shape modelling problem the shape factors need to be accurate to obtain a correct solution for the process model in the natural coordinate system. Besides, the boundary conditions of the transformed problem need to be solved accurately too. Consider for example, a uniform grid containing 10 points along the three principal directions in the natural coordinate system. The entire domain contains 1000 grid points out of which 488 are on the surface. Hence accurate solution to the boundary condition is essential for a good solution to the process model especially in the case of mixed boundary conditions. Hence a verification was performed using two simple cases for which there is no internal heat generation and for which analytical solutions are available. These are

1. A spherical domain with Dirichlet boundary conditions

2. A slab with mixed type boundary conditions.

The spherical domain problem is essentially an one dimensional problem as represented in spherical coordinates. But the three dimensional Cartesian coordinate counterpart for the problem can be solved to get the same solution because all the points on the surface of the body have the same temperature at all times and hence there is no conduction along the $\theta$ and $\phi$ directions of the spherical coordinate system. The problem being solved here is

$$\frac{\kappa}{r^2}\left(r^2\frac{\partial T}{\partial r}\right) = \rho c_p \frac{\partial T}{\partial t} \tag{5.1}$$

with the conditions

$$\begin{aligned} T &= T_i;\ 0 \leq r \leq R;\ t = 0 \\ T &= T_0;\ r = R;\ t > 0 \\ \frac{\partial T}{\partial r} &= 0;\ r = 0;\ t > 0 \end{aligned} \tag{5.2}$$

Of the above boundary conditions the symmetry boundary condition at r=0 cannot be represented easily in the transformed process model. But since this is only a symmetry condition it is inherently present in the shape simulation i.e all the shape factors are symmetric w.r.t the radial coordinate. The analytical solution to the above problem is

$$T(r,t) = T_i + (T_0 - T_i)\left(1 + \frac{2R}{\pi}\sum_{n=1}^{\infty}\frac{(-1)^n}{n} sin\left(\frac{n\pi r}{R}\right) exp\left(-\frac{n^2\pi^2\kappa t}{\rho c_p R^2}\right)\right) \tag{5.3}$$

Figure 5.4 shows the comparison of the analytical and simulation solutions for different radial positions within the material. It is evident that there is small error involved in the simulation. The maximum observed error is about 1.5% and there is absolutely no propagation of the error over time. The simulation was carried out at different radii and the error obtained in the other cases were about the same. The error is basically in the dynamics of the problem because over longer periods of time it dies down to zero as is evident from the figure. The numerical simulation was performed using 3 relaxation sweeps before and after correction and an F cycle algorithm as explained in Chapter 3. The next step in the verification is the case of a three dimensional slab with all dimensions comparable, and with convective boundary conditions. One end of the slab on each of the three directions is insulated while the other end is a convective boundary with an applicable heat transfer coefficient. For simplicity, it was assumed that all three dimensions were of the same magnitude and the heat transfer coefficient was the same on all the convective boundaries of the body. The problem

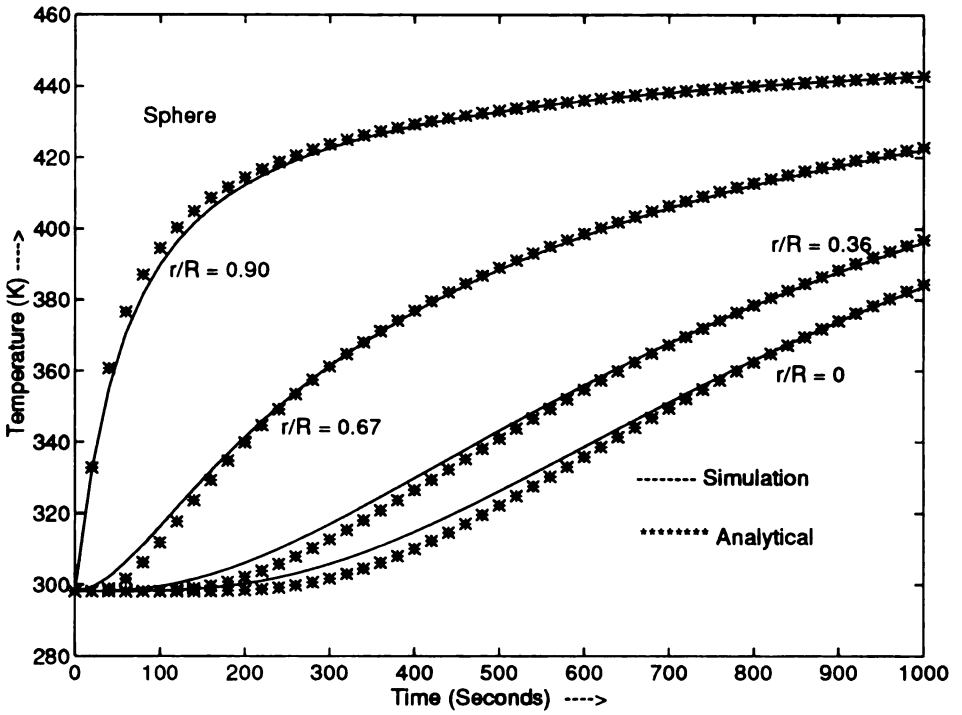



Figure 5.4. **Analytical verification for a spherical body with Dirichlet conditions**

in mathematical terms is stated as follows

$$\kappa\frac{\partial^2 T}{\partial x^2} + \kappa\frac{\partial^2 T}{\partial y^2} + \kappa\frac{\partial^2 T}{\partial z^2} = \rho c_p \frac{\partial T}{\partial t} \tag{5.4}$$

$$\begin{aligned}
\frac{\partial T}{\partial x} &= 0;\ x = 0;\ \forall\, y, z;\ t > 0 \\
\frac{\partial T}{\partial y} &= 0;\ y = 0;\ \forall\, x, z;\ t > 0 \\
\frac{\partial T}{\partial z} &= 0;\ z = 0;\ \forall\, x, y;\ t > 0
\end{aligned} \tag{5.5}$$

$$\begin{aligned}
-\kappa\frac{\partial T}{\partial x} &= h_c(T - T_0);\ ;x = L;\ \forall\, y, z;\ t > 0 \\
-\kappa\frac{\partial T}{\partial y} &= h_c(T - T_0);\ ;y = L;\ \forall\, x, z;\ t > 0
\end{aligned} \tag{5.6}$$

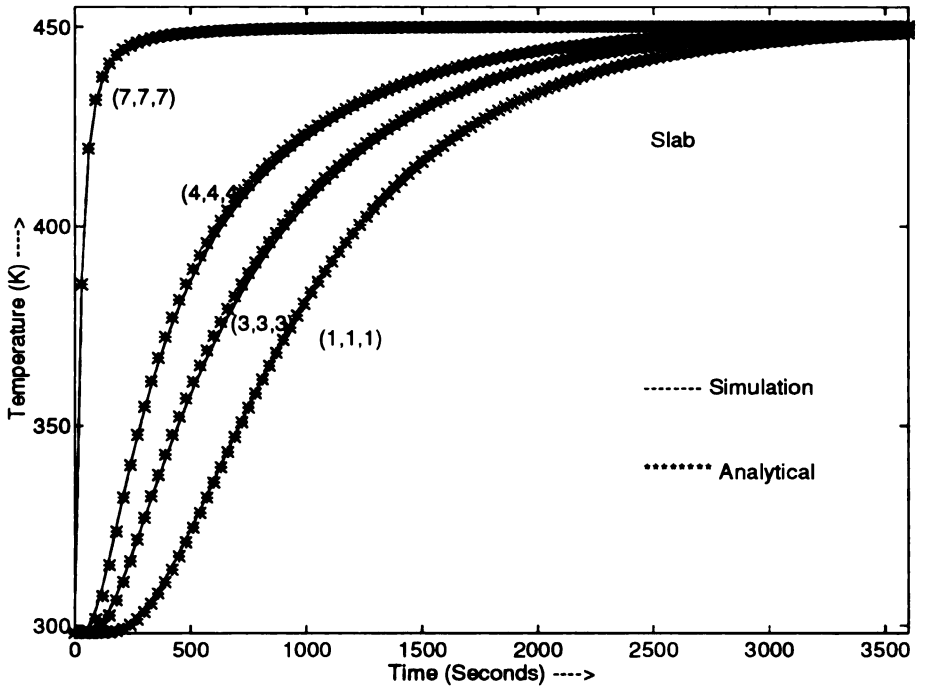



Figure 5.5. **Analytical verification for a slab with convective boundaries**

$$\begin{aligned} -\kappa\frac{\partial T}{\partial z} &= h_c(T - T_0);\ ; z = L;\ \forall\ x, y;\ t > 0 \\ T &= T_i;\ t = 0;\ \forall\ x, y, z \end{aligned} \tag{5.7}$$

The analytical solution for the above problem is not obvious but is constructed as the product of the solutions to three independent solutions of single dimensional problems under variable transformations detailed in Crank [12]. Under these transformations the solution to the above problem is given, as follows.

$$\begin{aligned} \Theta_x(x,t) &= \sum_{n=1}^{\infty} \frac{2\alpha cos\left(\frac{\beta_n x}{L}\right)}{(\beta_n^2 + \alpha^2 + \alpha)\, cos(\beta_n)} exp\left(-\frac{\beta_n^2 \kappa L}{\rho c_p t}\right) \\ \Theta_y(y,t) &= \sum_{m=1}^{\infty} \frac{2\alpha cos\left(\frac{\beta_n y}{L}\right)}{(\beta_m^2 + \alpha^2 + \alpha)\, cos(\beta_m)} exp\left(-\frac{\beta_m^2 \kappa L}{\rho c_p t}\right) \end{aligned}$$

$$\begin{aligned} \Theta_z(z,t) &= \sum_{k=1}^{\infty} \frac{2\alpha cos\left(\frac{\beta_k z}{L}\right)}{(\beta_k^2 + \alpha^2 + \alpha)\, cos(\beta_l)} exp\left(-\frac{\beta_k^2 \kappa L}{\rho c_p t}\right) \\ T(x,y,z,t) &= T_0 + (T_i - T_0)\Theta_x(x,t)\Theta_y(y,t)\Theta_z(z,t) \end{aligned} \tag{5.8}$$

where $\beta_n$ is the $n^{th}$ root of the equation

$$\beta_n tan(\beta_n) = \alpha = \frac{h_c L}{k} \tag{5.9}$$

The results of the simulation and the analytical solution obtained as above, are shown in Figure 5.5. Once again it is found that the simulation results are accurate and the maximum error is about 2.0%. This indicates that the nature of the error could be due to truncation as it seems to be independent of the nature of the problem. Smaller grid sizes are the solution to this problem but increasing the grid points in all the three directions is computationally inefficient.

### 5.2.2 Thermal Cure Simulation

The last section verified the results of the process model using certain simple cases. Both these cases do not involve any nonlinearities and they were basically heating experiments from the point of view of a chemical process. The advantage of a simulation package is that effects of different process parameter variations can be studied. Since this simulation is only in the developmental stages with regards to inclusion of different sub-models, it is necessary to at least assess the qualitative correctness of the results for a generalized case of thermal cure. A case study to this effect was performed on a general thermal cure process for an epoxy/graphite composite using the material property and cure cycle data from literature (Bogetti& Gillespie [1]). The data are given in Table 5.1. In the table the reaction

Table 5.1. **Data For Simulation Runs**

| Parameter | Value |
|---|---|
| $\rho$ | 1.52 g/cm$^3$ |
| $c_p$ | 0.942 kJ/(gm K) |
| $\kappa$ | 4.457 x $10^{-3}$ kW/(cm K) |
| h | 0.0022 kW/(cm$^2$ K) |
| $A_1$ | 35.03 x $10^6$/sec |
| $A_2$ | -33.5667 x $10^6$/sec |
| $A_3$ | 3266.67/sec |
| $\Delta E_1$ | 8.07 x $10^4$ J/mol |
| $\Delta E_2$ | 7.78 x $10^4$ J/mol |
| $\Delta E_3$ | 5.66 x $10^4$ J/mol |
| $\Delta H$ | -198.90 kJ/g |

rate parameters are for the following rate equation.

$$\begin{aligned} \frac{d\alpha}{dt} &= (k_1 + k_2\alpha)(1-\alpha)(0.47-\alpha);\ (\alpha \leq 0.30) \\ \frac{d\alpha}{dt} &= k_3(1-\alpha);\ (\alpha > 0.30) \\ k_i &= A_i exp(-\Delta E_i/RT);\ (i = 1,2,3) \end{aligned} \tag{5.10}$$

The shape on which the simulations were performed is shown in the Figure 5.6(an inverted V). The base case values of the distance between the top and bottom V-shaped surfaces and the length along the $\xi$-coordinate direction in Figure 5.6 were, 2.54cm and 7.62cm respectively. In the base case the top and bottom V surfaces are convective boundaries while all the other surfaces are assumed to be insulated. This reduces the problem to single dimensional one in natural coordinates. However it is to be noted that the dimensionality is reduced to one, only from the point of view of natural coordinates but not so in a Cartesian sense. Various parameters were then varied from their base case values shown in the Table 5.1, to observe the changes in the temperature and cure behavior at select positions within the ma-

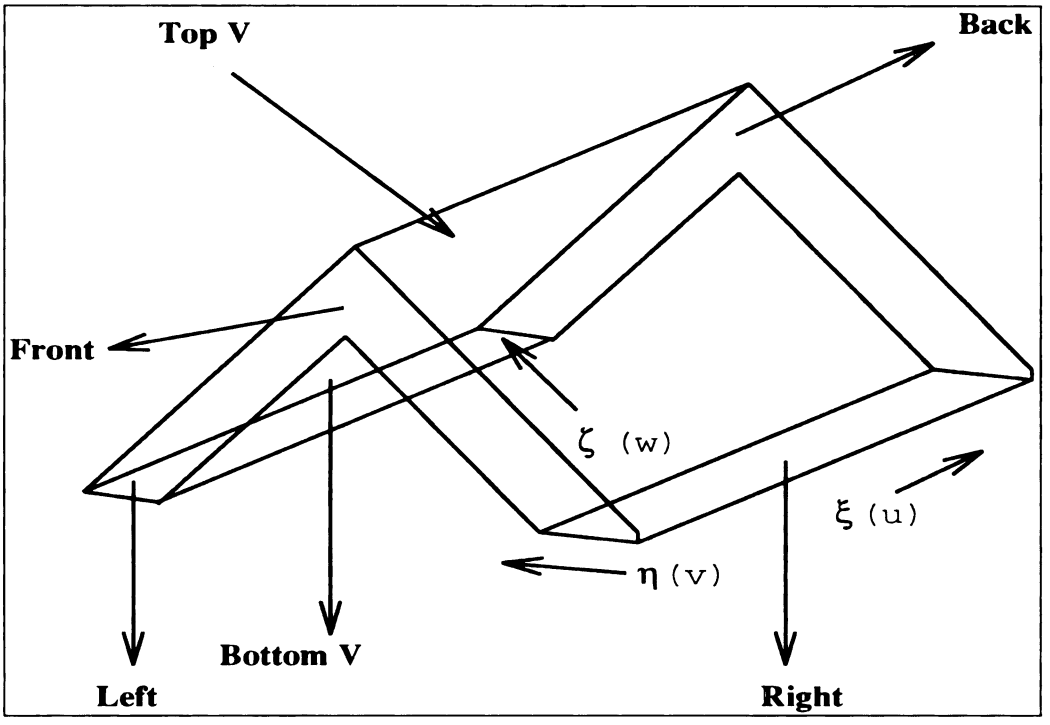



Figure 5.6. **Geometry for simulation runs**

terial. In the ensuing discussion temperature and cure profiles imply the time behavior of the two variables.

### Effect of Boundary Conditions

The boundary conditions play a very important role in the nature of temperature and cure gradients observed within the material. The effective heat transfer coefficient given by $(h_c/k)_{eff}$ is the critical parameter which determines the nature of the boundary conditions. Three different runs were performed each using a different value of $(h_c/k)_{eff}$ . The results are shown as temperature and cure profiles w.r.t time in Figure 5.7 and Figure 5.8. The values z/Z(= 0, 0.76) shown in the figure are the points on the central $\xi\zeta$ plane of the object, at

$\eta$=0 and $\eta$=4 natural coordinate values. Several points are to be noted here. An increase in $(h_c/k)_{eff}$ leads to decrease in the exotherm within the material. This is expected because, a larger heat transfer at the boundary convects heat away more efficiently compared to a smaller value of the same. However, the rise to the maximum temperature in this case is faster because a high value of the heat transfer coefficient also allows faster convection of heat to the material. For very large values of $(h_c/k)_{eff}$ the boundary conditions become of the Dirichlet type and there is no difference between the ambient and surface temperature of the material. The simulation results are in concurrence with these observations. In a three dimensional simulation especially one involving complex shapes boundary effects are not restricted to a single direction. Different boundary conditions create gradients along different directions which interact in a complex manner to affect the temperature and cure profiles of the material. To observe this effect, an additional run was performed, with the front and back surfaces of the V shape shown in Figure 5.6, being subjected to convective boundary conditions with the same heat transfer coefficient as in the base case. The results shown in Figure 5.9 indicate that the exotherms are now smaller but are attained quicker than in the base case. This is to be expected, as the addition of a convective boundary plays the same role as an increase in the heat transfer coefficient on a given boundary. However, the exact nature and magnitude of these exotherms is critically dependent on the relative dimensions of the material along these principal natural coordinate directions. To better understand the complexity involved, the simulation was performed with unsymmetrical boundary conditions along the different natural coordinate directions. The zero boundary along each coordinate line was assumed to be insulated while the other boundary was convective with a $(h_c/k)_{eff}$ equal to that of the base case value. The results of this simulation are shown in Figures 5.11 and 5.12. The (u,v,w) directions listed in the figures are the same as $(\xi, \eta, \zeta)$ of the natural coordinate system. The point (4,2,4) is the central point inside the body. Consider Figure 5.11 which shows the temperature profiles along $\eta$ coordinate lines on the central $\eta\zeta$

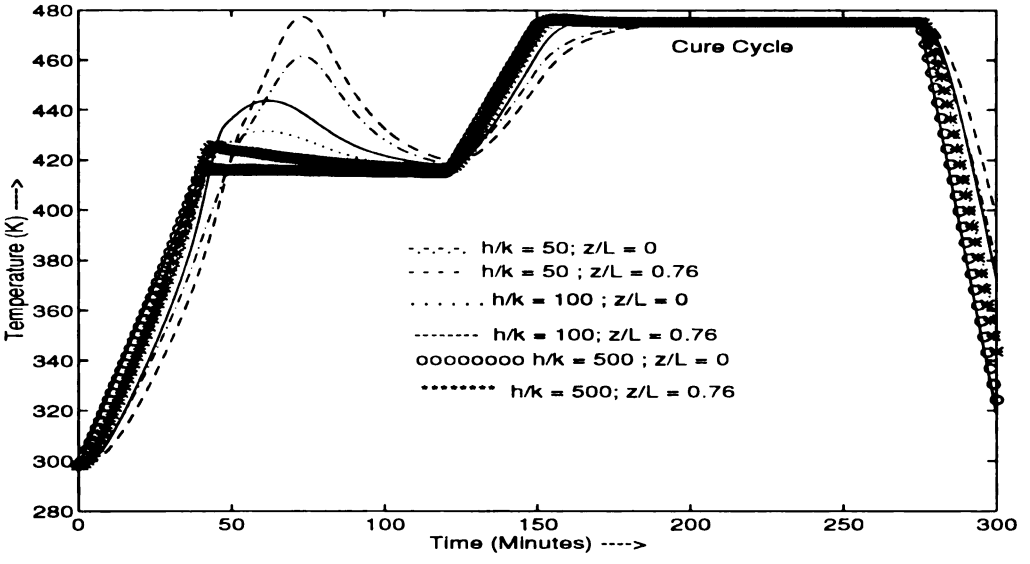



Figure 5.7. **Temperature profiles for varying heat transfer coefficient**

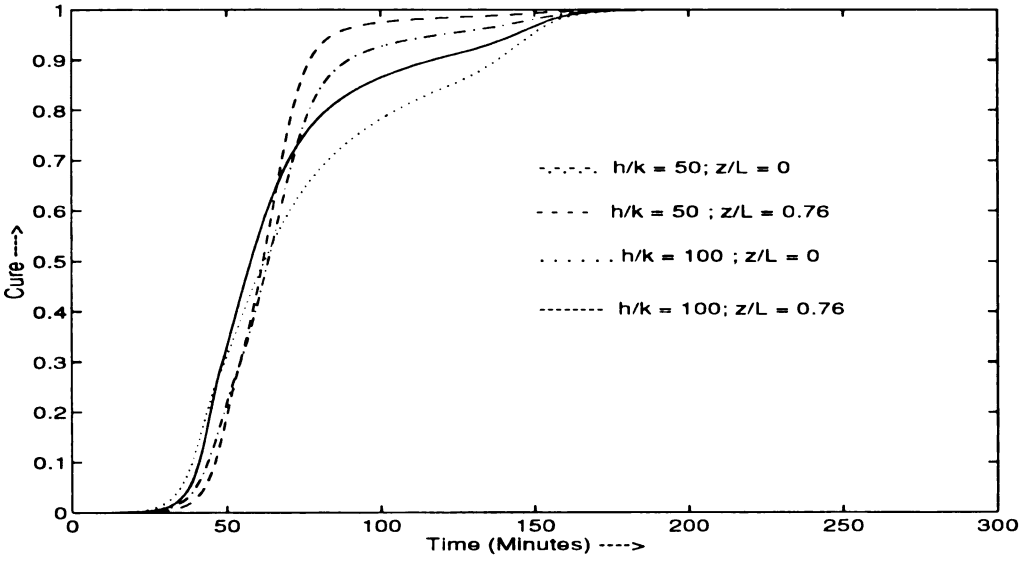



Figure 5.8. **Cure profiles for varying heat transfer coefficient**

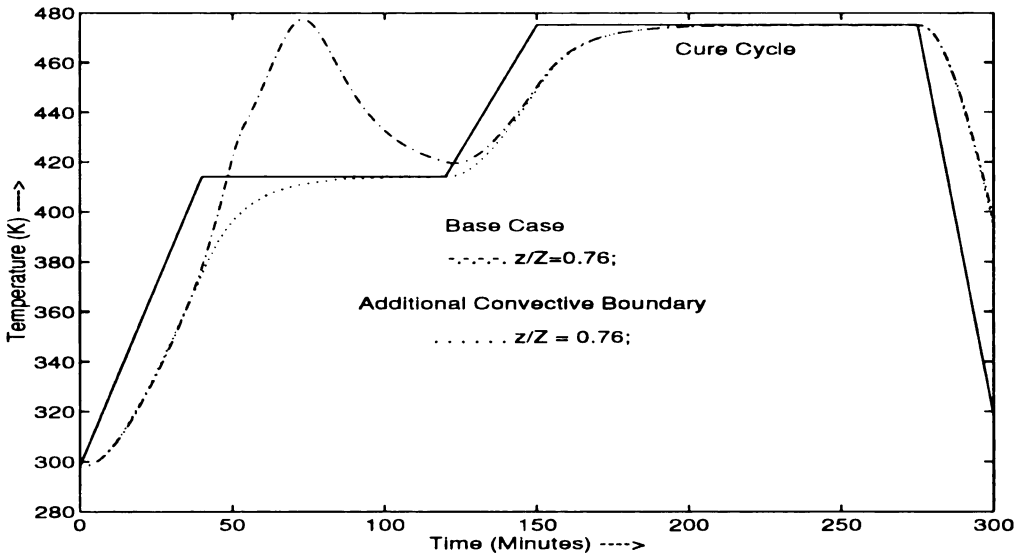



Figure 5.9. **Effect of addition of convective boundary on temperature profiles**

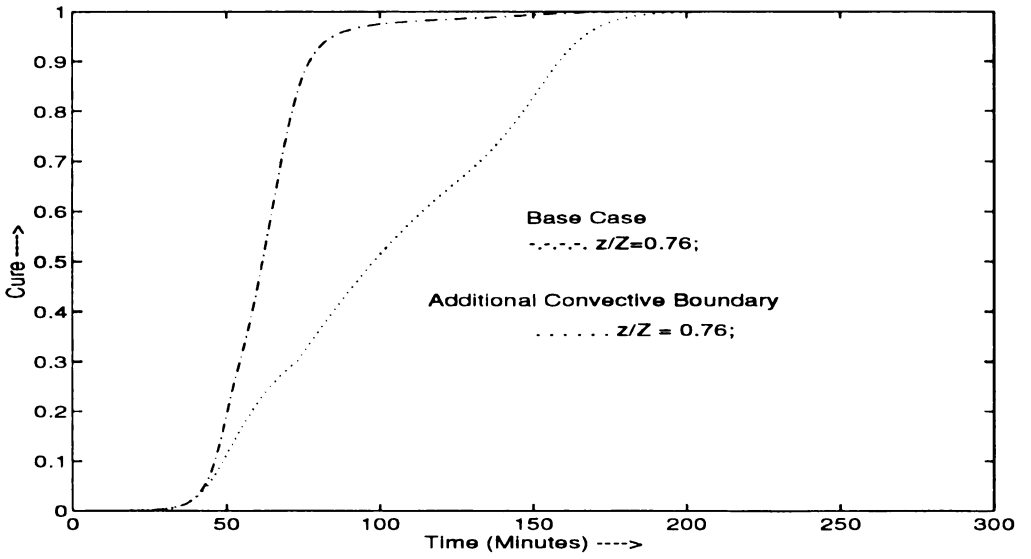



Figure 5.10. **Effect of addition of convective boundary on cure profiles**

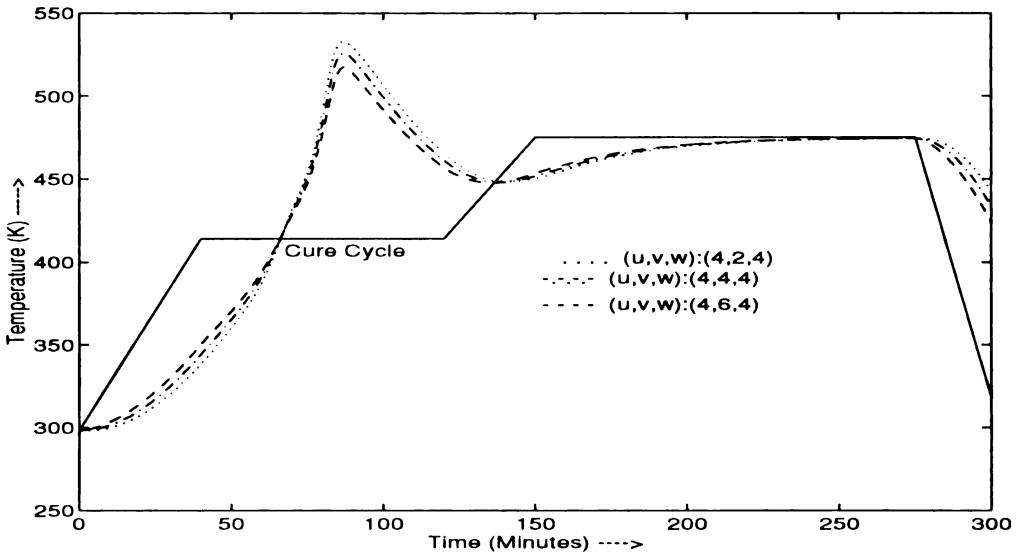



Figure 5.11. **Effect of unsymmetrical boundary conditions along $\eta$ coordinate line**

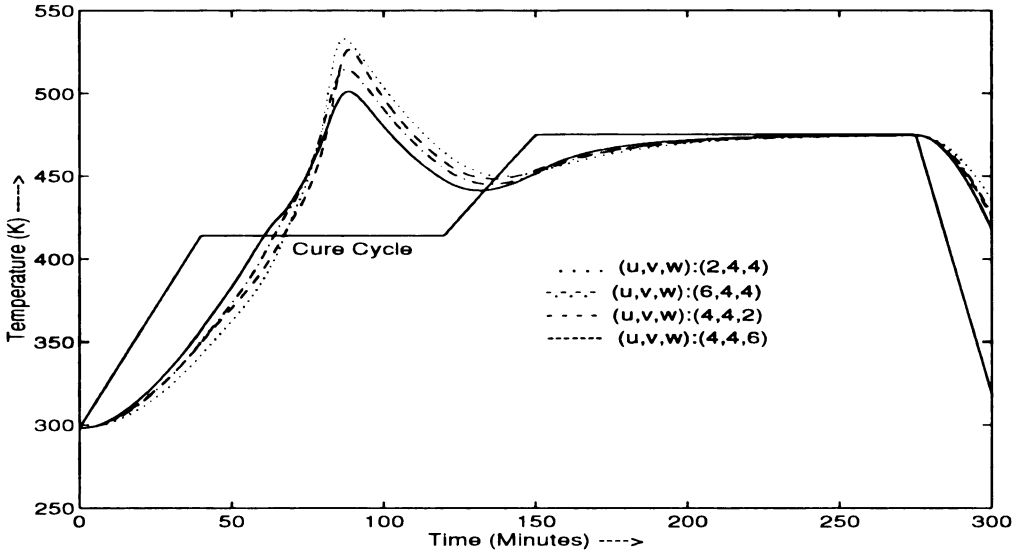



Figure 5.12. **Effect of unsymmetrical boundary conditions along $\xi$, $\zeta$ coordinate lines**

surface of the body. The exotherm is reduced in magnitude but is reached faster as the grid points move closer to the convective boundary. This is because, the effects of convection are more pronounced for points closer to the convective boundary, than ones farther away for the case in which the other boundary is insulated. Consider the Figure 5.12 which shows the temperature profiles along the $\xi$ coordinate line and $\zeta$ coordinate lines from the central point of the body. Comparing the temperature-time behavior of the points (2,4,4) and (6,4,4) with the central point (4,2,4), we find, that in the former case the point lies closer to the convective boundary in the v-direction while it is farther away from the convective boundary in the u-direction. There seems to be some offsetting of the deviations and the resulting plot for (2,4,4) is almost similar to that of point (4,2,4). In the latter case i.e for point (6,4,4) however, the grid point is closer to the convective boundary in both directions and hence a much lower exotherm. A similar analysis performed on the results for the points (4,4,2) and (4,4,6) shown in Figure 5.12 would indicate that the results would be expected to be similar to those of points (4,4,2) and (4,4,6) respectively, in a qualitative sense. However, it is evident from the results that the offsetting that occurred in the previous case does not happen in this case and there is a pronounced difference in the temperature -time behavior for the point (4,4,2) compared to that of (4,4,4). Similarly, the exotherm observed for the point (4,4,6) is much lower than that observed for the point (4,4,2). The underlying basis for this difference is due to the complex nature of the shape and the relative dimensions along the natural coordinate lines. In thick section composites like the one for which the simulation is performed such effects can only be captured by a three dimensional simulation especially when boundary effects across certain boundaries cannot be neglected.

**Effect of Conductivity**

The thermal conductivity of the material determines the magnitude of spatial gradients observed within the material. Three values of thermal conductivity including the base case

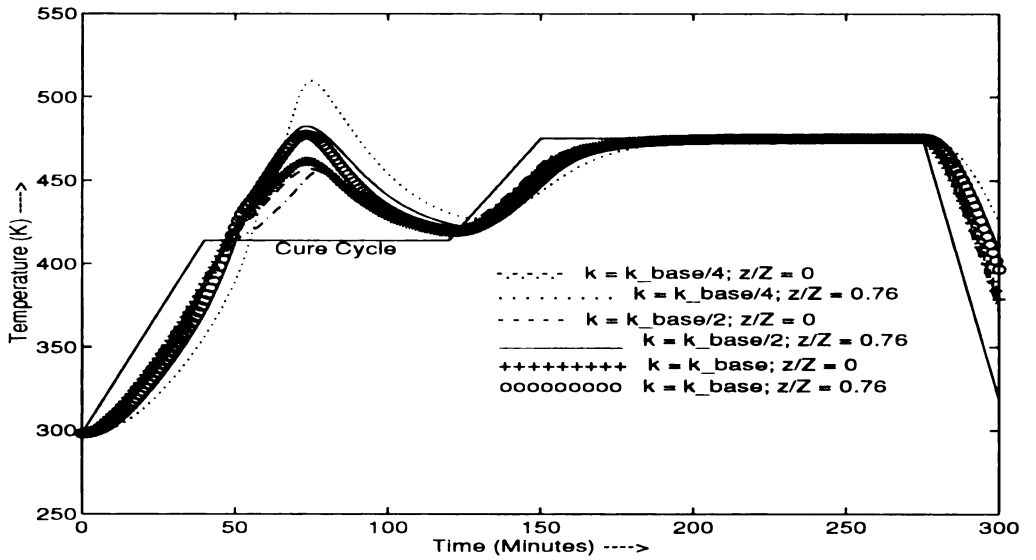



Figure 5.13. **Temperature profiles for varying thermal conductivity**

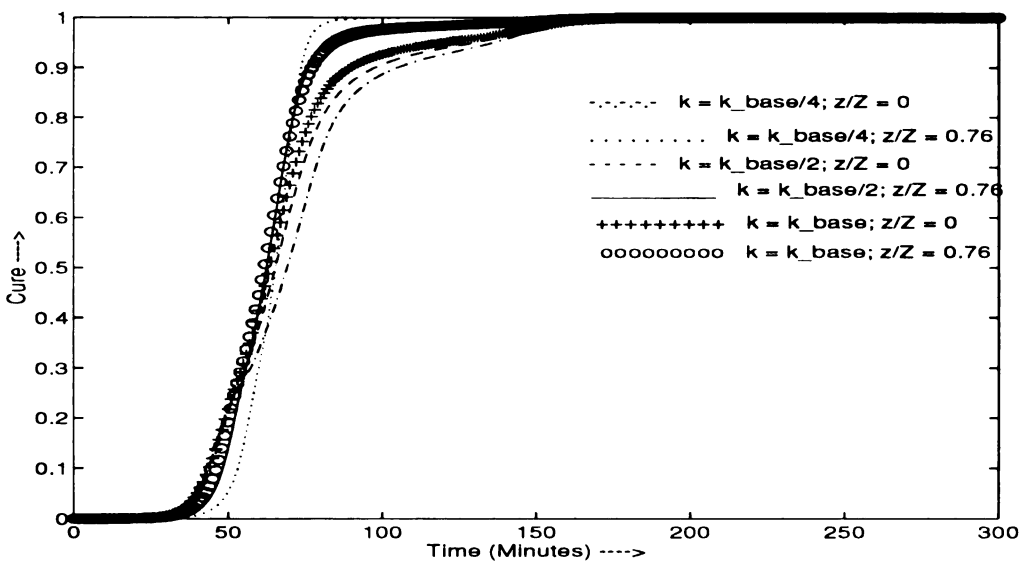



Figure 5.14. **Cure profiles for varying thermal conductivity**

were used with all the other parameters at their base values to study the effects of thermal conductivity. The results are as shown in Figure 5.13. For large values of thermal conductivity, heat is conducted very quickly between the different layers in the material and thus the spatial gradients are small. For lower values of thermal conductivity on the other hand, the heat conduction is slower and hence large differences in the temperature between the different layers occurs, leading to non-uniform curing patterns. These observations are also evident from the simulation results shown in Figure 5.13 and Figure 5.14.

**Effect of Thickness**

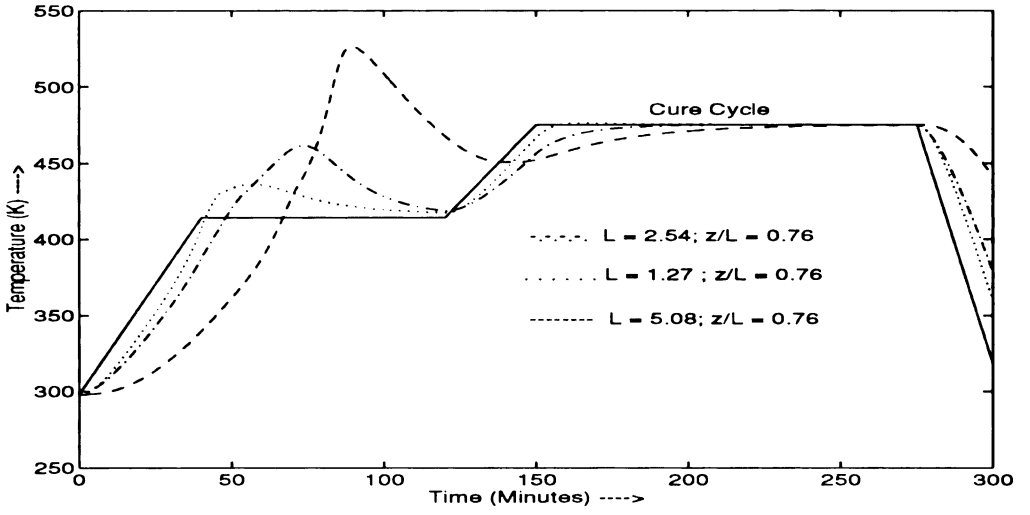



Figure 5.15. **Temperature profiles for varying thickness along $\eta$-direction**

Figure 5.15 shows the effect of thickness as defined earlier on the nature of the temperature profiles. As is evident the thickness of the material along different directions also critically affects the heating patterns as well as the nature of temperature and cure profiles within a material. Three different runs were performed in which the thickness of the V shaped com-

posite between the top and the bottom V surfaces was varied. The results are shown in Figure 5.15 and Figure 5.16. Increasing the thickness of the material causes larger exotherms. This is because the spatial gradients are large as the conduction phenomena is overshadowed by the kinetics due to the higher thickness. On the other hand the time to maximum temperature for the material decreases with increase in thickness. This is because the heat conduction takes longer to provide sufficient heat to the different layers which are now farther away from the surface.

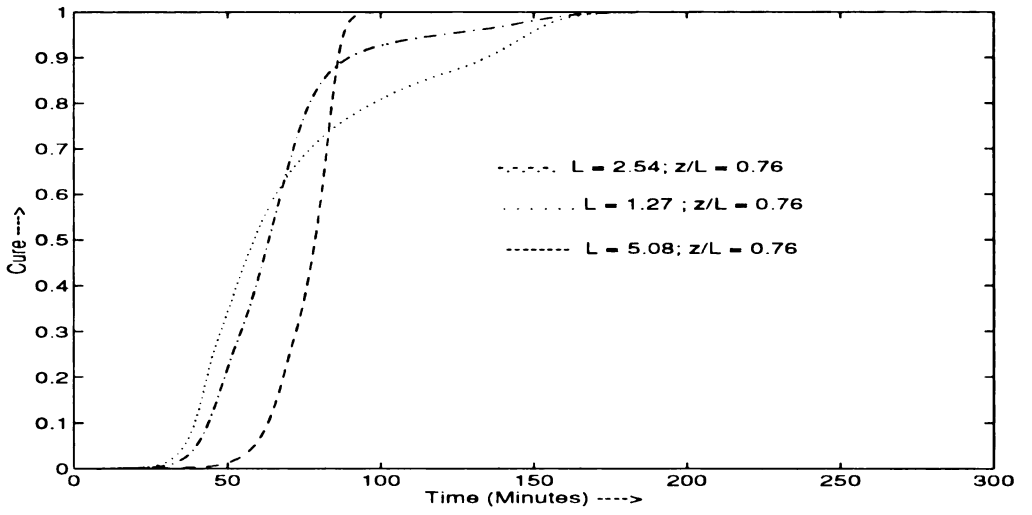



Figure 5.16. **Cure profiles for varying thickness along $\eta$-direction**

## 5.2.3 Numerical Errors

The last section outlined the errors involved in the simulation some simple cases. However, the original problem is highly nonlinear and the only verification that can be performed is through experiment. Nevertheless, the qualitative trends for the general cases give ample

evidence for the correctness of the simulation though the exact accuracy cannot be ascertained. Even given this limitation, it is evident that the numerical simulation is critically dependent on the time step being used for the simulation. The time step affects the nature of the results by influencing both the stability and accuracy of the simulation. For an arbitrary nonlinear problem as the one here no theoretical guarantee is available for the stability of any given method. A good method would give reasonably consistent results for different time steps within the range of its stability. Another factor which affects the accuracy of the results is the grid size. However, for an arbitrary complex shaped body the grid refinement needs to be performed equally well in all directions for solving the process model accurately. Such a refinement causes a large computational burden which is proportional to the cube of the relative grid sizes in any particular direction. This affects the computational speed of both the shape and process simulations. Hence for a reasonably sized grid, additional accuracy is usually sought by decreasing the time step in the process simulation. To identify the magnitude of influence of changing the time step different runs were performed for a thermal cure simulation with large gradients for three values of the time step within the range of stability. This range was determined by trial and error and it is dependent on the nature of the problem. However, it was found that for the various cases run, a time stepping of a maximum of 10 units in time serves well without problems of instability. The caution that is to be adopted here is, the time stepping cannot be larger then the smallest time of change in the thermal cure cycle. This would lead to certain regions of the cure cycle not being captured in the simulation and hence resulting in spurious profiles. The results shown in Figures 5.17 and 5.18 are for the base case except that the cure cycle and the heat of reaction value have been changed to shorten the region of simulation. The results show that accuracy is not greatly affected by using a large time step in spite of the great amount of nonlinearities involved. The maximum error over the entire range of time and space values in going from a time step of 2 seconds to 10 seconds is about 1.0%. This demonstrates the

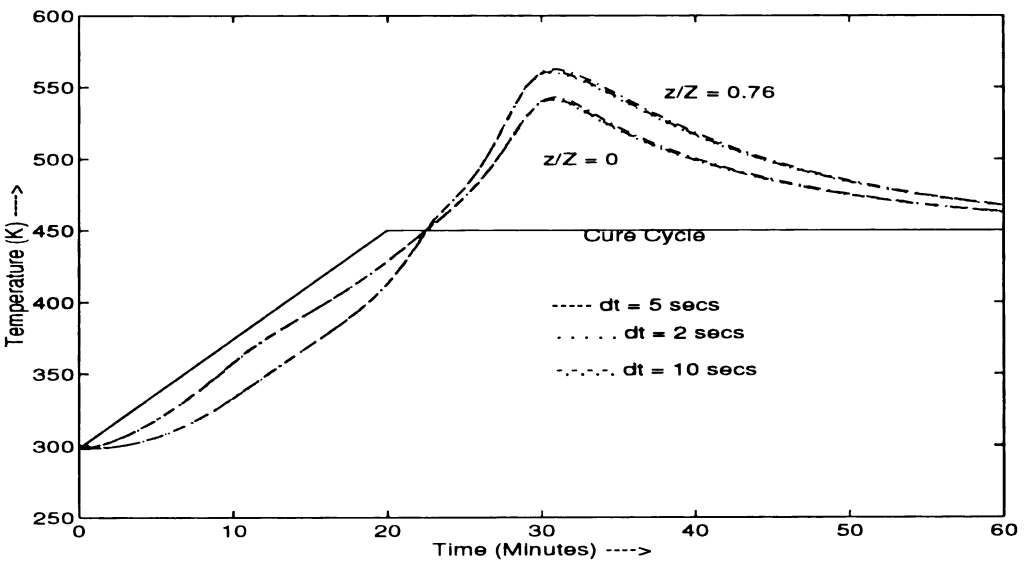



Figure 5.17. **Effect of time stepping on simulation : Temperature**

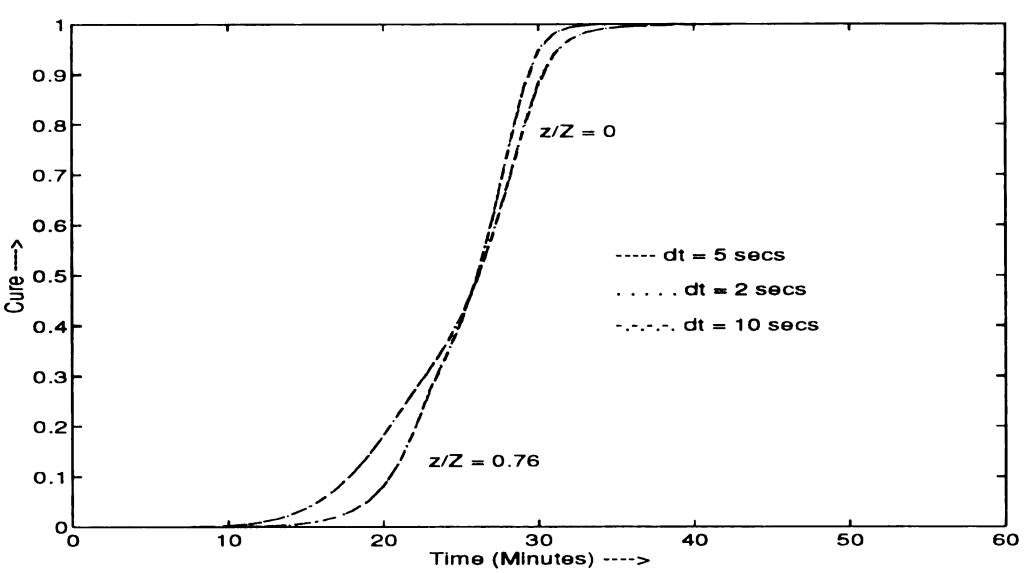



Figure 5.18. **Effect of time stepping on simulation : Cure**

ability of the simulation to perform very well within the range of its stability.

# CHAPTER 6

# Summary and Conclusions

Curing of polymer materials is an important process in the composites industry because it enhances several properties of the composite material . The end properties of the material are critically dependent on how well the process is controlled as well as the process conditions adopted. Thus a general modelling scheme for the process has to take into consideration all these factors. The process parameters not only include the ambient conditions but also the inherent properties of the object such as the shape of the material. The shape, as mentioned earlier in the Thesis influences the nature of interactions between the various phenomena at play during the curing process.

A differential equation based approach to modelling can include all the important aspects of the process as well as capture the various physical phenomena at play. The process model developed in this work is based on a system of differential equations. The curing process itself can be either thermal or microwave depending upon the manner in which heat is supplied to the material and the model is general enough to incorporate the curing mechanism (thermal/microwave). Most of the previous modelling efforts are either one or two dimensional in terms of accounting for the spatial variation of the process variables. Thus they suffer from problems of incomplete process representation in terms of accurately predicting the trends in the temperature and cure profiles. This is especially true for thick sec-

tion composites and for materials of complex shape.

The differential equation based model has been made to include the influence of arbitrary shapes by the use of the boundary fitted coordinate transformation technique. This method has been used to perform the shape modelling whereby the complex geometry of the object being processed is transformed by a suitable coordinate transform to obtain a natural coordinate system. Such a coordinate system has been generated by solving a system of nonlinear elliptic partial differential equations. Though several techniques could be used to arrive at the boundary fitted coordinates the elliptic generation system is more efficient as it captures the continuity of the coordinate lines across surfaces and it needs minimum user input in terms of making specialized decisions regarding the mesh generation aspect. The elliptic generation systems also readily provide the shape factors and the local jacobians for the coordinate transformation which are then directly used in solving the process model. The shape modelling is fairly general and requires only the surface or boundary information of the coordinates of the object under consideration.

The process model that is originally developed in the physical or complex domain is then transformed analytically to another system of differential equations in the natural coordinate domain. This transformation casts the original generalized boundary conditions on the regular boundaries defined by the natural coordinate system. Thus the tedious and error prone normal evaluations are avoided on the boundary of the object when calculating for the surface values of the dependent variables. The resulting system of differential equations are basically of the same nature as that of the generating system and hence the same numerical methods that are applicable for the solution of one can be used for the solution of the other.

Since all the differential equations are highly nonlinear and the boundary information is almost never available in a closed functional form, numerical methods have to employed to obtain the solution to the system of differential equations in the case of mesh generation as well as temperature profile simulation. Vertex centered finite difference formulations were

used to cast the differential equations into a system of algebraic difference equations.

The difference equations are generated by the discretization of the regular domain and the approach to solution is by relaxation of an initial guess. Since the degree of discretization governs the accuracy as well as the computational efficiency of relaxation, a method which makes use of multiple grid sizes is more effective in reducing the computational burden on a single grid relaxation. Thus multi-grid methods have been used to obtain the solution of the difference equations resulting from the differential formulations for both the shape and the process model. Initially a single grid relaxation method(alternating direction implicit method) was used to solve the system of difference equation for the shape generation part. When a multi-grid method was implemented for the same problem with a Gauss-Siedel relaxation strategy the computational time was reduced tremendously. The advantage of the multi-grid methods is the flexibility on the choice of the relaxation scheme. Multi-grid methods improve convergence of single grid relaxations by performing fewer relaxations over multiple grids. The same strategy combined with a Crank-Nicolson time based discretization was used for the solution of the energy balance and the material balance difference equations.

The simulation code was developed in the C language. Extensive work has been done in the area of boundary fitted coordinate system as well in multi-grid solutions to differential equations. Several versions of the implementation of these techniques exist in the public domain as Fortran codes. The difficulty in all the readily available codes is that the problems they handle are not very generic from a dynamic point of view. The problems they have been developed for, are mostly steady state problems and any dynamic problem has to be recast to use the existing code. This leads to problems in integration. But, the major problem in using the public domain tools is that there is no choice of methods available at every level for the programmer to incorporate into the code. For instance all the boundary fitted coordinate system based methods have been implemented in FORTRAN code using

single grid methods as the solution strategies for the difference equations. Besides, the process methodologies also impose restrictions on the kind of data structures that have to be used. In this regard one seldom finds a thermal cure cycle as part of the boundary conditions for instance. Integrating different available codes leads to mismatch of data structures between the component pieces. The most efficient structure from the point of view of a 3D multi-grid method has to be translated into a manipulatable form for the boundary fitted coordinate technique and vice-versa. Thus the option of using public domain tools for developing the code for the simulation was found to be less useful and less attractive compared to the development of an indigenous code for the specific purpose at hand.

The simulation may involve the use of complex shaped objects. Thus visualization of the results of the shape modelling phase of the simulation is important. The simulation presents the temperature and cure profiles as a function of time. A graphical display of the evolving profiles helps the user to view the trends and decide whether to change the grid parameters for better accuracy. A graphical output interface which displays 3D object images as well 2D plots of temperature and cure profiles and color maps of different sections of the heated object has been developed using the MATLAB external graphics library. Finally, the various segments of the model and the code have been tested using different analytical cases for their accuracy. For a general scenario involving a complex shaped body the qualitative trends of the results of the simulation have been analyzed for cases of different parametric variations and have been ascertained to be accurate for the situations considered. The model as well as the code are in their developmental stages and several issues need to be addressed before they could become powerful predictive tools. An outline and a brief discussion of some of the avenues for future work are discussed in the next chapter.

# CHAPTER 7

# Future Work

The previous chapter outlined the capabilities of the simulation as well as demonstrated the accuracy of the method using certain problems for which analytical solutions were available. The main objective of this effort was to provide a generalized framework for predicting different properties of a material being cured under different process conditions. The full potential of the different techniques involved in shape modelling have not been exploited in this effort. There are several tradeoffs that need to be considered when doing an extensive numerical shape modelling. Currently a basic modelling approach with allowances for additional features has been established. There are two main developments that can be made in the immediate future. They are

1. Adaptive mesh generation.

2. Composite and domain decomposed mesh generation

Both the above developments are basically offshoots from the current method and essentially need the current framework for their implementation. The first of the two developments involves rearranging the mesh without altering the entire grid. Figure 7.1 shows the effect of adaptive meshing. The grids are moved towards a particular coordinate line locally, without rearranging the entire grid pattern. Adaptive meshing is done when accurate

information is required, for large gradients of the dependent variable in certain regions of the domain. By attracting neighboring coordinate lines to that region, a local refinement of the mesh is obtained in that region, as shown in Figure 7.1. There are several points that

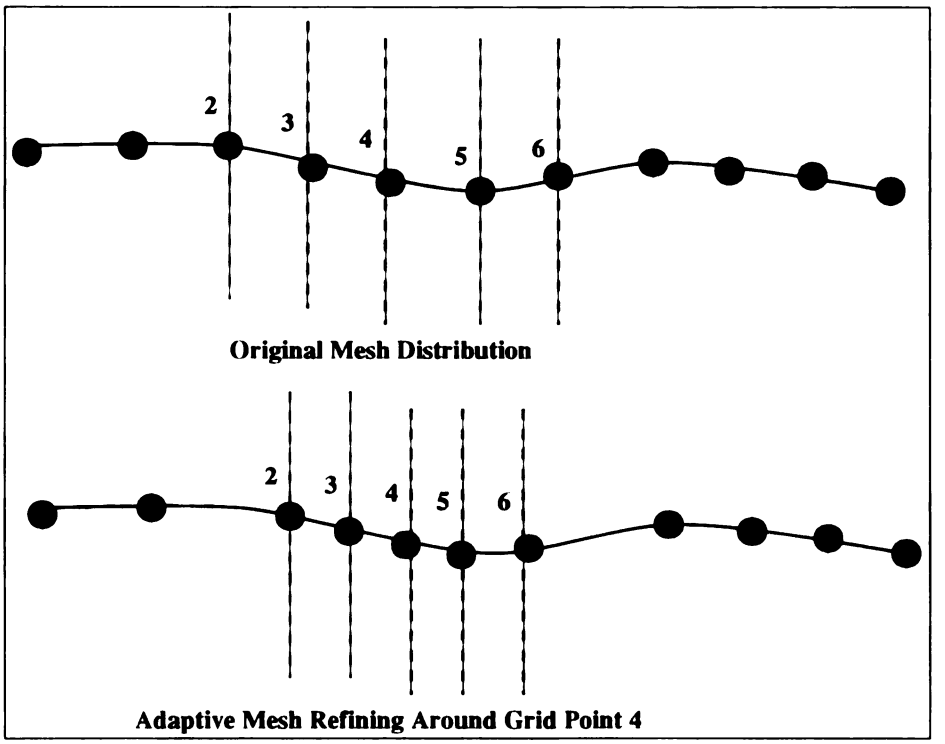



Figure 7.1. **Adaptive Mesh Generation**

need to be considered before actually implementing this method. The first one is that there might not be a necessity to do this because the range of variation in the gradients may not be very large. It is to be noted that adaptive meshing has to be performed at every time step and certain numerical criteria need to be established, like the magnitude of the gradients that has to trigger the re-meshing process. This process is itself iterative and hence consumes some computational power and cuts down on the speed of the method. Usually adaptive

mesh generation is performed for steady state problems and hence the re-meshing is done only once. Nevertheless one can foresee the need to do this in the case of microwave curing. Basically, curing of a material in a thermal autoclave is controlled using a thermal cure cycle. This control is over time because, the variation in the ambient temperature is w.r.t time. At best, a spatial control is obtained by establishing different heat transfer coefficients at different boundaries. In the case of microwave curing several process parameters could be changed to obtain very accurate spatial monitoring and control of the cure process. Thus any system which performs the control for microwave curing needs to monitor the spatial variation of temperature very accurately. Hence there is certainly a need for a numerical simulation system, which can predict large variations in temperature over space accurately in the local neighborhood of that region. Finally, the implementation of the adaptive meshing procedure needs to be discussed. The shape simulation was performed by solving for the system of Equations 2.13. There are no parameters in this system of equations that involve the distribution of the grid independently without changing the surface coordinate distribution. The adaptive meshing procedure involves solving for the set of equations given by,

$$\begin{aligned}
\alpha_{11}x_{\xi\xi} + 2\alpha_{12}x_{\xi\eta} + 2\alpha_{13}x_{\xi\zeta} + \alpha_{22}x_{\eta\eta} + 2\alpha_{23}x_{\eta\zeta} + \alpha_{33}x_{\zeta\zeta} & \\
+\frac{1}{J^2}\left\{x_\xi P(\xi,\eta,\zeta) + x_\eta Q(\xi,\eta,\zeta) + x_\zeta R(\xi,\eta,\zeta)\right\} &= 0 \\
\alpha_{11}y_{\xi\xi} + 2\alpha_{12}y_{\xi\eta} + 2\alpha_{13}y_{\xi\zeta} + \alpha_{22}y_{\eta\eta} + 2\alpha_{23}y_{\eta\zeta} + \alpha_{33}y_{\zeta\zeta} & \\
+\frac{1}{J^2}\left\{y_\xi P(\xi,\eta,\zeta) + y_\eta Q(\xi,\eta,\zeta) + y_\zeta R(\xi,\eta,\zeta)\right\} &= 0 \\
\alpha_{11}z_{\xi\xi} + 2\alpha_{12}z_{\xi\eta} + 2\alpha_{13}z_{\xi\zeta} + \alpha_{22}z_{\eta\eta} + 2\alpha_{23}z_{\eta\zeta} + \alpha_{33}z_{\zeta\zeta} & \\
+\frac{1}{J^2}\left\{z_\xi P(\xi,\eta,\zeta) + z_\eta Q(\xi,\eta,\zeta) + z_\zeta R(\xi,\eta,\zeta)\right\} &= 0 \qquad (7.1)
\end{aligned}$$

The boundary conditions are still the same as in Equation 2.16. The above equations include in them three functions P, Q, R which contain the various parameters to distribute the grid in any desired manner throughout the domain. But, the one to one mapping guarantee provided

by the Laplacian generating system of Equation 2.13 is not valid any more. But for certain classes of functions P,Q,R the mapping still holds (Thompson [13]). Adaptive meshing is discussed in great detail in [3]. Since the generating system for the shape model is now

$$\begin{aligned}\rho(T,X)c_p(T,X)\frac{\partial T}{\partial t} &= \frac{\alpha_{11}(\kappa T_\xi)_\xi + \alpha_{22}(\kappa T_\eta)_\eta + \alpha_{33}(\kappa T_\zeta)_\zeta}{J^2} \\ &+\frac{\alpha_{12}\left\{(\kappa T_\xi)_\eta + (\kappa T_\eta)_\xi\right\}}{J^2} \\ &+\frac{\alpha_{13}\left\{(\kappa T_\xi)_\zeta + (\kappa T_\zeta)_\xi\right\}}{J^2} \\ &+\frac{\alpha_{23}\left\{(\kappa T_\eta)_\zeta + (\kappa T_\zeta)_\eta\right\}}{J^2} \\ &+\kappa\left\{T_\xi P(\xi,\eta,\zeta) + T_\eta Q(\xi,\eta,\zeta) + T_\zeta R(\xi,\eta,\zeta)\right\} \\ &+\rho(T,X)r(T,X)(-\Delta H) + P_m(T,X) \qquad (7.2)\end{aligned}$$

Poisson instead of Laplace the process model is also appropriately transformed to include the new functionalities. The process model equations are now written as in Equation 7.2. The rate equation and the boundary conditions of the transformed process model Equation 2.17 still apply in the same form. The above equation has also to be iteratively solved along with the mesh generation equations at every step to obtain the refined solution at the new grid points. The current simulation code includes the terms P, Q, R, to be defined as functions of natural coordinates in its formulation, but, these terms have been set to zero in accordance with the Laplacian system of generating equations. The adaptive meshing procedure could be readily performed once the parameters and the functionalities of P,Q,R are established.

The second development in the shape modelling area is the method of domain decomposition. For very complex shapes, which are in reality a composite of different three dimensional bodies attached together at different boundaries, a single transformation into a composite cube as it is done currently is not very accurate. This problem could be handled

by considering a series of contiguous cubes each bearing a unique transformation relation to a part of the original domain. Thus the original domain is decomposed into a number of sub-domains each of which is transformed by the boundary fitted coordinate system. Figure 7.2 shows the pictorial representation of the concept of domain decomposition as it applies to the task of generating a shape model for a three dimensional complex object. Miki & Tak-

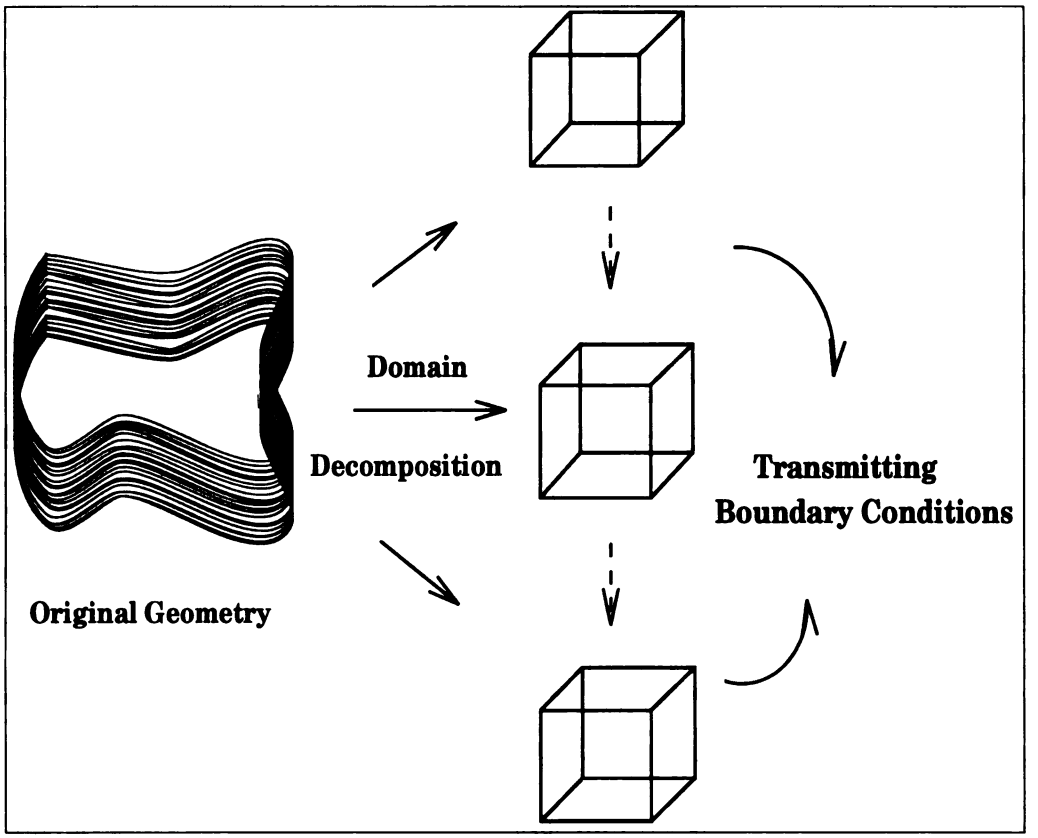



Figure 7.2. **Domain Decomposition Method For Shape Modelling**

agi [14] have performed a domain decomposition algorithm on three dimensional bodies and the same method could be used to solve the shape modelling problem. But, the major computational burden occurs at the next step, which is the profile generation using the pro-

cess model. Since the original domain now consists of different parts each having its own set of shape factors and Jacobian the shape model needs to maintain the continuity of these parameters and this is not automatic. Further, since some of the boundaries of any object or subdomain are contiguous with other boundaries the process parameter conditions along these surfaces needs to be solved iteratively to convergence between the different bodies. Since this step has to be performed at every point in time, this method becomes computationally very expensive for an unsteady state problem. In a real time environment where these predictive results are required, this method might involve a lot of time. Besides, the exact way in which the domain is to be decomposed and the number of component domains are decisions to be made by the user. There do not exist, at least at the present time, any generalized criteria to make these decisions. Thus, the accuracy of the required solution, the time available for simulation as well as the limitations on the amount of information that could be provided to the code have to be considered before going on to the actual implementation of this method. However, the basis of each transformation explained above is still the same as for a single independent body and hence the basic routines and code developed for this simulation can be used with few or no modifications.

From a process point of view there are some enhancements that can be made. Several simplifications have been made in performing the simulation as far the process is concerned and these were outlined in Chapter 2. The most important piece is the microwave power absorption model. A model which incorporates, the dimensionality of the problem, the complex nature of shape involved as well as dependence of dielectric properties on temperature and cure needs to be developed and tested. When available, this could be readily incorporated into the model. Several control criteria, such as, mode-switching have been evolved for the microwave curing process. Once a working model for power absorption is available these criteria could be included into the simulation to examine the effectiveness of these criteria for various shapes. The next major effect that has not been included in the present

work is the anisotropy of the material properties especially the thermal conductivity. This would greatly affect the formulation of the process model since the thermal conductivity in the vectorial flux balance is now a tensorial quantity instead of a scalar quantity. This would also change the formulation of the transformed process model with the inclusion of cross-derivatives for the temperature as well as derivatives for the Jacobians. The neccessary formulations in this case are provided in [1]. Since all the necessary primitives such as derivatives of three dimensional quantities are already present in the current framework the simulation for this case can be implemented using the same code with the additional terms in the process model. Finally, the simulation framework provides an interactive link between the different phenomena at play during the curing process. Each of the sub-models that is used is normally tested experimentally, independent from the effects of other phenomena. When used together in a simulation module these models may need to be corrected to include the effects of other interacting phenomena. Thus, experimental verification of the results of the simulation, once various sub-models have been incorporated, is necessary for the understanding and control of the curing process by different means.

# BIBLIOGRAPHY

# BIBLIOGRAPHY


[1] T. A. Bogetti and J. Gillespie, Jr., "Two-dimensional cure simulation of thick thermosetting composites," *Journal of Computational Physics*, vol. 25, pp. 1–273, 1991.

[2] C. Mastin and J. F. Thompson, "Transformations of three-dimensional regions onto rectangular regions by elliptic systems," *Numerische Mathematik*, vol. 29, pp. 397–407, 1978.

[3] J. F. Thompson, Z. Warsi, and C. W. Mastin, *Numerical Grid Generation : Foundations and Applications*. Amsterdam: Elsevier Science, 1985.

[4] W. L. Briggs, *Multigrid Tutorial*. Society for Induatrial and Applied Mathematics, 1987.

[5] W. Hackbusch, *Multi-Grid Methods and Applications*. Berlin: Springer-Verlag, 1985.

[6] S. F. McCormick, *Multi-level Adaptive Methods for Partial Differential Equations*. Philadelphia: Society for Industrial and Applied Mathematics, 1989.

[7] S. F. McCormick, Ed., *MultiGrid Methods*. Philadelphia: Society for Industrial and Applied Mathematics, 1987.

[8] W. H. Press, *Numerical Recipes in Fortran the Art of Scientific Computing*. Cambridge University Press, 2 ed., 1992.

[9] A. Brandt, "Multi-level adaptive solutions to boundary-value problems," *Math. Comp*, vol. 31, pp. 333–390, 1977.

[10] The MathWorks Inc., *The MATLAB External Interface Guide for UNIX Workstations*.

[11] The MathWorks Inc., *The MATLAB User's Guide for UNIX Workstations*.

[12] J. Crank, *The Mathematics of Diffusion*. Oxford University Press, 1956.

[13] J. F. Thompson, Z. Warsi, and C. W. Mastin, "Boundary fitted coordinate systems for numerical solution of partial differential equations - a review," *Journal of Computational Physics*, vol. 47, pp. 1–108, 1982.

[14] K. Miki and T. Takagi, "A domain decomposition and overlapping method for the generation of three-dimensional boundary-fitted coordinate systems," *Journal of Computational Physics*, vol. 53, pp. 319–330, 1984.

[15] P. R. Eiseman, "Coordinate generation with precise controls over mesh properties," *Journal of Computational Physics*, vol. 47, pp. 331–351, 1982.

[16] I. Woo and G. Springer, "Microwave curing of composites," *Journal of Composite Materials*, vol. 18, pp. 387–409, 1984.

[17] C. W. M. Joe F. Thompson, Frank C. Thames, "Automatic numerical generation of body-fitted curvilinear coordinate system for field containing any number of arbitrary two-dimensional bodies," *Journal of Computational Physics*, vol. 15, pp. 299–319, 1974.

[18] J. Douglas, Jr., "Alternating direction implicit methods for three space variables," *Numerische Mathematik*, vol. 4, pp. 41–63, 1962.

[19] W.-H. Chu, "Development of a general finite difference approximation for a general domain-part 1 : Machine transformation," *Journal of Computational Physics*, pp. 392–408, 1971.

[20] A. Amsden and C. Hirt, "A simple scheme for generating general curvilinear grids," *Journal of Computational Physics*, vol. 11, pp. 348–359, 1973.

[21] R. B. Bird, W. E. Stewart, and E. N. Lightfoot, *Transport phenomena*. John Wiley and Sons, 1960.

# APPENDICES

# APPENDIX A

# User's Guide

## A.1 The Simulation Code

The simulation code has been written in C to be executed on a UNIX operating system. Currently the simulation has been run on the SUN and Hewlett Packard workstations. The process of running the simulation on a given workstation consists of two parts.

1. Compilation
2. Execution

The compilation step involves the generation of the executable file for the code using the different source files. The execution part is the actual running of the simulation under different conditions. Normally compilation is an inbuilt operation for any code. But in this case, UNIX environments are available on SUN and HP workstations which differ in their architecture and computational speeds. Programs compiled under one architecture are not executable in the other. Even the compilation commands are different and the system files to be included in each of these cases reside in different system directories. Besides, the simulation code at this stage is not stand alone in the sense of being complete only with the data specified through the various input files. Functionalities such as the rate equation need to

```
sundar:new_mesh> make -f HP.mk
        cc  -g -I/usr/local/matlab/extern/include -c mesh_generation.c
        cc  -g -c memory.c
        cc  -g -c shape_in_out.c
        cc  -g -c matrix_operations.c
        cc  -g -I/usr/local/matlab/extern/include -c shape_guess.c
        cc  -g -I/usr/local/matlab/extern/include -lm  -c plot_surfaces.c
        cc  -g -lm  -c derivatives.c
        cc  -g -lm  -c shape_solve.c
        cc  -g -lm  -c multigrid.c
        cc  -g -c profile_simulate.c -I/usr/local/matlab/extern/include
        cc -g -lm  -c properties.c
        cc -g -c process_conditions.c
        cc -g -c process_in_out.c
        cc -g -c solve_cure.c
        cc -g -c power_model.c
        cc -g -lm  -c temp_grid.c
        cc -g -lm  -c fictitious.c
        cc -g -I/usr/local/matlab/extern/include -lm  -c plot_profiles.c
        cc -g mesh_generation.o memory.o shape_in_out.o matrix_operations.o \
        shape_guess.o plot_surfaces.o derivatives.o shape_solve.o multigrid.o \
        profile_simulate.o properties.o process_conditions.o process_in_out.o \
        solve_cure.o power_model.o temp_grid.o fictitious.o plot_profiles.o \
        /usr/local/matlab/extern/lib/hp700/libmat.a  \
        -I/usr/local/matlab/extern/include -lm  -o profile_gen
```

Figure A.1. **Compiling the code**

be specified and these are done at this stage by changing the appropriate routine within the program. Once the code is changed it has to be recompiled and thus this makes compilation an important aspect in the running of the simulation.

The UNIX environment provides for certain files and commands which can specify and control the sequence of the compilation operation itself. These are called the *make* files and have a '.mk' extension to their name. Two such files are provided with the simulation, namely *SUN.mk* and *HP.mk* for use under the two different architectures. Use of the appropriate file in the *make* command compiles the different C programs and links them to form an executable file. Figure A.1 shows an example of a compilation session on the SUN workstation. Every time the code needs to be compiled the make process can be used. Another

Table A.1. **Components of Code : MATLAB Script Files**

| Script File | Comments |
|---|---|
| arrange_shape.m | Arranges matrix for 3D plot. |
| distribute.m | Distributes matrix for plots between coordinate lines |
| find_pic.m | Finds named plot |
| init_display.m | Initializes display variables |
| shape_plot.m | Generates 3D shapes and maps |
| surf_plot.m | Generates surfaces of 3D volume |
| time_tplots.m time_xplots.m | Generates time plots for temperature and cure. |

advantage of *make* files is the fact that they compile only the updated files and use the object files from the other previously compiled programs. Thus in a code consisting of many programs only the modified ones are compiled instead of the whole set. The README file provided in the Appendices along with the make files gives the various directories in which the system files i ncluded within the program reside in different architectures (HP/SUN etc). Once the compilation is complete, the executable file is generated and resides in the file *profile_generate*. This filename can be used as a command in the UNIX environment and when used so it executes the simulation code. The user is allowed the option of proceeding to the profile simulation or just stop with the mesh generation aspect. This is essential, because the accuracy of the shape model needs to be ascertained before moving to the predictive part in some cases. The entire code consists of a list of 18 C programs and 8 MATLAB(R) script files (with '.m' extensions). Table A.1 and Table A.2 give the names of the various programs and their area of operation. The details of these programs are presented in Appendix B. As mentioned earlier some of these programs need to be modified by the user, to take care of model changes, and the subroutines that need to be changed in such cases contain comments, that indicate this as well as other routines that need to be changed.

Table A.2. **Components of Code : C Language Programs**

| Program | Comments |
|---|---|
| mesh_generation.c | Control program for entire code. |
| shape_in_out.c | Reads in surface or edge coordinates and stores in appropriate format. |
| shape_guess.c | Performs algebraic guess generation using Transfinite interpolation |
| derivatives.c | Evaluates different derivatives |
| memory.c | Performs dynamic memory allocation and freeing. |
| matrix.c | Performs various matrix operations |
| multi-grid.c | Contains all routines and operators for the Multi-grid method |
| shape_solve.c | Solves elliptic generation system |
| profile_simulate.c | Control program for profile generation |
| process_in_out.c<br>process_conditions.c<br>properties.c | Read in process and property data and evaluates process conditions and material properties |
| solve_cure.c | Calculates cure using rate equation |
| power_model.c | Evaluates MW power absorbed (Has to be updated) (Not in use currently) |
| plot_profiles.c<br>plot_surfaces.c | Manipulate data structures and and generate plots and maps. |
| temp_grid.c | Contains all operations to solve the transformed process model |
| fictitious.c | Evaluates boundary conditions using fictitious surface. |

## A.2 Input Files

The structure of the code is such that all the user specified data such as surface coordinates, process conditions, rate equation parameters are modified according to the requirements before the code is executed. These is done by the use of input files which are files with a '.DAT' extension. The above construction is specifically based on the fact that, in the future the numerical simulation module needs to be integrated into an expert control system for the curing process. Data files can easily be updated with property and other information by the control-

Table A.3. **Input files for simulation code**

| File | Content |
|---|---|
| PROPERTY.DAT | Properties of material and kinetic parameters |
| PROCESS.DAT | Process variables such as heat transfer coefficient, cure cycle information |
| NUMERICAL.DAT | Simulation parameters such as time step, mesh size, number of relaxation sweeps etc. |
| DISPLAY.DAT | Plotting options like display time, surface and time plot options. |
| COORD.DAT | Surface coordinate specification for complex geometry |

ling system and this would allow information flow from the different components in the expert system to the numerical module. These files contain different types of data which have to be changed according to user preferences before the simulation is executed. Table A.3 gives the list of different input files that are used and the type of information contained in each of them. The Appendices contain a sample of all these data files with the details of the exact sequence of information to be provided to the program. Once the modifications are completed the simulation code can be executed. The input files also contain information regarding the display of the different kinds of plots. Accordingly, the simulation code displays different shape and surface plots of the types shown in Chapter 5. There is minimum user input information to be provided during the execution of the code. There are however, facilities provided for the expert MATLAB user to execute standard MATLAB commands from within the simulation to manipulate the various plots generated. The program provides these options to the user at execution time and whenever shape plots are generated by the program.

The construction of all except one of the files is very straightforward. The commented listings in the following pages describe the various input files used and the data presented in

each. The surface coordinate information has to be provided in a certain format to be meaningfully used by the code for shape modelling. The shape to be generated is envisioned as system of packed surfaces and each of these surfaces is a system of orthogonal or nonorthogonal coordinate lines. The first step in providing the surface coordinates is to identify the direction of packing of the surfaces and the direction of movement of coordinate lines within each of these surfaces. Thus three coordinate directions are obtained for the surface generation. Along each of the bounding surfaces two of the natural coordinate lines vary and the third is assumed to be constant. The surface information as read by the program is to be provided on six bounding surfaces for the transformed cube. Thus the shape is actually assumed to be cut up into six bounding surfaces in any particular manner for providing the surface coordinate information. If $\xi$, $\eta$, $\zeta$ are the three natural coordinates, then the input file is constructed in three steps as follows. The variables n, m, l are the number of natural coordinate grid points along $\xi$, $\eta$ and $\zeta$ directions respectively.

1. $\xi$ = 0,n-1 ($\eta$ = 0 to m-1 [$\zeta$ = 0 to l-1])

2. $\eta$ = 0,m-1 ($\zeta$ = 0 to l-1 [$\xi$ = 0 to n-1])

3. $\zeta$ = 0,l-1 ($\xi$ = 0 to n-1 [$\eta$ = 0 to m-1])

In the above steps the brackets indicate nesting of the coordinate information to be provided to the program. For example, the first step implies that at either end of the $\xi$ coordinate line, for every point along the $\eta$ coordinate line, the surface coordinate information at all the $\zeta$ coordinate lines has to be provided. The ordering of this information is important and has to follow the sequence shown in the above steps. It is suggested that while generating the coordinate system a right hand coordinate orientation be maintained i.e., move along coordinate directions in such a manner, as to keep the surface being generated to the right. Once the input files are updated the simulation can be executed as explained in the previous section.

# APPENDIX B

# Supporting Files

## B.1 Make File Setup

**File : README**

Make File Changes
The main system dependent change that needs to be nmade to the make file is the resident directories and programs for the MATLAB external libraries.

SUNOS :
INCLUDE_PATH = -I/usr/local/matlab/extern/include
libmat.a resides in /usr/local/matlab/extern/lib/sun4/
engine.h resides in /usr/local/matlab/extern/include/

HP-UX :
INCLUDE_PATH = -I/usr/local/matlab/extern/include
libmat.a resides in /usr/local/matlab/extern/lib/hp700
engine.h resides in /usr/local/matlab/extern/include

Solaris :
INCLUDE_PATH = -I/opt/matlab4.2/extern/include
libmat.a resides in /opt/matlab4.2/extern/lib/sol2/libmat.a
engine.h resides in/opt/matlab4.2/extern/include

NOTE: In all the above search paths the directory 'matlab' can be any version of matlab for e.g 'matlab4.0a' or 'matlab4.2' etc.

**File : Sample.mk**

```
CFLAGS= -g
LIBS= -lm
INCLUDE_PATH = -I/usr/local/matlab/extern/include

all: prof_compile

prof_compile: mesh_generation.o memory.o shape_in_out.o
matrix_operations.o \
        shape_guess.o plot_surfaces.o derivatives.o shape_solve.o multigrid.o \
        profile_simulate.o properties.o process_conditions.o  process_in_out.o \
        solve_cure.o power_model.o temp_grid.o fictitious.o plot_profiles.o ;
        $(CC) $(CFLAGS) mesh_generation.o memory.o shape_in_out.o
matrix_operations.o \
        shape_guess.o plot_surfaces.o derivatives.o shape_solve.o multigrid.o \
        profile_simulate.o properties.o process_conditions.o process_in_out.o \
        solve_cure.o power_model.o temp_grid.o fictitious.o plot_profiles.o \
        /usr/local/matlab/extern/lib/sun4/libmat.a $(INCLUDE_PATH) $(LIBS)
-o profile_gen


profile_simulate.o: profile_simulate.c
/usr/local/matlab/extern/lib/sun4/libmat.a grid.h
        $(CC)  $(CFLAGS) -c profile_simulate.c $(INCLUDE_PATH)

properties.o: properties.c
        $(CC) $(CFLAGS) $(LIBS) -c properties.c

process_conditions.o: process_conditions.c
        $(CC) $(CFLAGS) -c process_conditions.c

process_in_out.o: process_in_out.c
        $(CC) $(CFLAGS) -c process_in_out.c

solve_cure.o: solve_cure.c
        $(CC) $(CFLAGS) -c solve_cure.c

power_model.o: power_model.c
        $(CC) $(CFLAGS) -c power_model.c

temp_grid.o: temp_grid.c grid.h
        $(CC) $(CFLAGS) $(LIBS) -c temp_grid.c
```

```
fictitious.o : fictitious.c grid.h
	$(CC) $(CFLAGS) $(LIBS) -c fictitious.c

plot_profiles.o: plot_surfaces.o plot_profiles.c grid.h  \
		/usr/local/matlab/extern/include/engine.h
	$(CC) $(CFLAGS) $(INCLUDE_PATH) $(LIBS) plot_surfaces.o -c
plot_profiles.c

multigrid.o:  multigrid.c grid.h
		$(CC)  $(CFLAGS) -c multigrid.c

shape_solve.o :	shape_solve.c grid.h
		$(CC)  $(CFLAGS) $(LIBS) -c shape_solve.c

derivatives.o :	derivatives.c
		$(CC)  $(CFLAGS) $(LIBS) -c derivatives.c

plot_surfaces.o : plot_surfaces.c /usr/local/matlab/extern/include/engine.h
		$(CC)  $(CFLAGS) $(INCLUDE_PATH) -c plot_surfaces.c

shape_guess.o :	shape_guess.c
		$(CC)  $(CFLAGS) $(INCLUDE_PATH) -c shape_guess.c

matrix_operations.o : matrix_operations.c
			$(CC)  $(CFLAGS) -c matrix_operations.c

shape_in_out.o : shape_in_out.c
		$(CC)  $(CFLAGS) -c shape_in_out.c

memory.o : memory.c grid.h
		$(CC)  $(CFLAGS) -c memory.c

mesh_generation.o : mesh_generation.c grid.h
/usr/local/matlab/extern/include/engine.h
			$(CC)  $(CFLAGS) $(INCLUDE_PATH) -c mesh_generation.c
```

## B.2 Input Files

**File : COORDINATE.DAT**

```
yes /* Volumetric shape plot option */
2 /* 1 - Surface info available from input, 2 - Edge info. available */
/* The next three inputs indicate the number of grid points along Psi, Eta
and
/* Zeta directions */
9
9
9
/* The next three inputs are the scale factors for the x,y and z coordinates
/* along Psi, Eta and Zeta directions respectively */
0.0625
0.0625
0.0625
/* The next three inputs are */
/*      1. No of V sweeps */
/*      2. The number of pre smoothing operations for mesh generation */
/*      3. The number of post smoothing operations for mesh generation */
/* respectively */
2
5
5
/* File into which the final results of mesh generation are stored */
Temp_new.m
/* The rest of this file is the coordinate information in surface or edge  */
/* See the Appendix A of the Thesis for a description of the format of */
/* this input */
   0   0   0
   0   1   0
   0   2   0
   0   3   0
   0   4   0
   0   5   0
   0   6   0
   0   7   0
   0   8   0
   1   0   0
   1   1   0
   .   .   .
   .   .   .
   9   9   9
```

## File : DISPLAY.DAT

```
5 /* Display time every 'n' time units */
4 /* The next three inputs indicate the */
4 /* surfaces for which the color maps are */
4 /* to be plotted */
yes /* Option for time Vs variable plots */
no /* Option for surface color maps */
Temp  /* Indicates the variable to be plotted for  */
        /* Temp - Temperature */
        /* Cure - Cure */
       /* Both - Temperature and Cure */
       /* Choosing this option also saves the generated data */
       /* into matlab binary '.mat' files. */
```

## File : NUM.DAT

```
300 /* Final time upto which simulation needs to be performed */
5 /* Time step :- delt */
5 /* The last three parameters are the number of pre and post smoothing
5 /* relaxation sweeps and the number of V iterations respectively. */
5
5 /* Time step (delt) */
```

## File : PROCESS.DAT

```
y /* Type of conv coeff y => constant on each of the 6 surfaces
     n => different at each point on each of the 6 surfaces */
0
0
0.0018
0.0018
0
0
2 / * Number of steps in thermal cure cycle*/
0 /* Startup time */
375.15 /* 1st step end temperature */
900 /* Second step startup time */
450.15  /* Second step end temperature */
4800 /* Third step startup time */
450.15/* Third step ending temperature */
303.15 /* Initial temperature */
0 /* Initial cure */
10000 /* Final time for thermal cure cycle*/
```

## File : PROPERTY.DAT

/* Property data : This is subject to change depending on the functionalities
/* used within the program for the different properties */
4.457e-3(Thermal Conductivity)
0.940(Specific Heat)
0(Fiber volume fraction)
1.520(Density of material)
/* RxN data */
35.033e6 (k1o)
8.07e4 (E1)
-33.5667e6 (k2o)
7.78e4 (E2)
3266.6667 (k3o)
5.66e4 (E3)
0.60 (X_gel)
0.47 (B)
-175.20 (DELTA H)

# APPENDIX C

## Matlab Script Files

### Program: init_display.m

```
% This is an initial setup file for Matlab plots. Edit the indicated portion
% of this file before choosing plot option in the COORD.DAT file.
global tfinal tdisp imagp Tt Xt time Tin Xin x y z;
tfinal=0;
tdisp=0;
% Edit the following matrix 'imagp' . Each row of the matrix indicates the
% natural coordinate point of the shape. Simultaneous plots of all these
% points is done during the simulation.
imagp=[
4 0 4
4 2 4
4 4 4
4 6 4
4 8 4
];
Tt=[];
Xt=[];
time=[];
Tin=[];
Xin=[];
x=[];
y=[];
z=[];
```

## Program: find_pic.m

```
% This function finds the picture/figure with a given title/name.
function handle = find_pic(string)
f=1;
g=0;
while(g~=1),
str1=get(f,'name');
if(strcmp(string,str1)==1)
g=1;
else
f=f+1;
end;
end;
if(g==1)
handle=f;
else
handle=gcf;
end;
```

**Program: arrange_shape.m**

```
% This function arranges the dimension of the 3D matrix to reflect different
% primary directions.
function [xx,yy,zz] = arrange_shape(x,y,z,n)
nr=max(size(x))/n;
xx=reshape(x,nr,n);
yy=reshape(y,nr,n);
zz=reshape(z,nr,n);
```

## Program : distribute.m

```
% The 3D volume is visualized as a stack of surfaces. These surfaces can be
% stacked along different natural coordinate directions.
% The following function redistributes these stacks along a given direction
% specified by the index argument.
function f = distribute(x,n,m,l,index)
% x - 3D volumetric matrix
% n,m,l - 3D dimensions
% index - 1,2 or 3 specifying one of the 3 natural coordinate directions,
% psi, eta or zeta.
if (index == 1)
f=x;
else
s=size(x);
if index == 2,
count_new=1;
jumpi=m*l;
jumpj=m;
for j=1:m,
for k=1:l,
for i=1:n,
count_old=(j-1)*jumpj+(i-1)*jumpi+k;
if min(s) > 1,
if s(1) < s(2),
f(1:min(s),count_new)=x(1:min(s),count_old);
else
f(count_new,1:min(s))=x(count_old,1:min(s));
end;
else
f(count_new)=x(count_old);
end;
count_new=count_new+1;
end;
end;
end;
else
count_new=1;
jumpi=m*l;
jumpj=m;
for k=1:l,
for i=1:n,
for j=1:m,
```

```
count_old=(j-1)*jumpj+(i-1)*jumpi+k;
if min(s) > 1,
if s(1) < s(2),
f(1:min(s),count_new)=x(1:min(s),count_old);
else
f(count_new,1:min(s))=x(count_old,1:min(s));
end;
else
f(count_new)=x(count_old);
end;
count_new=count_new+1;
end;
end;
end;
end;
end;
```

**Program : surf_plot.m**

```
% Plots the different surfaces of the 3D volumetric matrix.
function h = surf_plot(x,y,z,i,n,c)
if nargin == 5,
c=abs(sqrt(x.^2+y.^2+z.^2));
end;
s=size(x);
nr=max(s);
m=round(nr/n);
if s(1) > s(2),
xg=reshape(x([1:nr],i),n,m);
yg=reshape(y([1:nr],i),n,m);
zg=reshape(z([1:nr],i),n,m);
cg=reshape(c([1:nr],i),n,m);
else
xg=reshape(x(i,[1:nr]),n,m);
yg=reshape(y(i,[1:nr]),n,m);
zg=reshape(z(i,[1:nr]),n,m);
cg=reshape(c(i,[1:nr]),n,m);
end;
surf(xg,yg,zg,cg);
```

## Program : shape_plot.m

```
% Recursively calls the surf_plot function to recreate the volume of the
% complex shaped body.
function h = shape_plot(x,y,z,n,m,l,index,c)
if nargin == 7,
c=abs(sqrt(x.^2+y.^2+z.^2));
end;
x=distribute(x,n,m,l,1);
y=distribute(y,n,m,l,1);
z=distribute(z,n,m,l,1);
nr=round(max(size(x))/n);
[xx,yy,zz]=arrange_shape(x,y,z,n);
c=reshape(c,nr,n);
for i=1:n,
surf_plot(xx,yy,zz,i,m,-c);
if i==1,
hold on;
end;
end;
x=distribute(x,n,m,l,2);
y=distribute(y,n,m,l,2);
z=distribute(z,n,m,l,2);
nr=round(max(size(x))/m);
[xx,yy,zz]=arrange_shape(x,y,z,m);
c=reshape(c,nr,m);
for i=1:m,
surf_plot(xx,yy,zz,i,l,-c);
if i==1,
hold on;
end;
end;
x=distribute(x,n,m,l,3);
y=distribute(y,n,m,l,3);
z=distribute(z,n,m,l,3);
nr=round(max(size(x))/l);
[xx,yy,zz]=arrange_shape(x,y,z,l);
c=reshape(c,nr,l);
for i=1:l,
surf_plot(xx,yy,zz,i,n,-c);
if i==1,
hold on;
end;
end;
```

## Program : time_tplots.m

```
% This is a function which saves time Vs temperature details
% and plots the information for different grid points as specified in the input
% files.
% The saved data is in the form of a matrices containing gridpoint
% information (mats) and also the time Vs variable information (Mat).
function h = time_plots(t,c,n,m,l)
global tfinal tdisp imagp Tt time Tin Xin x y z;
if t<1.0e-8,
set(gca,'NextPlot','add');
end;
if size(imagp,1) > 1,
for s=1:size(imagp,1),
d(s)=locate(c,imagp(s,1),imagp(s,2),imagp(s,3),n,m,l);
%plot(t,d(s),'o');
end;
else
d=locate(c,imagp(1),imagp(2),imagp(3),n,m,l);
%plot(t,d,'o');
end;
Tt=[Tt;d];
time=[time;t];
plot(time,Tt);
if abs(t-tfinal)<=1.0e-6,
if size(imagp,1) > 1,
for s=1:size(imagp,1),
xd(s)=locate(x,imagp(s,1),imagp(s,2),imagp(s,3),n,m,l);
yd(s)=locate(y,imagp(s,1),imagp(s,2),imagp(s,3),n,m,l);
zd(s)=locate(z,imagp(s,1),imagp(s,2),imagp(s,3),n,m,l);
end;
else
xd=locate(x,imagp(1),imagp(2),imagp(3),n,m,l);
yd=locate(y,imagp(1),imagp(2),imagp(3),n,m,l);
zd=locate(z,imagp(1),imagp(2),imagp(3),n,m,l);
end;
mats=[xd' yd' zd'];
Mat=[time Tt];
save pltT.mat mats Mat;
end;
```

## Program time_xplots.m

```
% This is a function which saves time Vs cure details
% and plots the information for different grid points as specified in the input
% files.
% The saved data is in the form of a matrices containing gridpoint
% information (mats) and also the time Vs variable information (Matx).
function h = time_plots(t,c,n,m,l)
global tfinal tdisp imagp Tt Xt time Tin Xin x y z;
if t<1.0e-8,
set(gca,'NextPlot','add');
end;
if size(imagp,1) > 1,
for s=1:size(imagp,1),
d(s)=locate(c,imagp(s,1),imagp(s,2),imagp(s,3),n,m,l);
%plot(t,d(s),'o');
end;
else
d=locate(c,imagp(1),imagp(2),imagp(3),n,m,l);
%plot(t,d,'o');
end;
Xt=[Xt;d];
time=[time;t];
plot(time,Xt);
if abs(t-tfinal)<=1.0e-6,
if size(imagp,1) > 1,
for s=1:size(imagp,1),
xd(s)=locate(x,imagp(s,1),imagp(s,2),imagp(s,3),n,m,l);
yd(s)=locate(y,imagp(s,1),imagp(s,2),imagp(s,3),n,m,l);
zd(s)=locate(z,imagp(s,1),imagp(s,2),imagp(s,3),n,m,l);
end;
else
xd=locate(x,imagp(1),imagp(2),imagp(3),n,m,l);
yd=locate(y,imagp(1),imagp(2),imagp(3),n,m,l);
zd=locate(z,imagp(1),imagp(2),imagp(3),n,m,l);
end;
mats=[xd' yd' zd'];
Mat=[time Xt];
save pltX.mat matx Matx;
end;
```

# APPENDIX D

## C CODE FOR SIMULATION

**Program : mesh_generation.c**

This is the main program which calculates the mesh for a given complex body by using various input files as indicated and also passes the results of the mesh generation procedure on to the profile simulator which calculates the temperature and cure profiles.

```
#include<stdio.h>
#include<math.h>
#include<string.h>
#include"grid.h"
#include"engine.h"
char gra[5]; /* Choice for graphical output */
char plot_choice[10]; /* Temperature/Cure/Both profiles */
struct varray_xyz head; /* XYZ grid structure for multigrid method for shape generation */
Engine *eptr; /* Matlab Workspace pointer */
char map[5],ttx[5]; /* Options for map plots and time-temperature-cure plots */
int N,M,K;  /* No of mesh points along Psi,Eta and Zei directions respectively */
int n0,n1,n2; /* Multigrid iteration parameters for shape generation */
/* The following are different files of input and output for data transfer
inp_f --- Shape and coordinates input  */
FILE *inp_f,*new_f,*sc_f,*bc_f,*r1_f,*r2_f,*r3_f;
FILE
*numal_input,*proc_inp,*prop_f,*disp_f,*mat_lab;
struct varray top; /* Head list for varray */
double scale_x,scale_y,scale_z; /* Scaling parameters for x,y and z coordinates for the shape to be modelled */
double ***transfer_coeff; /* Heat transfer coefficient matrix */
double **thermal_cycle_data; /* Thermal cycle data matrix */
double T_initial,X_initial; /* Initial temperature and cure */
double Cp_f,Cp_m,rho_f,rho_m,kf,km; /* Specific heat, density and thermal conductivity parameters for property data */
double vol_f,delh; /* Volume fraction and heat of rxn respectively */
double k1o,k2o,k3o,E1,E2,E3,X_gel,B; /* Rxn rate equation parameters */
int no,nt0,nt1,nt2; /* Iteration parameters for multigrid method */
int psi,eta,zei; /* Display coordinates for graphical output (natural) */
int bound_condition; /* Nature of boundary condition
1. Dirichlet
 2. Mixed
 */
double final_time,t_display,delt; /* Final time for simulation, time of display and step time */
main()
{
extern char gra[5];
extern int N,M,K;
extern double scale_x,scale_y,scale_z;
struct point *grid_pt,*set_grid();
double ***mem_alloc_3D();
double ***x,***y,***z;
char mat_file[20];
int i,ch;
/* Open input file to read in coordinates of surfaces or edges */
inp_f=fopen("COORD.DAT","r");
fscanf(inp_f,"%s",gra);
fscanf(inp_f,"%d",&ch);
fscanf(inp_f,"%d",&N);
fscanf(inp_f,"%d",&M);
fscanf(inp_f,"%d",&K);
fscanf(inp_f,"%lf",&scale_x);
fscanf(inp_f,"%lf",&scale_y);
fscanf(inp_f,"%lf",&scale_z);
fscanf(inp_f,"%d",&n0);
fscanf(inp_f,"%d",&n1);
fscanf(inp_f,"%d",&n2);
fscanf(inp_f,"%s",mat_file);
mat_lab=fopen(mat_file,"w");
/* Allocate memory for x,y,z values */
x=mem_alloc_3D(N,M,K);
y=mem_alloc_3D(N,M,K);
z=mem_alloc_3D(N,M,K);
read_input(ch,x,y,z);
fclose(inp_f);
/* bc_f=fopen("check.m","w");
print_input(bc_f,x,y,z,N,M,K);
fclose(bc_f); */
/* If display is required open matlab workspace */
if(strcmp(gra,"yes")==0)
eptr=engOpen();
surface_approx(ch,x,N,M,K);
surface_approx(ch,y,N,M,K);
surface_approx(ch,z,N,M,K);
/* Set appropriate structures for multigrd method for mesh generation */
Struct_set(x,y,z);
/* new_f=fopen("Out.m","w");
print_out(new_f,head.x,head.y,head.z,N,M,K);
fclose(new_f);  */
if(strcmp(gra,"yes")==0)
{
plot_surf(1,x,y,z,(double ***)NULL,N,M,K);
view_options(x,y,z,(double ***)NULL,N,M,K,1);
print_plot_options();
}
```

```
FAS_xyz(&head);
sc_f=fopen("O.m","w");
print_out(sc_f,head.x,head.y,head.z,N,M,K);
fclose(sc_f);
if(strcmp(gra,"yes")==0)
{
plot_surf(2,x,y,z,(double ***)NULL,N,M,K);
view_options(x,y,z,(double ***)NULL,N,M,K,2);
print_plot_options();
}
/* Set structures to pass shape information to profile
generation program */
grid_pt=set_grid(N,M,K);
update_grid(grid_pt,x,y,z,N,M,K);
free_list();
printf("Mesh generation complete. Beginning thermal
simulation. \n");
/* Call profile simulation */
profile_gen(grid_pt,N,M,K);
fclose(mat_lab);
/* If Matlab workspace is open close it and shut down
*/
if(strcmp(gra,"yes")==0)
engClose(eptr);
}
```

**Program : multigrid.c**

This program contains routines for all the different operators for the multigrid method as well as the relaxation procedure for three dimensional elliptic generation difference equations with dirichlet boundary conditions for mesh generation.

```
#include<stdio.h>
#include<math.h>
#include<malloc.h>
#include "grid.h"
/* Restriction operator for multigrid algorithm */
Restrict(fp,f,n,m,l)
double ***fp,***f; /* fp : coarse grid matrix of
dependent variabes
      f : fine grid matrix of dependent variables */
int n,m,l; /* Mesh size */
{
double term1,term2,term3,term4;
int p,q,r;
int i,j,k;
p=(n+1)/2;
q=(m+1)/2;
r=(l+1)/2;
for(i=0;i<p;i++)
{
for(j=0;j<q;j++)
{
for(k=0;k<r;k++)
{
if((i==0)||(i==n-1)||(j==0)||(j==m-
1)||(k==0)||(k==l-1))
fp[i][j][k]=f[2*i][2*j][2*k];
else
{
term1=f[2*i][2*j][2*k];
term2=f[2*i-1][2*j][2*k]+f[2*i+1][2*j][2*k]+f[2*i][2*j-
1][2*k]+f[2*i][2*j+1][2*k]+
   f[2*i][2*j][2*k-1]+f[2*i][2*j][2*k+1];
term3=f[2*i-1][2*j-1][2*k]+f[2*i-
1][2*j+1][2*k]+f[2*i+1][2*j-
1][2*k]+f[2*i+1][2*j+1][2*k]+
   f[2*i-1][2*j][2*k-1]+f[2*i-
1][2*j][2*k+1]+f[2*i+1][2*j][2*k-
1]+f[2*i+1][2*j][2*k+1]+
   f[2*i][2*j-1][2*k-1]+f[2*i][2*j-
1][2*k+1]+f[2*i][2*j+1][2*k-1]+f[2*i][2*j+1][2*k+1];
term4=f[2*i-1][2*j-1][2*k-1]+f[2*i-1][2*j-
1][2*k+1]+f[2*i-1][2*j+1][2*k-1]+f[2*i-
1][2*j+1][2*k+1]+
   f[2*i+1][2*j-1][2*k-1]+f[2*i+1][2*j-
1][2*k+1]+f[2*i+1][2*j+1][2*k-
1]+f[2*i+1][2*j+1][2*k+1];

fp[i][j][k]=1.0/64.0*(8.0*term1+4.0*term2+2.0*term3
+term4);
}
}
}
}
return;
}
```

```
/* Interpolation operator to calculate values from a
coarse grid into a fine grid (Used to correct fine grid
values in mu
ltigrid algorithm) */
Interpolate(fin,f,n,m,l)
double ***fin,***f;
int n,m,l;
{
int i,j,k;
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
fin[2*i][2*j][2*k]=f[i][j][k];
if((i==n-1)||(j==m-1)||(k==l-1))
{
if((i==n-1)&&(j!=m-1)&&(k!=l-1))
{
fin[2*i][2*j+1][2*k]=0.50*(f[i][j][k]+f[i][j+1][k]);
fin[2*i][2*j][2*k+1]=0.50*(f[i][j][k]+f[i][j][k+1]);

fin[2*i][2*j+1][2*k+1]=0.25*(f[i][j][k]+f[i][j+1][k]+f[i][j
][k+1]+f[i][j+1][k+1]);
}
if((i!=n-1)&&(j==m-1)&&(k!=l-1))
{

fin[2*i+1][2*j][2*k+1]=0.25*(f[i][j][k]+f[i+1][j][k]+f[i][j
][k+1]+f[i+1][j][k+1]);
fin[2*i][2*j][2*k+1]=0.50*(f[i][j][k]+f[i][j][k+1]);
fin[2*i+1][2*j][2*k]=0.50*(f[i][j][k]+f[i+1][j][k]);
}
if((i!=n-1)&&(j!=m-1)&&(k==l-1))
```

```
 {
 fin[2*i+1][2*j][2*k]=0.50*(f[i][j][k]+f[i+1][j][k]);
 fin[2*i][2*j+1][2*k]=0.50*(f[i][j][k]+f[i][j+1][k]);

fin[2*i+1][2*j+1][2*k]=0.25*(f[i][j][k]+f[i+1][j][k]+f[i][j
+1][k]+f[i+1][j+1][k]);
 }
 if((i!=n-1)&&(j==m-1)&&(k==l-1))
 fin[2*i+1][2*j][2*k]=0.50*(f[i][j][k]+f[i+1][j][k]);
 if((i==n-1)&&(j==m-1)&&(k!=l-1))
 fin[2*i][2*j][2*k+1]=0.50*(f[i][j][k]+f[i][j][k+1]);
 if((i==n-1)&&(j!=m-1)&&(k==l-1))
 fin[2*i][2*j+1][2*k]=0.50*(f[i][j][k]+f[i][j+1][k]);
 }
 else
 {
 fin[2*i+1][2*j][2*k]=0.50*(f[i][j][k]+f[i+1][j][k]);
 fin[2*i][2*j+1][2*k]=0.50*(f[i][j][k]+f[i][j+1][k]);
 fin[2*i][2*j][2*k+1]=0.50*(f[i][j][k]+f[i][j][k+1]);

fin[2*i+1][2*j+1][2*k]=0.25*(f[i][j][k]+f[i+1][j][k]+f[i][j
+1][k]+f[i+1][j+1][k]);

fin[2*i+1][2*j][2*k+1]=0.25*(f[i][j][k]+f[i+1][j][k]+f[i][j
][k+1]+f[i+1][j][k+1]);

fin[2*i][2*j+1][2*k+1]=0.25*(f[i][j][k]+f[i][j+1][k]+f[i][j
][k+1]+f[i][j+1][k+1]);

fin[2*i+1][2*j+1][2*k+1]=0.125*(f[i][j][k]+f[i+1][j][k]+f
[i][j+1][k]+f[i+1][j+1][k]+

f[i][j][k+1]+f[i+1][j][k+1]+f[i][j+1][k+1]+f[i+1][j+1][k+1
]);
 }
 }
 }
}
return;
}


/* Setting strctures for multigrid algorithm for mesh
generation */
Struct_set(x,y,z)
double ***x,***y,***z;
{
extern int N,M,K;
extern struct varray_xyz head;
struct varray_xyz *ptr,*temp;
double ***mem_alloc_3D();
int i,j,k;
head.n=N;
head.m=M;
head.l=K;
head.next=NULL;
head.x=x;
head.y=y;
head.z=z;
head.Sx=mem_alloc_3D(N,M,K);
head.Sy=mem_alloc_3D(N,M,K);
head.Sz=mem_alloc_3D(N,M,K);
set_mat_zero(head.Sx,N,M,K);
set_mat_zero(head.Sy,N,M,K);
set_mat_zero(head.Sz,N,M,K);
ptr=&head;
while((ptr->n>3)&&(ptr->m>3)&&(ptr->l>3))
{
temp=(struct varray_xyz *)malloc(sizeof(struct
varray_xyz));
temp->n=(int)(ptr->n+1)/2;
temp->m=(int)(ptr->m+1)/2;
temp->l=(int)(ptr->l+1)/2;
temp->x=mem_alloc_3D(temp->n,temp->m,temp->l);
temp->y=mem_alloc_3D(temp->n,temp->m,temp->l);
temp->z=mem_alloc_3D(temp->n,temp->m,temp->l);
temp->Sx=mem_alloc_3D(temp->n,temp->m,temp-
>l);
temp->Sy=mem_alloc_3D(temp->n,temp->m,temp-
>l);
temp->Sz=mem_alloc_3D(temp->n,temp->m,temp-
>l);
temp->next=NULL;
ptr->next=temp;
ptr=ptr->next;
}
return;
}


/* Fully multigrid V algorithm for mesh generation */
/* Details of the routine are in FMV algorithm of the
program temp_grid.c */
FMV_xyz(v)
struct varray_xyz *v;
{
extern int n1,n2;
int i,j,k;
struct varray_xyz *temp;
double
***fx,***fy,***fz,***f1x,***f1y,***f1z,***f2x,***f2y,*
**f2z;
double ***rx,***ry,***rz,trunc,err,norm_xyz();
double ***tx,***ty,***tz,***vx,***vy,***vz;
int vn,vm,vl,tn,tm,tl;
double ***mem_alloc_3D();
/* printf("FMV_xyz begins here\n"); */
vn=v->n;
vm=v->m;
vl=v->l;
vx=v->x;
vy=v->y;
vz=v->z;
if(v->next==NULL)
{
/* printf("FMV_xyz : Coarsest Grid \n"); */
Relax_xyz(vx,vy,vz,v->Sx,v->Sy,v-
>Sz,vn,vm,vl,n1+n2);
}
else
{
Relax_xyz(vx,vy,vz,v->Sx,v->Sy,v->Sz,vn,vm,vl,n1);
temp=v->next;
tn=temp->n;
tm=temp->m;
tl=temp->l;
tx=temp->x;
ty=temp->y;
tz=temp->z;
```

```
f1x=mem_alloc_3D(tn,tm,tl);
f1y=mem_alloc_3D(tn,tm,tl);
f1z=mem_alloc_3D(tn,tm,tl);
fx=mem_alloc_3D(tn,tm,tl);
fy=mem_alloc_3D(tn,tm,tl);
fz=mem_alloc_3D(tn,tm,tl);
rx=mem_alloc_3D(tn,tm,tl);
ry=mem_alloc_3D(tn,tm,tl);
rz=mem_alloc_3D(tn,tm,tl);
f2x=mem_alloc_3D(vn,vm,vl);
f2y=mem_alloc_3D(vn,vm,vl);
f2z=mem_alloc_3D(vn,vm,vl);
Restrict(rx,vx,tn,tm,tl);
Restrict(ry,vy,tn,tm,tl);
Restrict(rz,vz,tn,tm,tl);
A_xyz(fx,fy,fz,rx,ry,rz,tn,tm,tl);
A_xyz(f2x,f2y,f2z,vx,vy,vz,vn,vm,vl);
Restrict(f1x,f2x,tn,tm,tl);
Restrict(f1y,f2y,tn,tm,tl);
Restrict(f1z,f2z,tn,tm,tl);
mat_sub(fx,fx,f1x,tn,tm,tl);
mat_sub(fy,fy,f1y,tn,tm,tl);
mat_sub(fz,fz,f1z,tn,tm,tl);
v->trunc=norm_xyz(fx,fy,fz,tn,tm,tl);
Restrict(tx,vx,tn,tm,tl);
Restrict(ty,vy,tn,tm,tl);
Restrict(tz,vz,tn,tm,tl);
mat_copy(rx,tx,tn,tm,tl);
mat_copy(ry,ty,tn,tm,tl);
mat_copy(rz,tz,tn,tm,tl);
A_xyz(f1x,f1y,f1z,tx,ty,tz,tn,tl,tn);
A_xyz(f2x,f2y,f2z,vx,vy,vz,vn,vm,vl);
mat_sub(f2x,f2x,v->Sx,vn,vm,vl);
mat_sub(f2y,f2y,v->Sy,vn,vm,vl);
mat_sub(f2z,f2z,v->Sy,vn,vm,vl);
Restrict(fx,f2x,tn,tm,tl);
Restrict(fy,f2y,tn,tm,tl);
Restrict(fz,f2z,tn,tm,tl);
mat_sub(temp->Sx,f1x,fx,tn,tm,tl);
mat_sub(temp->Sy,f1y,fy,tn,tm,tl);
mat_sub(temp->Sz,f1z,fz,tn,tm,tl);
for(i=0;i<=1;i++)
FMV_xyz(temp);
mat_sub(f1x,tx,rx,tn,tm,tl);
mat_sub(f1y,ty,ry,tn,tm,tl);
mat_sub(f1z,tz,rz,tn,tm,tl);
Interpolate(f2x,f1x,tn,tm,tl);
Interpolate(f2y,f1y,tn,tm,tl);
Interpolate(f2z,f1z,tn,tm,tl);
mat_add(vx,f2x,vx,vn,vm,vl);
mat_add(vy,f2y,vy,vn,vm,vl);
mat_add(vz,f2z,vz,vn,vm,vl);
Relax_xyz(vx,vy,vz,v->Sx,v->Sy,v->Sz,vn,vm,vl,n2);
free_3D(rx,tn,tm,tl);
free_3D(ry,tn,tm,tl);
free_3D(rz,tn,tm,tl);
free_3D(fx,tn,tm,tl);
free_3D(fy,tn,tm,tl);
free_3D(fz,tn,tm,tl);
free_3D(f1x,tn,tm,tl);
free_3D(f1y,tn,tm,tl);
free_3D(f1z,tn,tm,tl);
free_3D(f2x,vn,vm,vl);
free_3D(f2y,vn,vm,vl);
free_3D(f2z,vn,vm,vl);
}
/* printf("FMV_xyz (N = %d) ends here\n",vn); */
return;
}


/* FAS algorithm for mesh generation */
/* Details of the different steps are given in FAS
routine in the program 'temp_grid.c' */
FAS_xyz(v)
struct varray_xyz *v;
{
extern int n0;
int i,j,k,cnt;
struct varray_xyz *temp;
double g;
double
***fx,***fy,***fz,***f1x,***f1y,***f1z,***f2x,***f2y,*
**f2z;
double ***rx,***ry,***rz,err,trunc,norm_xyz();
double ***mem_alloc_3D();
double ***tx,***ty,***tz,***vx,***vy,***vz;
int vn,vm,vl,tn,tm,tl;
vn=v->n;
vm=v->m;
vl=v->l;
vx=v->x;
vy=v->y;
vz=v->z;
if(v->next==NULL)
{
/* printf("FAS_xyz : Coarsest Grid \n"); */
for(i=1;i<=n0;i++)
FMV_xyz(v);
}
else
{
/* printf("FAS_xyz (N = %d) begins here \n",vn); */
temp=v->next;
tn=temp->n;
tm=temp->m;
tl=temp->l;
tx=temp->x;
ty=temp->y;
tz=temp->z;
Restrict(tx,vx,tn,tm,tl);
Restrict(ty,vy,tn,tm,tl);
Restrict(tz,vz,tn,tm,tl);
f1x=mem_alloc_3D(tn,tm,tl);
f1y=mem_alloc_3D(tn,tm,tl);
f1z=mem_alloc_3D(tn,tm,tl);
fx=mem_alloc_3D(tn,tm,tl);
fy=mem_alloc_3D(tn,tm,tl);
fz=mem_alloc_3D(tn,tm,tl);
rx=mem_alloc_3D(tn,tm,tl);
ry=mem_alloc_3D(tn,tm,tl);
rz=mem_alloc_3D(tn,tm,tl);
f2x=mem_alloc_3D(vn,vm,vl);
f2y=mem_alloc_3D(vn,vm,vl);
f2z=mem_alloc_3D(vn,vm,vl);
mat_copy(rx,tx,tn,tm,tl);
mat_copy(ry,ty,tn,tm,tl);
mat_copy(rz,tz,tn,tm,tl);
A_xyz(f1x,f1y,f1z,tx,ty,tz,tn,tm,tl);
A_xyz(f2x,f2y,f2z,vx,vy,vz,vn,vl,vn);
```

```
mat_sub(f2x,f2x,v->Sx,vn,vm,vl);
mat_sub(f2y,f2y,v->Sy,vn,vm,vl);
mat_sub(f2z,f2z,v->Sz,vn,vm,vl);
Restrict(fx,f2x,tn,tm,tl);
Restrict(fy,f2y,tn,tm,tl);
Restrict(fz,f2z,tn,tm,tl);
mat_sub(temp->Sx,f1x,fx,tn,tm,tl);
mat_sub(temp->Sy,f1y,fy,tn,tm,tl);
mat_sub(temp->Sz,f1z,fz,tn,tm,tl);
FAS_xyz(temp);
mat_sub(f1x,tx,rx,tn,tm,tl);
mat_sub(f1y,ty,ry,tn,tm,tl);
mat_sub(f1z,tz,rz,tn,tm,tl);
Interpolate(f2x,f1x,tn,tm,tl);
Interpolate(f2y,f1y,tn,tm,tl);
Interpolate(f2z,f1z,tn,tm,tl);
mat_add(vx,f2x,vx,vn,vm,vl);
mat_add(vy,f2y,vy,vn,vm,vl);
mat_add(vz,f2z,vz,vn,vm,vl);
/*if(vn==9)
print_out(vx,vy,vz,vn,vm,vl); */
err=1;
cnt=1;
g=0;
while(cnt<=n0)/* &&(err>=0.333*v-
>trunc)&&(fabs((g-err/v->trunc)/g)>1.0e-4))*/
{
if(cnt!=1)
g=err/v->trunc;
else
g=1;
FMV_xyz(v);
A_xyz(f2x,f2y,f2z,vx,vy,vz,vn,vm,vl);
mat_sub(f2x,v->Sx,f2x,vn,vm,vl);
mat_sub(f2y,v->Sy,f2y,vn,vm,vl);
mat_sub(f2z,v->Sz,f2z,vn,vm,vl);
err=norm_xyz(f2x,f2y,f2z,vn,vm,vl);
/* printf("FAS_xyz (N = %d): Error (Old) : %6.4lf
Error (New) : %6.4lf \n",vn,g,err/v->trunc); */
   cnt++;
}
free_3D(rx,tn,tm,tl);
free_3D(ry,tn,tm,tl);
free_3D(rz,tn,tm,tl);
free_3D(fx,tn,tm,tl);
free_3D(fy,tn,tm,tl);
free_3D(fz,tn,tm,tl);
free_3D(f1x,tn,tm,tl);
free_3D(f1y,tn,tm,tl);
free_3D(f1z,tn,tm,tl);
free_3D(f2x,vn,vm,vl);
free_3D(f2y,vn,vm,vl);
free_3D(f2z,vn,vm,vl);
}
/* printf("FAS_xyz (N = %d) ends here \n",vn);  */
return;
}
```

**Program : shape_guess.c**

This program calculates the first approximation to the shape of the body using transfinite interpolation in two and three dimensions.

```
#include<stdio.h>
#include<math.h>
#include<malloc.h>
#include "grid.h"
/* Transfinite Interpolation Routine */
transfinite_3D(f,n,m,l)
double ***f;
int n,m,l;
{
double ***F1,***mem_alloc_3D(),int_plt();
double ***F2;
double ***F3;
int i,j,k;
F1=mem_alloc_3D(l,n,m);
/* Correction along j(eta) direction */
for(k=0;k<l;k++)
{
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
F1[k][i][j]=int_plt(k,l,f[i][j][0],f[i][j][l-1]);
}
}
F2=mem_alloc_3D(m,l,n);
/* Correction along i(psi) direction */
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
for(i=0;i<n;i++)
F2[j][k][i]=int_plt(j,m,f[i][0][k]-F1[k][i][0],f[i][m-1][k]-
F1[k][i][m-1]);
}
}
F3=mem_alloc_3D(n,m,l);
/* Correction along k(zei) direction */
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
F3[i][j][k]=int_plt(i,n,f[0][j][k]-F1[k][0][j]-
F2[j][k][0],f[n-1][j][k]-F1[k][n-1][j]-F2[j][k][n-1]);
}
}
/* Algebraic grid is sum of all corrections  */
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
f[i][j][k]=F1[k][i][j]+F2[j][k][i]+F3[i][j][k];
}
}
free_3D(F1,l,n,m);
free_3D(F2,m,l,n);
free_3D(F3,n,m,l);
return;
}


/* Cubic spline interpolation (will be linear if only two
end points are given) */
double int_plt(i,n,r1,r2) /* Cubic Spline (Linear if only
6 surfaces are specified in all )
 interpolation */
```

```
int i,n;
double r1,r2;
{
double ret;
/* if((i==0)||(i==n-1))   */
ret=r1*(n-1-i)/(double)(n-1)+r2*i/(double)(n-1);
/* else
ret=2.3*((n-1-i)*r1+i*r2)/((double)(i*(n-1)));  */
return(ret);
}



/* Transfinite interpolation along two dimensions .
Will be used if only edge information is given */
transfinite_2D(f,n,m)
double **f;
int n,m;
{
double **f1,**f2,**mem_alloc_2D(),int_plt();
int i,j;
f1=mem_alloc_2D(n,m);
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f1[i][j]=int_plt(i,n,f[0][j],f[n-1][j]);
}
f2=mem_alloc_2D(m,n);
for(j=0;j<m;j++)
{
for(i=0;i<n;i++)
f2[j][i]=int_plt(j,m,f[i][0]-f1[i][0],f[i][m-1]-f1[i][m-1]);
}
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f[i][j]=f1[i][j]+f2[j][i];
}
free_2D(f1,n,m);
free_2D(f2,m,n);
return;
}



/* Approximation of surfaces by successive calling of
transfinite two dimensional routines */
surface_approx(ch,f,n,m,l)
int ch;
double ***f;
int n,m,l;
{
int i,j,k,d;
double **two_d,**mem_alloc_2D();
d=0;
if(ch==1)
{
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
if(((i==0)&&(j==0))||((i==0)&&(j==m-1))||((i==n-
1)&&(j==0))||((i==n-1)&&(j==m-
1))||((i==0)&&(k==0))||((i==0)&&(k==l-1))||((i==n-
1)&&(k==0))||((i==n-1)&&(k==l-
1))||((j==0)&&(k==0))||((j==m-
1)&&(k==0))||((j==0)&&(k==l-1))||((j==m-
1)&&(k==l-1)))
d++;
else
f[i][j][k]=0.00;
}
}
}
for(i=0;i<3;i++)
{
switch(i)
{
case 0: {
two_d=mem_alloc_2D(m,l);
for(j=0;j<n;j=j+n-1)
{
D3_to_D2(1,f,two_d,j,m,l);
transfinite_2D(two_d,m,l);
D2_to_D3(1,two_d,f,j,m,l);
}
free_2D(two_d,m,l);
break;
}
case 1: {
two_d=mem_alloc_2D(n,l);
for(j=0;j<m;j=j+m-1)
{
D3_to_D2(2,f,two_d,j,n,l);
transfinite_2D(two_d,n,l);
D2_to_D3(2,two_d,f,j,n,l);
}
free_2D(two_d,n,l);
break;
}
case 2: {
two_d=mem_alloc_2D(n,m);
for(j=0;j<l;j=j+l-1)
{
D3_to_D2(3,f,two_d,j,n,m);
transfinite_2D(two_d,n,m);
D2_to_D3(3,two_d,f,j,n,m);
}
free_2D(two_d,n,m);
break;
}
}
}
}
transfinite_3D(f,n,m,l);
return;
}



/* Convert a three dimensional matrix into two
dimensional matrix containing only surface
information*/
D3_to_D2(c,f1,f2,p,n,m)
int c;
double ***f1,**f2;
int p,n,m;
{
int i,j;
switch(c)
```

```
{
case 1: {
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f2[i][j]=f1[p][i][j];
}
break;
}
case 2: {
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f2[i][j]=f1[i][p][j];
}
break;
}
case 3: {
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f2[i][j]=f1[i][j][p];
}
break;
}
}
return;
}


/* Reading back surface information into 3D matrix */
D2_to_D3(c,f1,f2,p,n,m)
int c;
double **f1,***f2;
int p,n,m;
{
int i,j;
switch(c)
{
case 1: {
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f2[p][i][j]=f1[i][j];
}
break;
}
case 2: {
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f2[i][p][j]=f1[i][j];
}
break;
}
case 3: {
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f2[i][j][p]=f1[i][j];
}
break;
}
}
return;
}
```

**Program: shape_solve.c**

This program calculates the various parameters associated with the elliptic generation system of equations such as alphas and betas as well as the jacobians at every grid point in the domain.

```
#include<stdio.h>
#include<math.h>
#include<malloc.h>
#include "grid.h"
int cyclic(i) /* Routine which returns the cyclic value
of i in (1,2,3) */
int i;
{
int cyl;
switch(i)
{
case 0: {
cyl=1;
break;
}
case 1: {
cyl=2;
break;
}
case 2: {
cyl=0;
break;
}
}
return(cyl);
}



set_matrix(X,Y,Z,n,m,l,i,j,k,mat) /* Sets the jacobian
matrix at any grid point in
 the transformed coordinate */
double ***X,***Y,***Z;
int n,m,l,i,j,k;
double **mat;
{
double f_psi(),f_eta(),f_zei();
double ***temp_ptr,sum;
int p,q;
for(p=0;p<3;p++)
{
switch(p)
{
case 0: {
temp_ptr=X;
break;
}
case 1: {
temp_ptr=Y;
break;
}
case 2: {
temp_ptr=Z;
break;
```

```
}
}
for(q=0;q<3;q++)
{
switch(q)
{
case 0: {
mat[p][q]=f_psi(temp_ptr,i,j,k,n,m,l,1);
break;
}
case 1: {
mat[p][q]=f_eta(temp_ptr,i,j,k,n,m,l,1);
break;
}
case 2: {
mat[p][q]=f_zei(temp_ptr,i,j,k,n,m,l,1);
break;
}
}
}
}
/* for(p=0;p<3;p++)
{
sum=0;
for(q=0;q<3;q++)
sum=sum+pow(fabs(mat[p][q]),2.0);
for(q=0;q<3;q++)
mat[p][q]=mat[p][q]/sqrt(sum);
} */
return;
}



calc_beta(Mat,b) /* Calculate the beta matrix using
'M' matrix values at a given gridpoint */
double **Mat,**b; /* Mat - Matrix 'M' (for the
transformation) ,
    b - Returned matrix of betas */
{
int i,j,k,l,m,n;
for(i=0;i<3;i++)
{
k=cyclic(i);
m=cyclic(k);
for(j=0;j<3;j++)
{
l=cyclic(j);
n=cyclic(l);
b[i][j]=Mat[k][l]*Mat[m][n]-Mat[k][n]*Mat[m][l];
}
}
return;
}



double calc_alpha_jac(a,X,Y,Z,n,m,l,i,j,k) /*
Calculation of alpha values using beta matrix */
double **a,***X,***Y,***Z;/* X,Y,Z - x,y,z arrays; i,j,k
- location; a - returned matrix of alphas */
int n,m,l,i,j,k;
{
double sum,**mat,**b;
double jac,Jacobian();
int o,p,q;
mat=(double **)malloc(3*sizeof(double *));
b=(double **)malloc(3*sizeof(double *));
for(o=0;o<3;o++)
{
mat[o]=(double *)malloc(3*sizeof(double));
b[o]=(double *)malloc(3*sizeof(double));
}
set_matrix(X,Y,Z,n,m,l,i,j,k,mat);
calc_beta(mat,b);
jac=Jacobian(mat);
for(o=0;o<3;o++)
{
for(p=0;p<3;p++)
{
sum=0.0;
for(q=0;q<3;q++)
sum=sum+b[q][o]*b[q][p];
a[o][p]=sum;
}
}
free_2D(mat,3,3);
free_2D(b,3,3);
return(jac);
}



double Jacobian(mat) /* Calculates Jacobian(J) given
the transformation matrix (M) J=|M| */
double **mat; /* mat - Transformation matrix (M) */
{
double ret;
ret=mat[0][0]*(mat[1][1]*mat[2][2]-
mat[1][2]*mat[2][1])-mat[0][1]*(mat[1][0]*mat[2][2]-
mat[1][2]*mat[2][0])+mat[0][2]*(mat[1][0]*mat[2][1]-
mat[1][1]*mat[2][0]);
return(ret);
}



/* Setting Boundary Conditions Over The Volume */
set_boundary(fx,fy,fz,fx1,fy1,fz1,n,m,l)
double ***fx,***fy,***fz;
double ***fx1,***fy1,***fz1;
int n,m,l;
{
int step_i,step_j,step_k,i,j,k,p;
for(p=1;p<=3;p++)
{
switch(p)
{
/* Along Psi Direction */
case 1: {
step_i=1;
step_j=1;
step_k=l-1;
break;
}
/* Along Eta Direction */
case 2: {
step_i=n-1;
step_j=1;
step_k=1;
```

```
break;
}
/* Along Zei Direction */
case 3: {
step_i=1;
step_j=m-1;
step_k=1;
break;
}
}
for(i=0;i<n;i=i+step_i)
{
for(j=0;j<m;j=j+step_j)
{
for(k=0;k<l;k=k+step_k)
{
fx1[i][j][k]=fx[i][j][k];
fy1[i][j][k]=fy[i][j][k];
fz1[i][j][k]=fz[i][j][k];
}
}
}
}
return;
}



/* Sets the grid_pt structure to make shape
information available from
 the mesh generation to the profile simulation
program */
struct point *set_grid(n,m,l)
int n,m,l;
{
int i,j,k,p;
struct point *grid_pt;
double ***mem_alloc_3D();
grid_pt=(struct point *)malloc(sizeof(struct point));
grid_pt->x=mem_alloc_3D(n,m,l);
grid_pt->y=mem_alloc_3D(n,m,l);
grid_pt->z=mem_alloc_3D(n,m,l);
grid_pt->J=mem_alloc_3D(n,m,l);
grid_pt->a=(struct D2_array
***)malloc(n*sizeof(struct D2_array **));
for(i=0;i<n;i++)
{
grid_pt->a[i]=(struct D2_array
**)malloc(m*sizeof(struct D2_array *));
for(j=0;j<m;j++)
{
grid_pt->a[i][j]=(struct D2_array
*)malloc(l*sizeof(struct D2_array));
for(k=0;k<l;k++)
{
grid_pt->a[i][j][k].c=(double **)malloc(3*sizeof(double
*));
for(p=0;p<3;p++)
grid_pt->a[i][j][k].c[p]=(double
*)malloc(3*sizeof(double));
}
}
}
return(grid_pt);
}



/* Updation of grid values given the coordinate values
at each grid po
int and the mesh size */
update_grid(grid_pt,fx,fy,fz,n,m,l)
struct point *grid_pt;
double ***fx,***fy,***fz;
int n,m,l;
{
int i,j,k,p,q;
double **an,norm(),jn,**A_norm();
mat_copy(grid_pt->x,fx,n,m,l);
mat_copy(grid_pt->y,fy,n,m,l);
mat_copy(grid_pt->z,fz,n,m,l);
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
grid_pt->J[i][j][k]=calc_alpha_jac(grid_pt-
>a[i][j][k].c,fx,fy,fz,n,m,l,i,j,k);
}
}
return;
}



double **A_norm(grid_pt,n,m,l)
struct point *grid_pt;
int n,m,l;
{
int i,j,k,p,q;
double
***temp,**a,***mem_alloc_3D(),**mem_alloc_2D();
a=mem_alloc_2D(3,3);
temp=mem_alloc_3D(n,m,l);
for(p=0;p<3;p++)
{
for(q=p;q<3;q++)
{
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
temp[i][j][k]=grid_pt->a[i][j][k].c[p][q];
}
}
a[p][q]=norm(temp,n,m,l);
}
}
free_3D(temp,n,m,l);
return(a);
}



/* The mesh generation operator evaluation for each
of the three ellip
tic equations involved in the generation */
A_xyz(outx,outy,outz,ux,uy,uz,n,m,l)
double ***outx,***outy,***outz;/* Output matrices
corresponding to the three equations */
```

```
double ***ux,***uy,***uz; /* Input values for the
three coordinates x,
y, z as a function of grid points in the natural
coordinates */
int n,m,l; /* Mesh size (required for multigrid
operation) */
{
int i,j,k;
double
fx1,fx2,fx3,fy1,fy2,fy3,fz1,fz2,fz3,jac,P(),Q(),R(),**al;
double
f_psi(),f_eta(),f_zei(),f_psi_psi(),f_zei_zei(),f_eta_eta(),f
_psi_eta();
double
f_eta_zei(),f_zei_psi(),calc_alpha_jac(),**mem_alloc_2
D();
set_boundary(ux,uy,uz,outx,outy,outz,n,m,l);
al=mem_alloc_2D(3,3);
for(i=1;i<n-1;i++)
{
for(j=1;j<m-1;j++)
{
for(k=1;k<l-1;k++)
{
jac=calc_alpha_jac(al,ux,uy,uz,n,m,l,i,j,k);
fx1=al[0][0]*f_psi_psi((double
***)NULL,ux,i,j,k,n)+al[1][1]*f_eta_eta((double
***)NULL,ux,i,j,k,m)+al[2][2]*f_zei_zei((double
***)NULL,ux,i,j,k,l);
fy1=al[0][0]*f_psi_psi((double
***)NULL,uy,i,j,k,n)+al[1][1]*f_eta_eta((double
***)NULL,uy,i,j,k,m)+al[2][2]*f_zei_zei((double
***)NULL,uy,i,j,k,l);
fz1=al[0][0]*f_psi_psi((double
***)NULL,uz,i,j,k,n)+al[1][1]*f_eta_eta((double
***)NULL,uz,i,j,k,m)+al[2][2]*f_zei_zei((double
***)NULL,uz,i,j,k,l);
fx2=2.0*(al[0][1]*f_psi_eta((double
***)NULL,ux,i,j,k,n,m)+al[1][2]*f_eta_zei((double
***)NULL,ux,i,j,k,m,l)+al[2][0]*f_zei_psi((double
***)NULL,ux,i,j,k,l,n));
fy2=2.0*(al[0][1]*f_psi_eta((double
***)NULL,uy,i,j,k,n,m)+al[1][2]*f_eta_zei((double
***)NULL,uy,i,j,k,m,l)+al[2][0]*f_zei_psi((double
***)NULL,uy,i,j,k,l,n));
fz2=2.0*(al[0][1]*f_psi_eta((double
***)NULL,uz,i,j,k,n,m)+al[1][2]*f_eta_zei((double
***)NULL,uz,i,j,k,m,l)+al[2][0]*f_zei_psi((double
***)NULL,uz,i,j,k,l,n));
fx3=pow(fabs(jac),2.0)*(P(i,j,k)*f_psi(ux,i,j,k,n,m,l,1)+
Q(i,j,k)*f_eta(ux,i,j,k,n,m,l,1)+R(i,j,k)*f_zei(ux,i,j,k,n,
m,l,1));
fy3=pow(fabs(jac),2.0)*(P(i,j,k)*f_psi(uy,i,j,k,n,m,l,1)+
Q(i,j,k)*f_eta(uy,i,j,k,n,m,l,1)+R(i,j,k)*f_zei(uy,i,j,k,n,
m,l,1));
fz3=pow(fabs(jac),2.0)*(P(i,j,k)*f_psi(uz,i,j,k,n,m,l,1)+
Q(i,j,k)*f_eta(uz,i,j,k,n,m,l,1)+R(i,j,k)*f_zei(uz,i,j,k,n,
m,l,1));
outx[i][j][k]=fx1+fx2+fx3;
outy[i][j][k]=fy1+fy2+fy3;
outz[i][j][k]=fz1+fz2+fz3;
}
}
}
free_2D(al,3,3);
return;
}



/* Relaxation iterations for mesh generation (Gauss
Siedel) */
Relax_xyz(ux,uy,uz,Sx,Sy,Sz,n,m,l,nu)
double ***ux,***uy,***uz;
double ***Sx,***Sy,***Sz;
int n,m,l,nu;
{
extern int N,M,K;
int i,j,k,limit;
double
error,errord,tem_err,norm_xyz(),***tempx_err,***te
mpy_err,***tempz_err;
double
**al,fx1,fx2,fx3,fy1,fy2,fy3,fz1,fz2,fz3,jac,P(),Q(),R(),**
*mem_alloc_3D();
double
f_psi(),f_eta(),f_zei(),f_psi_psi(),f_zei_zei(),f_eta_eta(),f
_psi_eta();
double
f_eta_zei(),f_zei_psi(),calc_alpha_jac(),h,g,r,mult,tol;
int begi,begj,begk,cnti,cntj,cntk,count;
/* printf("Relax Begins \n"); */
h=(double)(N-1)/(n-1);
g=(double)(M-1)/(m-1);
r=(double)(K-1)/(l-1);
al=mem_alloc_2D(3,3);
count=1;
errord=1.0;
limit=nu;
tol=1.0e-6;
if((n==3)||(m==3)||(l==3))
{
tempx_err=mem_alloc_3D(n,m,l);
tempy_err=mem_alloc_3D(n,m,l);
tempz_err=mem_alloc_3D(n,m,l);
errord=1.0;
}
else
errord=1.0e-8;
/* Gauss Siedel Iterative (Red-Black) Sweeps */
while((count<=limit)||(errord>tol))
{
if((n==3)||(m==3)||(l==3))
{
mat_copy(tempx_err,ux,n,m,l);
mat_copy(tempy_err,uy,n,m,l);
mat_copy(tempz_err,uz,n,m,l);
error=norm_xyz(ux,uy,uz,n,m,l);
}
cnti=1;
begi=2;
/* Red-Black Sweeps */
while(cnti<=2)
{
/* Psi direction */
for(i=begi;i<n-1;i=i+2)
{
cntj=1;
begj=2;
/* Red-Black Sweeps */
```

```
while(cntj<=2)
{
/* Eta direction */
for(j=begj;j<m-1;j=j+2)
{
cntk=1;
begk=2;
/* Red-Black Sweeps */
while(cntk<=2)
{
/* Zei direction */
for(k=begk;k<l-1;k=k+2)
{
jac=calc_alpha_jac(al,ux,uy,uz,n,m,l,i,j,k);
fx1=al[0][0]/pow(h,2.0)*(ux[i+1][j][k]+ux[i-
1][j][k])+al[1][1]/pow(g,2.0)*(ux[i][j+1][k]+ux[i][j-
1][k])+al[2][2]/pow(r,2.0)*(ux[i][j][k+1]+ux[i][j][k-1]);
fy1=al[0][0]/pow(h,2.0)*(uy[i+1][j][k]+uy[i-
1][j][k])+al[1][1]/pow(g,2.0)*(uy[i][j+1][k]+uy[i][j-
1][k])+al[2][2]/pow(r,2.0)*(uy[i][j][k+1]+uy[i][j][k-1]);
fz1=al[0][0]/pow(h,2.0)*(uz[i+1][j][k]+uz[i-
1][j][k])+al[1][1]/pow(g,2.0)*(uz[i][j+1][k]+uz[i][j-
1][k])+al[2][2]/pow(r,2.0)*(uz[i][j][k+1]+uz[i][j][k-1]);
fx2=2.0*(al[0][1]*f_psi_eta((double
***)NULL,ux,i,j,k,n,m)+al[1][2]*f_eta_zei((double
***)NULL,ux,i,j,k,m,l)+al[2][0]*f_zei_psi((double
***)NULL,ux,i,j,k,l,n));
fy2=2.0*(al[0][1]*f_psi_eta((double
***)NULL,uy,i,j,k,n,m)+al[1][2]*f_eta_zei((double
***)NULL,uy,i,j,k,m,l)+al[2][0]*f_zei_psi((double
***)NULL,uy,i,j,k,l,n));
fz2=2.0*(al[0][1]*f_psi_eta((double
***)NULL,uz,i,j,k,n,m)+al[1][2]*f_eta_zei((double
***)NULL,uz,i,j,k,m,l)+al[2][0]*f_zei_psi((double
***)NULL,uz,i,j,k,l,n));
fx3=pow(fabs(jac),2.0)*(P(i,j,k)*f_psi(ux,i,j,k,n,m,l,1)+
Q(i,j,k)*f_eta(ux,i,j,k,n,m,l,1)+R(i,j,k)*f_zei(ux,i,j,k,n,
m,l,1));
fy3=pow(fabs(jac),2.0)*(P(i,j,k)*f_psi(uy,i,j,k,n,m,l,1)+
Q(i,j,k)*f_eta(uy,i,j,k,n,m,l,1)+R(i,j,k)*f_zei(uy,i,j,k,n,
m,l,1));
fz3=pow(fabs(jac),2.0)*(P(i,j,k)*f_psi(uz,i,j,k,n,m,l,1)+
Q(i,j,k)*f_eta(uz,i,j,k,n,m,l,1)+R(i,j,k)*f_zei(uz,i,j,k,n,
m,l,1));
mult=0.50/(al[0][0]/pow(h,2.0)+al[1][1]/pow(g,2.0)+al[2
][2]/pow(r,2.0));
ux[i][j][k]=mult*(fx1+fx2+fx3-Sx[i][j][k]);
uy[i][j][k]=mult*(fy1+fy2+fy3-Sy[i][j][k]);
uz[i][j][k]=mult*(fz1+fz2+fz3-Sz[i][j][k]);
}
cntk++;
begk--;
}
}
cntj++;
begj--;
}
}
cnti++;
begi--;
}
if((n==3)||(m==3)||(l==3))
{
mat_sub(tempx_err,tempx_err,ux,n,m,l);
mat_sub(tempy_err,tempy_err,uy,n,m,l);
mat_sub(tempz_err,tempz_err,uz,n,m,l);
errord=norm_xyz(tempx_err,tempy_err,tempz_err,n,
m,l)/error;
}
count++;
}
free_2D(al,3,3);
if((n==3)||(m==3)||(l==3))
{
free_3D(tempx_err,n,m,l);
free_3D(tempy_err,n,m,l);
free_3D(tempz_err,n,m,l);
}
/* printf("Relax Done\n"); */
return;
}


/* These are control functions to be used for adaptive
meshing and these are currently zero */
/* Control Function - P(i,j,k) */
double P(i,j,k)
int i,j,k;
{
double ret,sum;
ret=0.00;
return(ret);
}



/* Control Function - Q(i,j,k) */
double Q(i,j,k)
int i,j,k;
{
double ret,sum;
ret=0.00;
return(ret);
}



/* Control Function - R(i,j,k) */
double R(i,j,k)
int i,j,k;
{
double ret,sum;
ret=0.00;
return(ret);
}
```

**Program: profile_simulate.c**

This program obtains the grid point structures from the results of the mesh generation procedure and calculates the temperature and cure profiles using the process differential equations and generalized boundary conditions.

```
#include<stdio.h>
#include<math.h>
#include "grid.h"
#include "engine.h"
```

```
/* Profile generation program to evaluate
temperature and cure profiles as a function of time */
profile_gen(grid_pt,n,m,l)
struct point *grid_pt; /* Shape information for every
grid point on finest grid */
int n,m,l; /* Mesh size over finest grid */
{
extern int psi,eta,zei,bound_condition;
int i,bc;
extern char gra[5];
double ***T,***X,td,***mem_alloc_3D();
extern double final_time,delt,t_display;
extern struct varray top;
extern Engine *eptr;
extern int N,M,K;
extern FILE *numal_input,*proc_inp,*prop_f,*disp_f;
double t,find_min(),find_max(),MinT,MaxT;
/* Read in property, process and simulation
parameters for multigrid method */
proc_inp=fopen("PROCESS.DAT","r");
prop_f=fopen("PROPERTY.DAT","r");
numal_input=fopen("NUM.DAT","r");
read_process_input(grid_pt,n,m,l);
/* Read in display options for profile simulation
program */
disp_f=fopen("DISPLAY.DAT","r");
read_profile_plot_options(n,m,l);
if(strcmp(gra,"yes")==0)
initialize_workspace(grid_pt,N,M,K);
T=mem_alloc_3D(N,M,K);
X=mem_alloc_3D(N,M,K);
t=0;
td=t_display;
/* Set initial conditions */
Initial_Condition(T,X,N,M,K);
printf("Time : %6.3lf; ",t);
MinT=find_min(T,N,M,K);
MaxT=find_max(T,N,M,K);
printf("MinT : %6.4lf; MaxT : %6.4lf \n",MinT,MaxT);
if(strcmp(gra,"yes")==0)
Display_variables(t,grid_pt,T,X,N,M,K);
bc=bound_condition;
/* Begin simulating over time until final time */
while(fabs(t-final_time)>1.0e-6)
{
/* Set and allocate appropriate structures for
multigrid method */
arrange_struct(t,grid_pt,T,X,n,m,l,bc);
/* Perform FAS algorithm at current time */
FAS(t,&top,bc);
/* Increment time */
t=t+delt;
/* If it is time for display, display the results and
show graphically if required */
if(fabs(t-td)<=1.0e-6)
{
printf("Time : %6.3lf; ",t);
MinT=find_min(top.T,N,M,K);
MaxT=find_max(top.T,N,M,K);
printf("MinT : %6.4lf; MaxT : %6.4lf \n",MinT,MaxT);
if(strcmp(gra,"yes")==0)
Display_variables(t,grid_pt,top.T,top.X,N,M,K);
td=td+t_display;
}
/* Update next guess as the current temperature and
cure profiles */
mat_copy(T,top.T,n,m,l);
mat_copy(X,top.X,n,m,l);
}
/* Print final values at all grid points into file for
fututre use */
print_values(grid_pt,top.T,top.X,N,M,K);
print_temp(grid_pt,T,X,n,m,l);
mat_lab_options();
print_plot_options();
free_3D(T,N,M,K);
free_3D(X,N,M,K);
return;
}



print_temp(grid_pt,T,X,n,m,l)
struct point *grid_pt;
double ***T,***X;
int n,m,l;
{
int i,j,k;
double r;
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
r=sqrt(pow(grid_pt->x[n-1][j][j],2.0)+pow(grid_pt-
>y[n-1][j][j],2.0)+pow(grid_pt->z[n-1][j][j],2.0));
printf("R = %6.5lf ; T[%d %d %d] = %6.4lf ; X[%d %d
%d] : %7.5lf \n",r,n-1,j,k,T[n-1][j][k],n-1,j,k,X[n-
1][j][k]);
}
}
return;
}



print_shape_temp(grid_pt,T,X,n,m,l)
struct point *grid_pt;
double ***T,***X;
int n,m,l;
{
double ***x,***y,***z,**a,jac;
extern FILE *mat_lab;
int i,j,k,lim,s;
double r;
x=grid_pt->x;
y=grid_pt->y;
z=grid_pt->z;
mat_lab=fopen("TempV.m","w");
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
for(k=0;k<3;k++)
{
r=sqrt(pow(x[i][j][k],2.0)+pow(y[i][j][k],2.0)+pow(z[i][j]
[k],2.0));
fprintf(mat_lab,"(%d,%d,%d); x:%6.4lf y:%6.4lf
z:%6.4lf r:%6.4lf T:%6.3lf
X:%6.3lf\n",i,j,k,x[i][j][k],y[i][j][k],z[i][j][k],r,T[i][j][k],
X[i][j][k]);
```

```
}
}
}
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
for(k=0;k<3;k++)
{
a=grid_pt->a[i][j][k].c;
jac=grid_pt->J[i][j][k];
fprintf(mat_lab,"(%d,%d,%d);a11:%8.7lf; a12:%8.7lf;
a13:%8.7lf; a22:%8.7lf; a23:%8.7lf ; a33:%8.7lf;
J:%8.7lf\n",i,j,k,a[0][0],a[0][1],a[0][2],a[1][1],a[1][2],a[
2][2],jac);
}
}
}
for(i=8;i>5;i--)
{
for(j=8;j>5;j--)
{
for(k=8;k>5;k--)
{
r=sqrt(pow(x[i][j][k],2.0)+pow(y[i][j][k],2.0)+pow(z[i][j]
[k],2.0));
fprintf(mat_lab,"(%d,%d,%d); x:%6.4lf y:%6.4lf
z:%6.4lf r:%6.4lf T:%6.3lf
X:%6.3lf\n",i,j,k,x[i][j][k],y[i][j][k],z[i][j][k],r,T[i][j][k],
X[i][j][k]);
}
}
}
for(i=8;i>5;i--)
{
for(j=8;j>5;j--)
{
for(k=8;k>5;k--)
{
a=grid_pt->a[i][j][k].c;
jac=grid_pt->J[i][j][k];
fprintf(mat_lab,"(%d,%d,%d);a11:%8.7lf; a12:%8.7lf;
a13:%8.7lf; a22:%8.7lf; a23:%8.7lf ; a33:%8.7lf;
J:%8.7lf\n",i,j,k,a[0][0],a[0][1],a[0][2],a[1][1],a[1][2],a[
2][2],jac);
}
}
}
return;
}



print_values(grid_pt,T,X,n,m,l)
struct point *grid_pt;
double ***T,***X;
int n,m,l;
{
double ***x,***y,***z;
double **a;
extern FILE *mat_lab;
int i,j,k;
double r,find_min(),find_max();
x=grid_pt->x;
y=grid_pt->y;
z=grid_pt->z;
fprintf(mat_lab,"tmp=[");
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
r=sqrt(pow(x[i][j][k],2.0)+pow(y[i][j][k],2.0)+pow(z[i][j]
[k],2.0));
fprintf(mat_lab,"%d %d %d %8.7lf %8.7lf %8.7lf
%8.7lf %6.3lf
%8.7lf\n",i,j,k,x[i][j][k],y[i][j][k],z[i][j][k],r,T[i][j][k],X[i
][j][k]);
}
}
}
fprintf(mat_lab,"];\n");
fprintf(mat_lab,"%%T_min : %6.3lf ; T_max : %6.3lf
\n",find_min(T,n,m,l),find_max(T,n,m,l));
/* for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
r=sqrt(pow(x[i][j][k],2.0)+pow(y[i][j][k],2.0)+pow(z[i][j]
[k],2.0));
fprintf(mat_lab,"%d %d %d  x:%8.7lf y:%8.7lf
z:%8.7lf\n",i,j,k,x[i][j][k],y[i][j][k],z[i][j][k]);
}
}
}
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
a=grid_pt->a[i][j][k].c;
fprintf(mat_lab,"%d %d %d  a11:%8.7lf a22:%8.7lf
a33:%8.7lf a12:%8.7lf a13:%8.7lf a23:%8.7lf
J:%8.7lf\n",i,j,k,a[0][0],a[1][1],a[2][2],a[0][1],a[0][2],a[
1][2],grid_pt->J[i][j][k]);
}
}
}*/
return;
}
```

**Program : temp_grid.c**

This part of the code calculates the interior and boundary difference operators of the process differential equations and also relaxes the set of equations at the different grid levels using the FAS algorithm.

```
#include<stdio.h>
#include<math.h>
#include<malloc.h>
#include "grid.h"
#define gas_const 8.314
/* Calculate L_h(u_h) given u_h */
Construct_L(t,grid_pt,conv,Opr,mul,T,X,n,m,l,bc)
double t; /* Time */
struct point *grid_pt; /* Shape factors for current grid
'h' */
double ***conv; /* Transfer coefficient matrix for
current 'h' */
double ***Opr,***mul,***T,***X; /* Opr : Return
operator
   mul : Time derivative coefficient
   T,X : Dependnet variables (u_h) */
int n,m,l,bc; /* Mesh sizes and natuer of BC */
{
double ***Th,***mem_alloc_3D(),L_interior();
double boundary_operator();
int i,j,k,bound(),m1,m2,m3;
/* Allocate memory and calculate space derivative
coefficients */
Th=mem_alloc_3D(n,m,l);
fill_conductivity(Th,T,X,n,m,l);
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
/* If the point is a boundary point */
if(((i==0)||(i==n-1))||((j==0)||(j==m-
1))||((k==0)||(k==l-1)))
{
/* If nature of BC is mixed */
if(bc==2)
{
if(((i==0)||(i==n-1))&&((j==0)||(j==m-
1))&&((k==0)||(k==l-1)))
{
m1=bound(i,n);
m2=bound(j,m);
m3=bound(k,l);
/* Corner points (no normals) and hence averages of
three nearest neighbours */
Opr[i][j][k]=T[i][j][k]-1.0/3.0*(T[i-m1][j][k]+T[i][j-
m2][k]+T[i][j][k-m3]);
}
/* Edge points (no normals) and hence averages of
two nearest neighbours */
if(((i==0)||(i==n-1))&&((j==0)||(j==m-
1))&&((k>0)&&(k<l-1)))
{
m1=bound(i,n);
m2=bound(j,m);
Opr[i][j][k]=T[i][j][k]-1.0/2.0*(T[i-m1][j][k]+T[i][j-
m2][k]);
}
if(((i==0)||(i==n-1))&&((j>0)&&(j<m-
1))&&((k==0)||(k==l-1)))
{
m1=bound(i,n);
m3=bound(k,l);
Opr[i][j][k]=T[i][j][k]-1.0/2.0*(T[i-m1][j][k]+T[i][j][k-
m3]);
}
if(((i>0)&&(i<n-1))&&((j==0)||(j==m-
1))&&((k==0)||(k==l-1)))
{
m2=bound(j,m);
m3=bound(k,l);
Opr[i][j][k]=T[i][j][k]-1.0/2.0*(T[i][j-m2][k]+T[i][j][k-
m3]);
}
/* Pure surface points (not corners or edges) calculate
the boundary operator
using fictitious boundary point assumptions */
if(((i==0)||(i==n-1))&&((j>0)&&(j<m-
1))&&((k>0)&&(k<l-1)))
Opr[i][j][k]=boundary_operator(t,1,grid_pt,T,X,Th,con
v,mul,i,j,k,n,m,l);
if(((i>0)&&(i<n-1))&&((j==0)||(j==m-
1))&&((k>0)&&(k<l-1)))
Opr[i][j][k]=boundary_operator(t,2,grid_pt,T,X,Th,con
v,mul,i,j,k,n,m,l);
if(((i>0)&&(i<n-1))&&((j>0)&&(j<m-
1))&&((k==0)||(k==l-1)))
Opr[i][j][k]=boundary_operator(t,3,grid_pt,T,X,Th,con
v,mul,i,j,k,n,m,l);
}
else
/* If nature of BC is Dirichlet we have equations of
the form
T(on the boundary) = Connstant */
Opr[i][j][k]=T[i][j][k];
}
else
/* If the point is an interior point */
Opr[i][j][k]=L_interior(grid_pt,mul,Th,T,X,i,j,k,n,m,l);
}
}
}
/* printf("Construct_L ends\n"); */
free_3D(Th,n,m,l);
return;
}


/* Calculate the interior point operator for a given
point */
double L_interior(grid_pt,mul,Th,T,X,i,j,k,n,m,l)
struct point *grid_pt; /* Shape factor structure for
current grid */
double ***mul,***Th,***T,***X; /* Time derivative
coefficients
   Space derivative coefficients
   Dependent variables (u_h) */
int i,j,k,n,m,l; /* Grid point coordinates and Mesh
sizes */
{
struct point *temp;
extern double delh,vol_f,rho_f,delt;
```

```
double
ret,term1,term2,term3,term4,term5,term6,term7,ter
m8,term9,term0,den;
double **a,jac,kav;
double
density(),f_psi(),f_eta(),f_zei(),f_zei_eta(),f_eta_zei(),f_
psi_psi(),f_eta_eta();
double
f_zei_zei(),f_psi_eta(),f_eta_psi(),f_zei_psi(),f_psi_zei(),
MW_power(),kinetics_rate(),P(),Q(),R();
temp=grid_pt;
/* Calculate average space derivative value (isotropic)
at current grid point */
kav=1.0/6.0*(Th[i+1][j][k]+Th[i-
1][j][k]+Th[i][j+1][k]+Th[i][j-
1][k]+Th[i][j][k+1]+Th[i][j][k-1]);
a=temp->a[i][j][k].c;
jac=temp->J[i][j][k];
den=density(T[i][j][k],X[i][j][k]);
/* Time derivative contribution */
term1=mul[i][j][k]*T[i][j][k];
/* Space derivative contributions */
term2=delt*a[0][0]*f_psi_psi(Th,T,i,j,k,n)/pow(jac,2.0)
;
term3=delt*a[1][1]*f_eta_eta(Th,T,i,j,k,m)/pow(jac,2.0
);
term4=delt*a[2][2]*f_zei_zei(Th,T,i,j,k,l)/pow(jac,2.0);
/* Cross derivatives */
term5=delt*a[0][1]*(f_psi_eta(Th,T,i,j,k,n,m)+f_eta_p
si(Th,T,i,j,k,n,m))/pow(jac,2.0);
term6=delt*a[1][2]*(f_eta_zei(Th,T,i,j,k,m,l)+f_zei_eta
(Th,T,i,j,k,m,l))/pow(jac,2.0);
term7=delt*a[0][2]*(f_zei_psi(Th,T,i,j,k,l,n)+f_psi_zei(
Th,T,i,j,k,l,n))/pow(jac,2.0);
/* Shape-Source term contributuions */
term8=delt*kav*(P(i,j,k)*f_psi(T,i,j,k,n,m,l,2)+Q(i,j,k)
*f_eta(T,i,j,k,n,m,l,2)+R(i,j,k)*f_zei(T,i,j,k,n,m,l,2));
/* Include following terms if using an implicit method
term9=delt*(-
delh)*kinetics_rate(T[i][j][k],X[i][j][k])*(den-
rho_f*vol_f);
term0=delt*MW_power(T[i][j][k],X[i][j][k]); */
/* Total is sum of all contributions */
ret=term1-
0.50*(term2+term3+term4+term5+term6+term7+ter
m8);
return(ret);
}


/* Calculate off-diagonal neighbour contribution to
relaxation source terms */
double L_off_diag(grid_pt,mul,Th,T,X,i,j,k,n,m,l)
struct point *grid_pt; /* Shape structure for current
grid size */
double ***mul,***Th,***T,***X; /* Time and space
derivative coefficients and dependent variable
matrices */
int i,j,k,n,m,l; /* Grid point coordibates (i,j,k) and
mesh sizes (n,m,l) */
{
extern int N,M,K;
struct point *temp;
extern double delh,vol_f,rho_f,delt;
double
ret,term2,term3,term4,term5,term6,term7,term8,ter
m9,term0,den,h,g,r;
double **a,jac,kav,kinetics_rate();
double
density(),f_psi(),f_eta(),f_zei(),f_zei_eta(),f_eta_zei(),f_
psi_psi(),f_eta_eta();
double
f_zei_zei(),f_psi_eta(),f_eta_psi(),f_zei_psi(),f_psi_zei(),
MW_power(),kinetics_rate(),P(),Q(),R();
/* Calculate current grid spacings */
h=(double)(N-1)/(n-1);
g=(double)(M-1)/(m-1);
r=(double)(K-1)/(l-1);
temp=grid_pt;
/* Calculate isotropic average of space derivative
coefficients */
kav=1.0/6.0*(Th[i+1][j][k]+Th[i-
1][j][k]+Th[i][j+1][k]+Th[i][j-
1][k]+Th[i][j][k+1]+Th[i][j][k-1]);
a=temp->a[i][j][k].c;
jac=temp->J[i][j][k];
/* Neighbours along Psi direction */
term2=delt*0.5*(Th[i+1][j][k]+Th[i-
1][j][k])*a[0][0]*(T[i+1][j][k]+T[i-
1][j][k])/(pow(h,2.0)*pow(jac,2.0));
/* Neighbours along Eta direction */
term3=delt*0.5*(Th[i][j+1][k]+Th[i][j-
1][k])*a[1][1]*(T[i][j+1][k]+T[i][j-1][k])/pow(jac*g,2.0);
/* Neighbours along Zei direction */
term4=delt*0.5*(Th[i][j][k+1]+Th[i][j][k-
1])*a[2][2]*(T[i][j][k+1]+T[i][j][k-1])/pow(jac*r,2.0);
/* Cross Derivative terms do not involve current grid
point so they are entirely off diagonal */
term5=delt*a[0][1]*(f_psi_eta(Th,T,i,j,k,n,m)+f_eta_p
si(Th,T,i,j,k,n,m))/pow(jac,2.0);
term6=delt*a[1][2]*(f_eta_zei(Th,T,i,j,k,m,l)+f_zei_eta
(Th,T,i,j,k,m,l))/pow(jac,2.0);
term7=delt*a[0][2]*(f_zei_psi(Th,T,i,j,k,l,n)+f_psi_zei(
Th,T,i,j,k,l,n))/pow(jac,2.0);
/* Shape factor contributions (currently zero) */
term8=delt*kav*(P(i,j,k)*f_psi(T,i,j,k,n,m,l,2)+Q(i,j,k)
*f_eta(T,i,j,k,n,m,l,2)+R(i,j,k)*f_zei(T,i,j,k,n,m,l,2));
/* Include following terms if using implicit method for
nonlinear terms
term9=delt*(-
delh)*kinetics_rate(T[i][j][k],X[i][j][k])*(den-
rho_f*vol_f);
term0=delt*MW_power(T[i][j][k],X[i][j][k]); */
/* Total is sum weighted by Crank-Nicolson average
weight */
ret=-
0.50*(term2+term3+term4+term5+term6+term7+ter
m8);
return(ret);
}


/* Calculate the source terms independent of current
dependent variable values */
set_rhs(t,grid_pt,conv,rhs,mul,Tg,Xg,n,m,l,bc)
double t; /* Current Time */
struct point *grid_pt; /* Shape structure */
```

```
double ***conv,***rhs,***mul,***Tg,***Xg; /*
Coefficient and Prev. time step dependent variable
matrices */
int n,m,l,bc; /* (n,m,l) : Mesh sizes; bc : Nature of BC
*/
{
extern int N,M,K;
extern double delt,delh,vol_f,rho_f;
int c,i,j,k;
double
ht,cp,thermal_cycle(),density(),kinetics_rate(),MW_po
wer(),h,Ts,rho;
double coeff,source,off,**a,jac,t1,t2,t3,t4,t5,t6,t7;
double
density(),f_psi(),f_eta(),f_zei(),f_zei_eta(),f_eta_zei(),f_
psi_psi(),f_eta_eta();
double
f_zei_zei(),f_psi_eta(),f_eta_psi(),f_zei_psi(),f_psi_zei();
double P(),Q(),R(),heat_transfer_coeff(),kav,rxn_mw;
double
***Th,***mem_alloc_3D(),boundary_operator(),bound
ary_const_temp(),thermal_conductivity();
double
compute_boundary_off_diagonal(),combined_boundar
y_source(),compute_boundary_coefficient();
/* Calculate space derivative coefficient values at
previous time */
Th=mem_alloc_3D(n,m,l);
fill_conductivity(Th,Tg,Xg,n,m,l);
/* Calculate ambient temperature from cure cycle
data */
Ts=thermal_cycle(t+delt);
/* Over all the grid points */
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
/* Set rhs on the boundaries from BC */
if((i==0)||(i==n-1)||(j==0)||(j==m-
1)||(k==0)||(k==l-1))
{
if(bc==1)
/* If BC is Dirichlet then the specification is T(i,j,k) =
T_amb */
rhs[i][j][k]=thermal_cycle(t+delt);
/* If BC is mixed */
else
{
/* For all corner and edge points the source term is
zero */
if(((i==0)||(i==n-1))&&((j==0)||(j==m-
1))&&((k==0)||(k==l-1)))
rhs[i][j][k]=0.00;
if(((i==0)||(i==n-1))&&(j>0)&&(j<m-
1))&&((k==0)||(k==l-1)))
rhs[i][j][k]=0.00;
if((i>0)&&(i<n-1))&&((j==0)||(j==m-
1))&&((k==0)||(k==l-1)))
rhs[i][j][k]=0.00;
if(((i==0)||(i==n-1))&&((j==0)||(j==m-
1))&&((k>0)||(k<l-1)))
rhs[i][j][k]=0.00;
/* For pure surface points (non-corner and non-edge)
*/
/* Psi constamnt surface */
if(((i==0)||(i==n-1))&&(j>0)&&(j<m-
1))&&((k>0)&&(k<l-1)))
{
rho=density(Tg[i][j][k],Xg[i][j][k]);
rxn_mw=delt*((rho-
vol_f*rho_f)*kinetics_rate(Tg[i][j][k],Xg[i][j][k])*(-
delh)+MW_power(Tg[i][j][k],Xg[i][j][k]));
c=1;
/* Calculate source terms from nonlinear previous
time value contributions
and nearest neighbour contributions due to
derivatives of variables at
previous time step (this will be zero if an explicit
method is used) */
source=2.0*combined_boundary_source(t,c,grid_pt,Tg
,Xg,Th,conv,mul,i,j,k,n,m,l);
off=compute_boundary_off_diagonal(t,c,grid_pt,Tg,Xg
,Th,conv,mul,i,j,k,n,m,l);
/* Old time contribution in time derivative */
coeff=mul[i][j][k]-
compute_boundary_coefficient(t,c,grid_pt,Tg,Xg,Th,co
nv,mul,i,j,k,n,m,l);
rhs[i][j][k]=coeff*Tg[i][j][k]-source-off+rxn_mw;
}
/* Eta = Const surface */
if(((i>0)&&(i<n-1))&&((j==0)||(j==m-
1))&&((k>0)&&(k<l-1)))
{
rho=density(Tg[i][j][k],Xg[i][j][k]);
rxn_mw=delt*((rho-
vol_f*rho_f)*kinetics_rate(Tg[i][j][k],Xg[i][j][k])*(-
delh)+MW_power(Tg[i][j][k],Xg[i][j][k]));
c=2;
source=2.0*combined_boundary_source(t,c,grid_pt,Tg
,Xg,Th,conv,mul,i,j,k,n,m,l);
off=compute_boundary_off_diagonal(t,c,grid_pt,Tg,Xg
,Th,conv,mul,i,j,k,n,m,l);
coeff=mul[i][j][k]-
compute_boundary_coefficient(t,c,grid_pt,Tg,Xg,Th,co
nv,mul,i,j,k,n,m,l);
rhs[i][j][k]=coeff*Tg[i][j][k]-source-off+rxn_mw;
}
/* Zei = constant surface */
if(((i>0)&&(i<n-1))&&((j>0)&&(j<m-
1))&&((k==0)||(k==l-1)))
{
rho=density(Tg[i][j][k],Xg[i][j][k]);
rxn_mw=delt*((rho-
vol_f*rho_f)*kinetics_rate(Tg[i][j][k],Xg[i][j][k])*(-
delh)+MW_power(Tg[i][j][k],Xg[i][j][k]));
c=3;
source=2.0*combined_boundary_source(t,c,grid_pt,Tg
,Xg,Th,conv,mul,i,j,k,n,m,l);
off=compute_boundary_off_diagonal(t,c,grid_pt,Tg,Xg
,Th,conv,mul,i,j,k,n,m,l);
coeff=mul[i][j][k]-
compute_boundary_coefficient(t,c,grid_pt,Tg,Xg,Th,co
nv,mul,i,j,k,n,m,l);
rhs[i][j][k]=coeff*Tg[i][j][k]-source-off+rxn_mw;
}
}
}
```

```
/* Interior Points */
else
{
/* Set rhs on the interior points */
a=grid_pt->a[i][j][k].c;
jac=grid_pt->J[i][j][k];
kav=1.0/6.0*(Th[i][j+1][k]+Th[i][j-
1][k]+Th[i+1][j][k]+Th[i-
1][j][k]+Th[i][j][k+1]+Th[i][j][k-1]);
rho=density(Tg[i][j][k],Xg[i][j][k]);
/* Space derivative contributions */
t1=a[0][0]*f_psi_psi(Th,Tg,i,j,k,n)/pow(jac,2.0);
t2=a[1][1]*f_eta_eta(Th,Tg,i,j,k,m)/pow(jac,2.0);
t3=a[2][2]*f_zei_zei(Th,Tg,i,j,k,l)/pow(jac,2.0);
t4=a[0][1]*(f_psi_eta(Th,Tg,i,j,k,n,m)+f_eta_psi(Th,Tg
,i,j,k,n,m))/pow(jac,2.0);
t5=a[1][2]*(f_eta_zei(Th,Tg,i,j,k,m,l)+f_zei_eta(Th,Tg,
i,j,k,m,l))/pow(jac,2.0);
t6=a[0][2]*(f_zei_psi(Th,Tg,i,j,k,l,n)+f_psi_zei(Th,Tg,i,
j,k,l,n))/pow(jac,2.0);
/* Shape contributions */
t7=kav*(P(i,j,k)*f_psi(Tg,i,j,k,n,m,l,2)+Q(i,j,k)*f_eta(T
g,i,j,k,n,m,l,2)+R(i,j,k)*f_zei(Tg,i,j,k,n,m,l,2));
/* Total is sume of different contributions including
time derivative
and contributions from nonlinear terms */
rhs[i][j][k]=mul[i][j][k]*Tg[i][j][k]+delt*((rho-
vol_f*rho_f)*kinetics_rate(Tg[i][j][k],Xg[i][j][k])*(-
delh)+MW_power(Tg[i][j][k],Xg[i][j][k]))+0.50*delt*(t1
+t2+t3+t4+t5+t6+t7);
}
}
}
}
free_3D(Th,n,m,l);
return;
}


/* Relaxation Routine For Temperature and Cure .
This could be used as
a relaxation routine for a single variable in a 3-D
parabolic different
ial equation. T (Temperature) is the main variable
being solved for.
In the case of only one variable pass a 'NULL' pointer
in the place of
X & Xg */
find_next(t,grid_pt,conv,mul,T,X,Tg,Xg,Sf,n,m,l,nu,bc
)
double t; /* Time */
struct point *grid_pt; /* Structural Parameters ast
current grid */
double ***conv,***mul,***T,***X,***Tg,***Xg,***Sf;
/* conv : Convective heat transfer coefficents
mul : Coefficient of time derivative
T : Temperature;
X : Cure;
Tg,Xg : Temperature and cure evaluated at previous
time step ;
Sf : Sum of all source terms (sum of non dependent
variable based
terms and terms evaluated at previous time step ; */
int n,m,l;/* Mesh sizes in three primary directions */
int nu,bc; /* nu : No of iterations to be performed
   bc : Indicates kind of BC ( 1 : Constant/Dirichlet
  2 : Mixed/General/Robin )
      */
{
extern double final_time,delt;
extern int N,M,K;
double
***Th,h,g,r,denm,t1,t2,norm(),boundary_surface_tem
p(),corner_point_temp(),edge_point_temp(),L_interior
(),solve_kinetics();
int
i,j,k,count,cnti,cntj,cntk,begi,begj,begk,limit,m1,m2,m
3,bound();
double
error,errord,Lop,dnl_dT,dLop,***mem_alloc_3D();
double
***old_T,T_max,T_min,**a,jac,nonlinear_derivative()
,temp_check();
double
thermal_conductivity(),L_off_diag(),boundary_const_t
emp(),find_boundary_temp(),ret,thermal_cycle();
double find_min(),find_max(),tol,rho,rxn_mw;
/* Allocate memory for storing thermal
conductivity/Coefficient of
space derivative data */
Th=mem_alloc_3D(n,m,l);
fill_conductivity(Th,T,X,n,m,l);
/* Calculate space steps in three principal directions
for current grid*/
h=((double)(N-1))/(n-1);
g=((double)(M-1))/(m-1);
r=((double)(K-1))/(l-1);
errord=1.0;
count=1;
limit=nu;
tol=1.0e-4;
if((n==3)||(m==3)||(l==3))
{
old_T=mem_alloc_3D(n,m,l);
errord=1;
}
else
errord=1.0e-8;
/* Gauss-Seidel Red-Black Relaxation Sweeps */
/* Outer most Loop Controlling no of sweeps */
while((count<=limit)||(errord>tol))
{
cnti=1;
begi=0;
if((n==3)||(m==3)||(l==3))
mat_copy(old_T,T,n,m,l);
/* Red or Black Sweeps (1)*/
while(cnti<=2)
{
/* Psi Direction */
for(i=begi;i<n;i=i+2)
{
cntj=1;
begj=0;
/* Red or Black Sweeps (2)*/
while(cntj<=2)
{
/* Eta Direction */
for(j=begj;j<m;j=j+2)
{
```

```
cntk=1;
begk=0;
/* Red or Black Sweeps (3)*/
while(cntk<=2)
{
/* Zei Direction */
for(k=begk;k<l;k=k+2)
{
if((i==0)||(i==n-1)||(j==0)||(j==m-
1)||(k==0)||(k==l-1))
{
/* Calculate surface variables using boundary
conditions
1. Corner and Edge points (Use averages)
2. Surface points (Use fictitious outside boundary
points
   to evaluate derivatives)
*/
m1=bound(i,n);
m2=bound(j,m);
m3=bound(k,l);
if(((i==0)||(i==n-1))&&((j==0)||(j==m-
1))&&((k==0)||(k==l-1)))
ret=1.0/3.0*(T[i-m1][j][k]+T[i][j-m2][k]+T[i][j][k-m3]);
if(((i==0)||(i==n-1))&&((j>0)&&(j<m-
1))&&((k==0)||(k==l-1)))
ret=0.50*(T[i-m1][j][k]+T[i][j][k-m3]);
if((i>0)&&(i<n-1))&&((j==0)||(j==m-
1))&&((k==0)||(k==l-1)))
ret=0.50*(T[i][j-m2][k]+T[i][j][k-m3]);
if(((i==0)||(i==n-1))&&((j==0)||(j==m-
1))&&((k>0)||(k<l-1)))
ret=0.50*(T[i-m1][j][k]+T[i][j-m2][k]);
if(((i==0)||(i==n-1))&&((j>0)&&(j<m-
1))&&((k>0)&&(k<l-1)))
ret=find_boundary_temp(t,1,grid_pt,conv,Sf,Th,mul,T
,X,Tg,Xg,n,m,l,i,j,k,bc);
if((i>0)&&(i<n-1))&&((j==0)||(j==m-
1))&&((k>0)&&(k<l-1)))
ret=find_boundary_temp(t,2,grid_pt,conv,Sf,Th,mul,T
,X,Tg,Xg,n,m,l,i,j,k,bc);
if(((i>0)&&(i<n-1))&&((j>0)&&(j<m-
1))&&((k==0)||(k==l-1)))
ret=find_boundary_temp(t,3,grid_pt,conv,Sf,Th,mul,T
,X,Tg,Xg,n,m,l,i,j,k,bc);
}
else
{
/* Calculate value of variables within the interior */
jac=grid_pt->J[i][j][k];
a=grid_pt->a[i][j][k].c;
denm=pow(jac,2.0);
/* Contribution from time derivative */
t1=mul[i][j][k];
/* Contributions from space derivatives */
t2=0.50*delt*(a[0][0]*(Th[i+1][j][k]+Th[i-
1][j][k])/pow(h,2.0)+a[1][1]*(Th[i][j+1][k]+Th[i][j-
1][k])/pow(g,2.0)+a[2][2]*(Th[i][j][k+1]+Th[i][j][k-
1])/pow(r,2.0))/denm;
/* Contribution from neighbouring points */
Lop=L_off_diag(grid_pt,mul,Th,T,X,i,j,k,n,m,l);
ret=(Sf[i][j][k]-Lop)/(t1+t2);
}
/* Check for consistency of calculated variable */
T[i][j][k]=temp_check(t,ret,Tg,Xg,mul,i,j,k);
/* Calculate coupled cure variable using 4th order RK
method */
X[i][j][k]=solve_kinetics(T[i][j][k],X[i][j][k],Tg[i][j][k],
Xg[i][j][k]);
/* Update Thermal Conductivity */
Th[i][j][k]=thermal_conductivity(T[i][j][k],X[i][j][k]);
}
begk++;
cntk++;
}
}
begj++;
cntj++;
}
}
begi++;
cnti++;
}
if((n==3)||(m==3)||(l==3))
{
mat_sub(old_T,T,old_T,n,m,l);
T_max=find_max(T,n,m,l);
T_min=find_min(T,n,m,l);
errord=norm(old_T,n,m,l)/(T_max-T_min);
}
count++;
}
/* Free allocated memory for future use */
free_3D(Th,n,m,l);
if((n==3)||(m==3)||(l==3))
free_3D(old_T,n,m,l);
return;
}



/* The Fully Multigrid V algorithm for a single
variable */
FMV(t,v,bc)
double t; /* Time */
struct varray *v; /* The V Structure containing the
variables and shape
i    parameters for the current grid */
int bc; /* Nature of BC 1. Dirichlet, 2. Mixed */
{
extern int nt1,nt2; /* FMV iteration parameters */
int i,j,k;
struct varray *temp; /* V structure for the next level
corase grid */
double ***f,***f1,***f2;
double ***rf,trunc,err,norm();
double ***tf,***vf,***vx,***tx,***rx;
int vn,vm,vl,tn,tm,tl;
double ***mem_alloc_3D();
vn=v->n;
vm=v->m;
vl=v->l;
vf=v->T;
vx=v->X;
/* Perform nt1 relaxations if the current grid is
coarsest i.e contains only one interior point */
if(v->next==NULL)
{
/* printf("Time : %lf; FMV : Coarsest Grid Begins
\n",t); */
```

```
find_next(t,v->grid_pt,v->conv,v->multp,vf,vx,v-
>Tg,v->Xg,v->Sf,vn,vm,vl,nt1,bc);
}
else
{
find_next(t,v->grid_pt,v->conv,v->multp,vf,vx,v-
>Tg,v->Xg,v->Sf,vn,vm,vl,nt1,bc);
temp=v->next;
tn=temp->n;
tm=temp->m;
tl=temp->l;
tf=temp->T;
tx=temp->X;
/* Use restriction operators to restrict current grid
variables to coarser grid */
Restrict(tf,vf,tn,tm,tl);
Restrict(tx,vx,tn,tm,tl);
f1=mem_alloc_3D(tn,tm,tl);
f=mem_alloc_3D(tn,tm,tl);
rf=mem_alloc_3D(tn,tm,tl);
rx=mem_alloc_3D(tn,tm,tl);
f2=mem_alloc_3D(vn,vm,vl);
mat_copy(rf,tf,tn,tm,tl);
/* Calculate the operated values L_2h(u_2h) (Coarse
grid operator) */
Construct_L(t,temp->grid_pt,temp->conv,f,temp-
>multp,tf,tx,tn,tm,tl,bc);
/* Calculate the operated values L_h(u_h) (Current
grid operator) */
Construct_L(t,v->grid_pt,v->conv,f2,v-
>multp,vf,vx,vn,vm,vl,bc);
/* Restrict current grid operated solution f1 = I_h-
2h(L_h(u_h)) */
Restrict(f1,f2,tn,tm,tl);
/* f = L_2h(u_2h)-I_h-2h(L_h(u_h)) */
mat_sub(f,f,f1,tn,tm,tl);
v->trunc=norm(f,tn,tm,tl);
/* f1 =L_2h(u_2h) */
Construct_L(t,temp->grid_pt,temp->conv,f1,temp-
>multp,tf,tx,tn,tm,tl,bc);
Construct_L(t,v->grid_pt,v->conv,f2,v-
>multp,vf,vx,vn,vm,vl,bc);
/* f2 = L_h(u_h) - Sf_h */
mat_sub(f2,f2,v->Sf,vn,vm,vl);
/* f = I_h-2h(L_h(u_h) - Sf_h) */
Restrict(f,f2,tn,tm,tl);
/* Sf_2h = L_2h(u_2h) - I_h-2h(L_h(u_h) - Sf_h) */
mat_sub(temp->Sf,f1,f,tn,tm,tl);
/* Recursive call to FMV using the next coarser grid
*/
FMV(t,temp,bc);
/* f1 = u_2h - I_h-2h(u_h) */
mat_sub(f1,tf,rf,tn,tm,tl);
/* f2 = I2h-h(u_2h - Ih-2h(u_h) */
Interpolate(f2,f1,tn,tm,tl);
/* u_h = u_h + I2h-h(u_2h - Ih-2h(u_h) */
mat_add(vf,f2,vf,vn,vm,vl);
/* Relax u_h */
find_next(t,v->grid_pt,v->conv,v->multp,vf,vx,v-
>Tg,v->Xg,v->Sf,vn,vm,vl,nt2,bc);
/* Free allocated memory */
free_3D(rf,tn,tm,tl);
free_3D(rx,tn,tm,tl);
free_3D(f,tn,tm,tl);
free_3D(f1,tn,tm,tl);
free_3D(f2,vn,vm,vl);
}
/* printf("Time : %lf; FMV (N = %d) ends
here\n",t,vn); */
return;
}



/* Fully Approximate Storage Algorithm */
FAS(t,v,bc)
double t; /* Time */
struct varray *v; /* V Structure for current grid */
int bc; /* Nature of BC */
{
extern int nt0; /* Iteration parameter for FAS */
extern double delt,T_initial;
double thermal_cycle();
int i,j,k,cnt;
struct varray *temp;
double g;
double ***f,***f1,***f2;
double ***rf,err,trunc,norm();
double ***mem_alloc_3D();
double ***sub_T,***tf,***tx,***vf,***vx;
int vn,vm,vl,tn,tm,tl;
vn=v->n;
vm=v->m;
vl=v->l;
vf=v->T;
vx=v->X;
if(v->next==NULL)
{
/* print_source(v->Sf,v->n,v->m,v->l);
print_grid(v->grid_pt,v->n,v->m,v->l); */
for(i=1;i<=nt0;i++)
FMV(t,v,bc);
}
else
{
temp=v->next;
tn=temp->n;
tm=temp->m;
tl=temp->l;
tf=temp->T;
tx=temp->X;
Restrict(tf,vf,tn,tm,tl);
Restrict(tx,vx,tn,tm,tl);
f1=mem_alloc_3D(tn,tm,tl);
f=mem_alloc_3D(tn,tm,tl);
rf=mem_alloc_3D(tn,tm,tl);
f2=mem_alloc_3D(vn,vm,vl);
mat_copy(rf,tf,tn,tm,tl);
/* f1 =L_2h(u_2h) */
Construct_L(t,temp->grid_pt,temp->conv,f1,temp-
>multp,tf,tx,tn,tm,tl,bc);
/* f2 = L_h(u_h) */
Construct_L(t,v->grid_pt,v->conv,f2,v-
>multp,vf,vx,vn,vm,vl,bc);
/* f2 = L_h(u_h) - Sf_h */
mat_sub(f2,f2,v->Sf,vn,vm,vl);
/* f2 = I_h-2h(L_h(u_h) - Sf_h) */
Restrict(f,f2,tn,tm,tl);
/* Sf_2h = L_2h(u_2h) - I_h-2h(L_h(u_h) - Sf_h) */
mat_sub(temp->Sf,f1,f,tn,tm,tl);
```

```
/* Recursive call to FAS over next coarser grid */
FAS(t,temp,bc);
mat_sub(f1,tf,rf,tn,tm,tl);
Interpolate(f2,f1,tn,tm,tl);
/* u_h = u_h + I2h-h(u_2h - Ih-2h(u_h) */
mat_add(vf,f2,vf,vn,vm,vl);
err=1;
cnt=1;
g=0;
/* Perform FMV nt0 times on u_h(corrected) */
while(cnt<=nt0)
{
if(cnt!=1)
g=err/v->trunc;
else
g=1;
FMV(t,v,bc);
Construct_L(t,v->grid_pt,v->conv,f2,v-
>multp,vf,vx,vn,vm,vl,bc);
mat_sub(f2,v->Sf,f2,vn,vm,vl);
err=norm(f2,vn,vm,vl);
cnt++;
}
/* Free allocated memory */
free_3D(rf,tn,tm,tl);
free_3D(f,tn,tm,tl);
free_3D(f1,tn,tm,tl);
free_3D(f2,vn,vm,vl);
}
return;
}



/* To allocate memory and read in parameters into
appropriate multigrid
structures (linked lists) */
arrange_struct(t,grid_pt,T,X,n,m,l,bc)
double t; /* Time */
struct point *grid_pt;/* Shape parameters for finest
grid */
double ***T,***X; /* Temperature and cure or
dependent
variables on the finest grid */
int n,m,l,bc; /* Mesh sizes and nature of boundary
conditions */
{
extern double delt;
extern double ***transfer_coeff;
extern struct varray top;
struct point *arrange_transform();
struct varray *ptr,*temp;
double jac,***mem_alloc_3D(),**a;
int i,j,k;
if(fabs(t)<=1.0e-8)
{
/* If the time is start time then create all structures
and link them */
/* Mesh sizes */
top.n=n;
top.m=m;
top.l=l;
top.next=NULL;
/* Copy finest grid values */
top.T=mem_alloc_3D(n,m,l);
top.X=mem_alloc_3D(n,m,l);
top.Xg=mem_alloc_3D(n,m,l);
top.Tg=mem_alloc_3D(n,m,l);
top.conv=mem_alloc_3D(n,m,l);
/* t=0 => Dependent variables are at initial values */
Initial_Condition(top.T,top.X,n,m,l);
/* Copy heat transfer coefficient values */
mat_copy(top.conv,transfer_coeff,n,m,l);
top.Sf=mem_alloc_3D(n,m,l);
top.multp=mem_alloc_3D(n,m,l);
top.grid_pt=grid_pt;
ptr=&top;
/* For all elements for all structures upto the coarsest
grid */
while((ptr->n>3)&&(ptr->m>3)&&(ptr->l>3))
{
temp=(struct varray *)malloc(sizeof(struct varray));
/* Mesh Size */
temp->n=(ptr->n+1)/2;
temp->m=(ptr->m+1)/2;
temp->l=(ptr->l+1)/2;
/* Allocate memory for dependent variable values at
current time step and previous time step */
temp->T=mem_alloc_3D(temp->n,temp->m,temp->l);
temp->X=mem_alloc_3D(temp->n,temp->m,temp->l);
temp->Xg=mem_alloc_3D(temp->n,temp->m,temp-
>l);
temp->Tg=mem_alloc_3D(temp->n,temp->m,temp-
>l);
/* Allocate memory for transfer coeeficient matrix */
temp->conv=mem_alloc_3D(temp->n,temp->m,temp-
>l);
/* Restrict transfer coefficient matrix to the coarser
grid */
Restrict(temp->conv,ptr->conv,temp->n,temp-
>m,temp->l);
/* Allocate memory for time derivative coefiicients
and source terms */
temp->Sf=mem_alloc_3D(temp->n,temp->m,temp->l);
temp->multp=mem_alloc_3D(temp->n,temp-
>m,temp->l);
/* Calculate shape factors (x,y,z and alpha,beta and
local
jacobians s functions of psi,eta and zei for the coarser
grid */
temp->grid_pt=arrange_transform(ptr-
>grid_pt,temp->n,temp->m,temp->l);
temp->next=NULL;
/* Set current structure to point to next coarser grid
*/
ptr->next=temp;
ptr=ptr->next;
}
}
/* rho*cp=> coefficient of time derivatives evaluated
at previous time */
mat_copy(top.Xg,top.X,n,m,l);
mat_copy(top.Tg,top.T,n,m,l);
evaluate_multiplier(top.multp,top.Tg,top.Xg,top.n,top
.m,top.l);
/* Sf(source terms) evaluated at previous time */
set_rhs(t,top.grid_pt,top.conv,top.Sf,top.multp,top.T,t
op.X,top.n,top.m,top.l,bc);
ptr=&top;
```

```
/* One time only calculations overall all multigrid
passes */
while(ptr->next)
{
temp=ptr->next;
/* Calculate source terms, time derivative coefficients
and
values of dependent variable at previous time step for
all the grid sizes */
Restrict(temp->multp,ptr->multp,temp->n,temp-
>m,temp->l);
Restrict(temp->Tg,ptr->Tg,temp->n,temp->m,temp-
>l);
Restrict(temp->Xg,ptr->Xg,temp->n,temp->m,temp-
>l);
ptr=temp;
}
return;
}



/* Setting up shape parameters for coarse grid,
given the same for the finer grid */
struct point *arrange_transform(f,n,m,l)
struct point *f; /* Shape structure for fine grid */
int n,m,l; /* Coarse grid sizes */
{
double calc_alpha_jac();
struct point *ret,*set_grid();
double
***tempx,***tempy,***tempz,***c,***c1,***mem_all
oc_3D();
int o,p,q,r,s,t,i,j,k;
/* Calculate fine grid sizes */
p=2*n-1;
q=2*m-1;
r=2*l-1;
c=mem_alloc_3D(p,q,r);
c1=mem_alloc_3D(n,m,l);
ret=set_grid(n,m,l);
/* Restrict x,y,z values to coarse grid */
Restrict(ret->x,f->x,n,m,l);
Restrict(ret->y,f->y,n,m,l);
Restrict(ret->z,f->z,n,m,l);
/* Restrict jacobian values to coarse grid */
/* Restrict alpha values over the coarse grid */
/* for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
ret->J[i][j][k]=calc_alpha_jac(ret->a[i][j][k].c,ret-
>x,ret->y,ret->z,n,m,l,i,j,k);
}
} */
update_grid(ret,f->x,f->y,f->z,n,m,l);
Restrict(ret->J,f->J,n,m,l);
for(s=0;s<3;s++)
{
for(t=0;t<3;t++)
{
for(i=0;i<p;i++)
{
for(j=0;j<q;j++)
{
for(k=0;k<r;k++)
{
c[i][j][k]=f->a[i][j][k].c[s][t];
}
}
}
Restrict(c1,c,n,m,l);
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
ret->a[i][j][k].c[s][t]=c1[i][j][k];
}
}
}
}
/* print_grid(ret,n,m,l); */
free_3D(c,p,q,r);
free_3D(c1,n,m,l);
return(ret);
}



/* Calculate numerical derivative (example) of all the
non-linear terms (include specific phenomena
as callable routines or functions */
double nonlinear_derivative(T,X)
double T,X;
{
extern double vol_f,rho_f,delh;
extern double k1o,k2o,k3o,E1,E2,E3,X_gel,B;
double k1,k2,k3;
double
ret,dT,den_dT,pow_dT,kin_dT,den,kinetics_rate(),M
W_power(),density();
den=density(T,X);
dT=1.0e-6*T;
den_dT=(density(T+dT,X)-density(T-dT,X))/(2.0*dT);
if(X<X_gel)
{
k1=k1o*E1/(gas_const*pow(T,2.0))*exp(-
E1/(gas_const*T));
k2=k2o*E2/(gas_const*pow(T,2.0))*exp(-
E2/(gas_const*T));
kin_dT=(k1+k2*X)*(1.00-X)*(B-X);
}
else
{
k3=k3o*E3/(gas_const*pow(T,2.0))*exp(-
E3/(gas_const*T));
kin_dT=k3*(1.0-X);
}
pow_dT=(MW_power(T+dT,X)-MW_power(T-
dT,X))/(2.0*dT);
ret=-0.50*((-delh)*((den-
vol_f*rho_f)*kin_dT+kinetics_rate(T,X)*den_dT)+po
w_dT);
return(ret);
}
```

```
/* Calculate the boundary variable at a given grid
point
depending on the nature of boundary conditions */
double
find_boundary_temp(t,c,grid_pt,conv,Sf,Th,mul,T,X,T
g,Xg,n,m,l,i,j,k,bc)
double t;
int c;
struct point *grid_pt;
double
***conv,***Sf,***Th,***mul,***T,***X,***Tg,***Xg;
int n,m,l,i,j,k,bc;
{
double ret,calculate_boundary_variable();
/* If nature of boundary condition is dirichlet the
variable value is equal to the sum of all source terms
*/
if(bc==1)
ret=Sf[i][j][k];
else
/* If the BC is mixed calculate variable value using
fictitious point assumptions */
ret=calculate_boundary_variable(t,c,grid_pt,T,X,Th,c
onv,mul,Sf,i,j,k,n,m,l);
return(ret);
}



/* Check for consistency in calculated temperature */
double temp_check(t,f,Tg,Xg,mul,i,j,k)
double t,f,***Tg,***Xg,***mul; /* t : Time
  f : Calculated temperature
  Tg,Xg : Previous time temperature
  and cure */
int i,j,k; /* Grid point coordinates */
{
extern double delt,delh,rho_f,vol_f;
double Ts,rho,heat_addition,density();
double ret,kinetics_rate(),MW_power();
int f1;
ret=f; /* Set return variable to calculated
temperature */
Ts=thermal_cycle(t+delt); /* Calculate ambient
temperature at
 current time */
rho=density(Tg[i][j][k],Xg[i][j][k]); /* Density */
/* Net heat addition at the current grid point due to
all phenomena */
heat_addition=rho*(1-vol_f*rho_f)*(-
delh)*kinetics_rate(Tg[i][j][k],Xg[i][j][k])+MW_power(
Tg[i][j][k],Xg[i][j][k])/mul[i][j][k];
/* If heat addition is negligible and convection is into
the
body and the new temperature is less than old set
new temperature equal to old temperature */
if((fabs(heat_addition)<=1.0e-
5)&&(Tg[i][j][k]<=Ts)&&(f<Tg[i][j][k]))
{
ret=Tg[i][j][k];
}
/* If heat addition is negligible and convection is into
the
body and the new temperature is greater than
ambient then set new temperature equal to ambient
temperature */
if((fabs(heat_addition)<=1.0e-
5)&&(Tg[i][j][k]<=Ts)&&(f>Ts))
ret=Ts;
/* If heat addition is negligible and convection is out
of the
body and the new temperature is less than ambient
then set
new temperature equal to ambient temperature */
if((fabs(heat_addition)<=1.0e-
5)&&(Tg[i][j][k]>=Ts)&&(f<Ts))
ret=Ts;
/* If heat addition is negligible and convection is out
of the
body and the new temperature is greater than old
then
set new temperature equal to old temperature */
if((fabs(heat_addition)<=1.0e-
5)&&(Tg[i][j][k]>=Ts)&&(f>Tg[i][j][k]))
ret=Tg[i][j][k];
/* If heat addition is +ve and convection is into the
body and the new temperature is less than old then
set new temperature equal to old temperature */
if((heat_addition>0)&&(Tg[i][j][k]<=Ts)&&(f<Tg[i][j][
k]))
ret=Tg[i][j][k];
/* If heat addition is -ve and convection is out of the
body and the new temperature is greater than old
then
set new temperature equal to old temperature */
if((heat_addition<0)&&(Tg[i][j][k]>=Ts)&&(f>Tg[i][j][
k]))
ret=Tg[i][j][k];
return(ret);
}
```

**Program: fictitious.c**

This code essenytially deals with the generalized boundary conditions associated with the process model and calculates using fictitious boundary point assumptions the various coefficients and source terms involving the interior points. These are then used to obtain the next iterates of the boundary point values in the Gauss-Siedel relaxation procedure.

```
#include<stdio.h>
#include<math.h>
#include<malloc.h>
#include "grid.h"
/* Collect all coefficients of the fictitous dependent
variable in the finite diference representation at the
boundary using fictitious boundaru points
assumptions */
double collect_coeff(t,c,grid_pt,T,X,Th,conv,i,j,k,n,m,l)
double t; /* Time */
int c; /* Parameter to indicate the type of boundary
  1. Psi = constant
  2. Eta = constant
  3. Zei = constant
*/
struct point *grid_pt; /* Shape structure */
double ***T,***X,***Th,***conv; /* Temperature and
Cure variables
   Space derivative coefficients and
```

```
   transfer coefficient matrices */
int i,j,k,n,m,l; /* Grid coordinates (i,j,k) and mesh size
*/
{
double ret_coeff,**a,jac,bcf,dcf,h,g,r,kavg;
extern int N,M,K;
int bound(),m1,m2,m3;
a=grid_pt->a[i][j][k].c;
jac=grid_pt->J[i][j][k];
h=(double)(N-1)/(n-1);
g=(double)(M-1)/(m-1);
r=(double)(K-1)/(l-1);
switch(c)
{
case 1: {
/* Psi = constant surface */
m1=bound(i,n); /* Boundary normal evaluation  = -1
Psi =0;
      = 1  Psi =Psi_max*/
/* Calculate average space derivative coefficient */
if(((j>0)&&(j<m-1))&&((k>0)&&(k<l-1)))
kavg=0.25*(Th[i][j][k+1]+Th[i][j][k-1]+Th[i][j-
1][k]+Th[i][j-1][k]);
else
kavg=Th[i][j][k];
/* Coefficient denomenator contributions */
bcf=m1*kavg/(jac*sqrt(a[0][0]));
dcf=m1*a[0][0]/(2.0*h);
break;
}
case 2: {
/* Eta = constant surface */
m2=bound(j,m);
if(((i>0)&&(i<n-1))&&((k>0)&&(k<l-1)))
kavg=0.25*(Th[i-
1][j][k]+Th[i+1][j][k]+Th[i][j][k+1]+Th[i][j][k-1]);
else
kavg=Th[i][j][k];
bcf=m2*kavg/(jac*sqrt(a[1][1]));
dcf=m2*a[1][1]/(2.0*g);
break;
}
case 3: {
/* Zei = constant surface */
m3=bound(k,l);
if(((i>0)&&(i<n-1))&&((j>0)&&(j<m-1)))
kavg=0.25*(Th[i+1][j][k]+Th[i][j-
1][k]+Th[i][j+1][k]+Th[i-1][j][k]);
else
kavg=Th[i][j][k];
bcf=m3*kavg/(jac*sqrt(a[2][2]));
dcf=m3*a[2][2]/(2.0*r);
break;
}
}
/* Return the coefficient of the boundary point
dependent variable */
ret_coeff=-conv[i][j][k]/(bcf*dcf);
return(ret_coeff);
}
```

```
/* Collect all the off diagonal terms using nearest
neighbour contributions for a fictitious boundary
point using the BCs */
double collect_off(t,c,grid_pt,T,X,Th,conv,i,j,k,n,m,l)
double t; /* Time */
int c; /* Surface parameter */
struct point *grid_pt; /* Shape structure */
double ***T,***X,***Th,***conv; /* Dependent
variables and coefficient amtrices */
int i,j,k,n,m,l; /* Grid coordinates and mesh sizes */
{
double ret_off,**a,jac,bcf,dcf,h,g,r;
double f_psi(),f_eta(),f_zei();
extern int N,M,K;
int bound(),m1,m2,m3;
a=grid_pt->a[i][j][k].c;
jac=grid_pt->J[i][j][k];
h=(double)(N-1)/(n-1);
g=(double)(M-1)/(m-1);
r=(double)(K-1)/(l-1);
switch(c)
{
/* Psi = constant surface */
case 1: {
m1=bound(i,n);
dcf=m1*a[0][0]/(2.0*h);
ret_off=T[i-m1][j][k]-
1.0/dcf*(a[0][1]*f_eta(T,i,j,k,n,m,l,2)+a[0][2]*f_zei(T,i,j
,k,n,m,l,2));
break;
}
/* Eta = constant surface */
case 2: {
m2=bound(j,m);
dcf=m2*a[1][1]/(2.0*g);
ret_off=T[i][j-m2][k]-
1.0/dcf*(a[1][0]*f_psi(T,i,j,k,n,m,l,2)+a[1][2]*f_zei(T,i,j,
k,n,m,l,2));
break;
}
/* Zei = constant surface */
case 3: {
m3=bound(k,l);
dcf=m3*a[2][2]/(2.0*r);
ret_off=T[i][j][k-m3]-
1.0/dcf*(a[2][0]*f_psi(T,i,j,k,n,m,l,2)+a[2][1]*f_eta(T,i,j
,k,n,m,l,2));
break;
}
}
return(ret_off);
}


/* Calculate the source terms (due to ambient
temperature) in the BCs usin the fictitious point
assumptions */
double
collect_source(t,c,grid_pt,T,X,Th,conv,i,j,k,n,m,l)
double t; /* Time */
int c; /* Surface parameter */
struct point *grid_pt; /* Shape structure */
double ***T,***X,***Th,***conv;
int i,j,k,n,m,l;
{
```

```
extern double delt;
double
ret_source,**a,jac,bcf,dcf,h,g,r,Ts,thermal_cycle(),kav
g;
extern int N,M,K;
int bound(),m1,m2,m3;
a=grid_pt->a[i][j][k].c;
jac=grid_pt->J[i][j][k];
h=(double)(N-1)/(n-1);
g=(double)(M-1)/(m-1);
r=(double)(K-1)/(l-1);
Ts=thermal_cycle(t+delt);
switch(c)
{
/* Psi = constant surface */
case 1: {
/* Evaluate normal to boundary */
m1=bound(i,n);
/* Calculate space derivative coeffn. average */
if(((j>0)&&(j<m-1))&&((k>0)&&(k<l-1)))
kavg=0.25*(Th[i][j][k+1]+Th[i][j][k-1]+Th[i][j-
1][k]+Th[i][j+1][k]);
else
kavg=Th[i][j][k];
/* Calculate source denominator contributions */
bcf=m1*kavg/(jac*sqrt(a[0][0]));
dcf=m1*a[0][0]/(2.0*h);
break;
}
/* Eta = constant surface */
case 2: {
m2=bound(j,m);
if(((i>0)&&(i<n-1))&&((k>0)&&(k<l-1)))
kavg=0.25*(Th[i+1][j][k]+Th[i][j][k-
1]+Th[i][j][k+1]+Th[i-1][j][k]);
else
kavg=Th[i][j][k];
bcf=m2*kavg/(jac*sqrt(a[1][1]));
dcf=m2*a[1][1]/(2.0*g);
break;
}
/* Zei = constant surface */
case 3: {
m3=bound(k,l);
if(((i>0)&&(i<n-1))&&((j>0)&&(j<m-1)))
kavg=0.25*(Th[i+1][j][k]+Th[i][j-1][k]+Th[i-
1][j][k]+Th[i][j+1][k]);
else
kavg=Th[i][j][k];
bcf=m3*kavg/(jac*sqrt(a[2][2]));
dcf=m3*a[2][2]/(2.0*r);
break;
}
}
/* The source term involves the ambient temperature
divided by appropriate contributions */
ret_source=conv[i][j][k]*Ts/(bcf*dcf);
return(ret_source);
}



/* Calculate the coefficient of actual point (i,j,k) on
the boundary using the substitution of coefficients
from fictitous point assumption into the interior point
difference equation */
double
compute_boundary_coefficient(t,c,grid_pt,T,X,Th,con
v,mul,i,j,k,n,m,l)
double t;
int c;
struct point *grid_pt;
double ***T,***X,***Th,***conv,***mul;
int i,j,k,n,m,l;
{
extern double delt;
extern int N,M,K;
int bound(),m1,m2,m3;
double P(),Q(),R();
double h,g,r,**a,jac;
double kavg,k_psi_avg,k_eta_avg,k_zei_avg;
double coeff_i,coeff_j,coeff_k;
double
psi_psi_coeff,eta_eta_coeff,zei_zei_coeff,force_psi_coef
f,force_eta_coeff,force_zei_coeff;
double coeff,ret_var;
a=grid_pt->a[i][j][k].c;
jac=grid_pt->J[i][j][k];
h=(double)(N-1)/(n-1);
g=(double)(M-1)/(m-1);
r=(double)(K-1)/(l-1);
switch(c)
{
/* Psi = constant surface */
case 1: {
m1=bound(i,n); /* evaluate boundary normal */
/* Calculate surface average of space derivative
coeffn. */
kavg=0.25*(Th[i][j+1][k]+Th[i][j-1][k]+Th[i][j][k-
1]+Th[i][j][k+1]);
/* Collect appropriate coefficients for T(i,j,k) from
boundary conditions*/
coeff_i=collect_coeff(t,c,grid_pt,T,X,Th,conv,i,j,k,n,m,l
);
/* Calculate averages for space derivatives */
k_eta_avg=0.50*(Th[i][j+1][k]+Th[i][j-1][k]);
k_zei_avg=0.50*(Th[i][j][k+1]+Th[i][j][k-1]);
/* Collect coefficients from double derivatives and
forcing terms */
psi_psi_coeff=a[0][0]*Th[i][j][k]/(pow(jac*h,2.0))*(-
2+coeff_i);
eta_eta_coeff=-2.0*a[1][1]*k_eta_avg/pow(jac*g,2.0);
zei_zei_coeff=-2.0*a[2][2]*k_zei_avg/pow(jac*r,2.0);
force_psi_coeff=m1*kavg*P(i,j,k)/(2.0*h)*coeff_i;
coeff=0.50*delt*(psi_psi_coeff+eta_eta_coeff+zei_zei_
coeff+force_psi_coeff);
break;
}
case 2: {
m2=bound(j,m);
kavg=0.25*(Th[i+1][j][k]+Th[i-1][j][k]+Th[i][j][k-
1]+Th[i][j][k+1]);
/* Collect appropriate coefficients for T(i,j,k) from
boundary conditions*/
coeff_j=collect_coeff(t,c,grid_pt,T,X,Th,conv,i,j,k,n,m,l
);
/* Calculate averages for thermal conductivities */
k_psi_avg=0.50*(Th[i+1][j][k]+Th[i-1][j][k]);
k_zei_avg=0.50*(Th[i][j][k+1]+Th[i][j][k-1]);
```

```
/* Collect coeficients from double derivatives and
forcing terms */
psi_psi_coeff=-2.0*a[0][0]*k_psi_avg/pow(jac*h,2.0);
eta_eta_coeff=a[1][1]*Th[i][j][k]/(pow(jac*g,2.0))*(-
2+coeff_j);
zei_zei_coeff=-2.0*a[2][2]*k_zei_avg/pow(jac*r,2.0);
force_eta_coeff=kavg*m2*Q(i,j,k)/(2.0*g)*coeff_j;
coeff=0.50*delt*(psi_psi_coeff+eta_eta_coeff+zei_zei_
coeff+force_eta_coeff);
break;
}
case 3: {
m3=bound(k,l);
kavg=0.25*(Th[i+1][j][k]+Th[i-1][j][k]+Th[i][j-
1][k]+Th[i][j+1][k]);
/* Collect appropriate coefficients for T(i,j,k) from
boundary conditions*/
coeff_k=collect_coeff(t,c,grid_pt,T,X,Th,conv,i,j,k,n,m,
l);
/* Calculate averages for thermal conductivities */
k_psi_avg=0.50*(Th[i+1][j][k]+Th[i-1][j][k]);
k_eta_avg=0.50*(Th[i][j+1][k]+Th[i][j-1][k]);
/* Collect coeficients from double derivatives and
forcing terms */
psi_psi_coeff=-2.0*a[0][0]*k_psi_avg/pow(jac*h,2.0);
eta_eta_coeff=-2.0*a[1][1]*k_eta_avg/pow(jac*g,2.0);
zei_zei_coeff=a[2][2]*Th[i][j][k]/(pow(jac*r,2.0))*(-
2.0+coeff_k);
force_zei_coeff=kavg*m3*R(i,j,k)/(2.0*r)*coeff_k;
coeff=0.50*delt*(psi_psi_coeff+eta_eta_coeff+zei_zei_
coeff+force_zei_coeff);
break;
}
}
ret_var=-coeff;
/* printf("Coeff (%d,%d,%d) : %6.3lf \n",i,j,k,coeff); */
return(ret_var);
}




double
compute_boundary_off_diagonal(t,c,grid_pt,T,X,Th,co
nv,mul,i,j,k,n,m,l)
double t;
int c;
struct point *grid_pt;
double ***T,***X,***Th,***conv,***mul;
int i,j,k,n,m,l;
{
extern double delt;
extern int N,M,K;
int bound(),m1,m2,m3;
double
f_psi(),f_eta(),f_psi_psi(),f_psi_eta(),f_psi_zei(),f_eta_e
ta(),f_eta_zei(),f_zei_zei(),P(),Q(),R();
double h,g,r,**a,jac;
double kavg,k_psi_avg,k_eta_avg,k_zei_avg;
double
coeff_i_plus,coeff_i_minus,coeff_j_plus,coeff_j_minus,
coeff_k_plus,coeff_k_minus;
double
off_i,off_j,off_k,off_i_plus,off_i_minus,off_j_plus,off_j_
minus,off_k_plus,off_k_minus;
double
psi_psi_off,psi_eta_off,psi_zei_off,eta_eta_off,eta_zei_
off,zei_zei_off,force_psi_off,force_eta_off,force_zei_off;
double off,ret_var;
a=grid_pt->a[i][j][k].c;
jac=grid_pt->J[i][j][k];
h=(double)(N-1)/(n-1);
g=(double)(M-1)/(m-1);
r=(double)(K-1)/(l-1);
switch(c)
{
case 1: {
m1=bound(i,n);
/* Collect appropriate coefficients from boundary
conditions*/
coeff_j_plus=collect_coeff(t,c,grid_pt,T,X,Th,conv,i,j+1
,k,n,m,l);
coeff_j_minus=collect_coeff(t,c,grid_pt,T,X,Th,conv,i,j
-1,k,n,m,l);
coeff_k_plus=collect_coeff(t,c,grid_pt,T,X,Th,conv,i,j,k
+1,n,m,l);
coeff_k_minus=collect_coeff(t,c,grid_pt,T,X,Th,conv,i,
j,k-1,n,m,l);
/* Collect relevant off diagonal terms from boundary
condition */
off_i=collect_off(t,c,grid_pt,T,X,Th,conv,i,j,k,n,m,l);
off_j_plus=collect_off(t,c,grid_pt,T,X,Th,conv,i,j+1,k,n
,m,l);
off_j_minus=collect_off(t,c,grid_pt,T,X,Th,conv,i,j-
1,k,n,m,l);
off_k_plus=collect_off(t,c,grid_pt,T,X,Th,conv,i,j,k+1,
n,m,l);
off_k_minus=collect_off(t,c,grid_pt,T,X,Th,conv,i,j,k-
1,n,m,l);
/* printf("Coeff_j_plus : %6.3lf \n",coeff_j_plus);
printf("Coeff_j_minus : %6.3lf \n",coeff_j_minus);
printf("Coeff_k_plus : %6.3lf \n",coeff_k_plus);
printf("Coeff_k_minus : %6.3lf \n",coeff_k_minus);
printf("I : %d; J : %d ; K : %d \n",i,j,k);
printf("off_i: %6.3lf \n",off_i);
printf("off_j_plus : %6.3lf \n",off_j_plus);
printf("off_j_minus : %6.3lf \n",off_j_minus);
printf("off_k_plus : %6.3lf \n",off_k_plus);
printf("off_k_minus : %6.3lf \n",off_k_minus);
printf("Tm1 : %6.3lf ; Tj-1 : %6.3lf ; Tk-1 : %6.3lf ;
Tj+1 : %6.3lf ; Tk+1 : %6.3lf\n",T[i-m1][j][k],T[i][j-
1][k],T[i][j][k-1],T[i][j+1][k],T[i][j][k+1]); */
/* Calculate averages for thermal conductivities */
kavg=0.25*(Th[i][j+1][k]+Th[i][j-1][k]+Th[i][j][k-
1]+Th[i][j][k+1]);
k_eta_avg=0.50*(Th[i][j+1][k]+Th[i][j-1][k]);
k_zei_avg=0.50*(Th[i][j][k+1]+Th[i][j][k-1]);
/* Collect off diagonal terms from double derivatives
and forcing terms */
psi_psi_off=a[0][0]*Th[i][j][k]/pow(jac*h,2.0)*(T[i-
m1][j][k]+off_i);
eta_eta_off=a[1][1]*k_eta_avg/pow(jac*g,2.0)*(T[i][j+
1][k]+T[i][j-1][k]);
zei_zei_off=a[2][2]*k_zei_avg/pow(jac*r,2.0)*(T[i][j][k
+1]+T[i][j][k-1]);
psi_eta_off=2*a[0][1]*k_eta_avg/(4.0*pow(jac,2.0)*h*
g)*(m1*(coeff_j_plus*T[i][j+1][k]+off_j_plus)-m1*T[i-
```

```
m1][j+1][k]+m1*T[i-m1][j-1][k]-
m1*(coeff_j_minus*T[i][j-1][k]+off_j_minus));
eta_zei_off=2.0*a[1][2]/pow(jac,2.0)*f_eta_zei(Th,T,i,j,
k,m,l);
psi_zei_off=2*a[0][2]*k_zei_avg/(4.0*pow(jac,2.0)*h*r
)*(m1*(coeff_k_plus*T[i][j][k+1]+off_k_plus)-m1*T[i-
m1][j][k+1]+m1*T[i-m1][j][k-1]-
m1*(coeff_k_minus*T[i][j][k-1]+off_k_minus));
force_psi_off=kavg*P(i,j,k)/(2.0*h)*m1*(off_i-T[i-
m1][j][k]);
force_eta_off=kavg*Q(i,j,k)*f_eta(T,i,j,k,n,m,l,2);
force_zei_off=kavg*R(i,j,k)*f_zei(T,i,j,k,n,m,l,2);
/* Compute boundary temperature using source
terms, off diagonals and coefficients */
off=psi_psi_off+eta_eta_off+zei_zei_off+psi_eta_off+e
ta_zei_off+psi_zei_off+force_psi_off+force_eta_off+for
ce_zei_off;
break;
}
case 2: {
m2=bound(j,m);
/* Collect appropriate coefficients from boundary
conditions*/
coeff_i_plus=collect_coeff(t,c,grid_pt,T,X,Th,conv,i+1,j
,k,n,m,l);
coeff_i_minus=collect_coeff(t,c,grid_pt,T,X,Th,conv,i-
1,j,k,n,m,l);
coeff_k_plus=collect_coeff(t,c,grid_pt,T,X,Th,conv,i,j,k
+1,n,m,l);
coeff_k_minus=collect_coeff(t,c,grid_pt,T,X,Th,conv,i,
j,k-1,n,m,l);
/* Collect relevant off diagonal terms from boundary
condition */
off_j=collect_off(t,c,grid_pt,T,X,Th,conv,i,j,k,n,m,l);
off_i_plus=collect_off(t,c,grid_pt,T,X,Th,conv,i+1,j,k,n
,m,l);
off_i_minus=collect_off(t,c,grid_pt,T,X,Th,conv,i-
1,j,k,n,m,l);
off_k_plus=collect_off(t,c,grid_pt,T,X,Th,conv,i,j,k+1,
n,m,l);
off_k_minus=collect_off(t,c,grid_pt,T,X,Th,conv,i,j,k-
1,n,m,l);
/* Calculate averages for thermal conductivities */
kavg=0.25*(Th[i+1][j][k]+Th[i-1][j][k]+Th[i][j][k-
1]+Th[i][j][k+1]);
k_psi_avg=0.50*(Th[i+1][j][k]+Th[i-1][j][k]);
k_zei_avg=0.50*(Th[i][j][k+1]+Th[i][j][k-1]);
/* Collect off diagonal terms from double derivatives
and forcing terms */
psi_psi_off=a[0][0]*k_psi_avg/pow(jac*h,2.0)*(T[i+1][j
][k]+T[i-1][j][k]);
eta_eta_off=a[1][1]*Th[i][j][k]/pow(jac*g,2.0)*(T[i][j-
m2][k]+off_j);
zei_zei_off=a[2][2]*k_zei_avg/pow(jac*r,2.0)*(T[i][j][k
+1]+T[i][j][k-1]);
psi_eta_off=2*a[1][0]*k_psi_avg/(4.0*pow(jac,2.0)*h*g
)*(m2*(coeff_i_plus*T[i+1][j][k]+off_i_plus)-
m2*T[i+1][j-m2][k]+m2*T[i-1][j-m2][k]-
m2*(coeff_i_minus*T[i-1][j][k]+off_i_minus));
eta_zei_off=2*a[1][2]*k_zei_avg/(4.0*pow(jac,2.0)*g*r
)*(m2*(coeff_k_plus*T[i][j][k+1]+off_k_plus)-
m2*T[i][j-m2][k+1]+m2*T[i][j-m2][k-1]-
m2*(coeff_k_minus*T[i][j][k-1]+off_k_minus));
psi_zei_off=2.0*a[0][2]/pow(jac,2.0)*f_psi_zei(Th,T,i,j,
k,l,n);
force_psi_off=kavg*P(i,j,k)*f_psi(T,i,j,k,n,m,l,2);
force_eta_off=kavg*Q(i,j,k)/(2.0*g)*m2*(off_j-T[i][j-
m2][k]);
force_zei_off=kavg*R(i,j,k)*f_zei(T,i,j,k,n,m,l,2);
/* Compute boundary temperature using source
terms, off diagonals and coefficients */
off=psi_psi_off+eta_eta_off+zei_zei_off+psi_eta_off+e
ta_zei_off+psi_zei_off+force_psi_off+force_eta_off+for
ce_zei_off;
break;
}
case 3: {
m3=bound(k,l);
/* Collect appropriate coefficients from boundary
conditions*/
coeff_i_plus=collect_coeff(t,c,grid_pt,T,X,Th,conv,i+1,j
,k,n,m,l);
coeff_i_minus=collect_coeff(t,c,grid_pt,T,X,Th,conv,i-
1,j,k,n,m,l);
coeff_j_plus=collect_coeff(t,c,grid_pt,T,X,Th,conv,i,j+1
,k,n,m,l);
coeff_j_minus=collect_coeff(t,c,grid_pt,T,X,Th,conv,i,j
-1,k,n,m,l);
/* Collect relevant off diagonal terms from boundary
condition */
off_k=collect_off(t,c,grid_pt,T,X,Th,conv,i,j,k,n,m,l);
off_i_plus=collect_off(t,c,grid_pt,T,X,Th,conv,i+1,j,k,n
,m,l);
off_i_minus=collect_off(t,c,grid_pt,T,X,Th,conv,i-
1,j,k,n,m,l);
off_j_plus=collect_off(t,c,grid_pt,T,X,Th,conv,i,j+1,k,n
,m,l);
off_j_minus=collect_off(t,c,grid_pt,T,X,Th,conv,i,j-
1,k,n,m,l);
/* Calculate averages for thermal conductivities */
kavg=0.25*(Th[i+1][j][k]+Th[i-1][j][k]+Th[i][j-
1][k]+Th[i][j+1][k]);
k_psi_avg=0.50*(Th[i+1][j][k]+Th[i-1][j][k]);
k_eta_avg=0.50*(Th[i][j+1][k]+Th[i][j-1][k]);
/* Collect off diagonal terms from double derivatives
and forcing terms */
psi_psi_off=a[0][0]*k_psi_avg/pow(jac*h,2.0)*(T[i+1][j
][k]+T[i-1][j][k]);
eta_eta_off=a[1][1]*k_eta_avg/pow(jac*g,2.0)*(T[i][j+
1][k]+T[i][j-1][k]);
zei_zei_off=a[2][2]*Th[i][j][k]/pow(jac*r,2.0)*(T[i][j][k-
m3]+off_k);
psi_eta_off=2.0*a[1][0]/pow(jac,2.0)*f_psi_eta(Th,T,i,j,
k,n,m);
eta_zei_off=2*a[1][2]*k_eta_avg/(4.0*pow(jac,2.0)*g*r
)*(m3*(coeff_j_plus*T[i][j+1][k]+off_j_plus)-
m3*T[i][j+1][k-m3]+m3*T[i][j-1][k-m3]-
m3*(coeff_j_minus*T[i][j-1][k]+off_j_minus));
psi_zei_off=2*a[2][0]*k_psi_avg/(4.0*pow(jac,2.0)*g*r
)*(m3*(coeff_i_plus*T[i+1][j][k]+off_i_plus)-
m3*T[i+1][j][k-m3]+m3*T[i-1][j][k-m3]-
m3*(coeff_i_minus*T[i-1][j][k]+off_i_minus));
force_psi_off=kavg*P(i,j,k)*f_psi(T,i,j,k,n,m,l,2);
force_eta_off=kavg*Q(i,j,k)*f_eta(T,i,j,k,n,m,l,2);
force_zei_off=kavg*R(i,j,k)/(2.0*r)*m3*(off_k-T[i][j][k-
m3]);
/* Compute boundary temperature using source
terms, off diagonals and coefficients */
```

```
off=psi_psi_off+eta_eta_off+zei_zei_off+psi_eta_off+e
ta_zei_off+psi_zei_off+force_psi_off+force_eta_off+for
ce_zei_off;
break;
}
}
ret_var=-0.50*delt*off;
/* printf("Off (%d,%d,%d) : %6.3lf \n",i,j,k,off); */
return(ret_var);
}



double
combined_boundary_source(t,c,grid_pt,T,X,Th,conv,m
ul,i,j,k,n,m,l)
double t;
int c;
struct point *grid_pt;
double ***T,***X,***Th,***conv,***mul;
int i,j,k,n,m,l;
{
extern double delt;
extern int N,M,K;
int bound(),m1,m2,m3;
double P(),Q(),R(),collect_source();
double h,g,r,**a,jac;
double kavg,k_psi_avg,k_eta_avg,k_zei_avg;
double
source_i,source_j,source_k,source_i_plus,source_i_mi
nus,source_j_plus,source_j_minus,source_k_plus,sour
ce_k_minus;
double
psi_psi_source,psi_eta_source,psi_zei_source,eta_eta_
source,eta_zei_source,zei_zei_source,force_psi_source
,force_eta_source,force_zei_source;
double source,ret_var;
a=grid_pt->a[i][j][k].c;
jac=grid_pt->J[i][j][k];
h=(double)(N-1)/(n-1);
g=(double)(M-1)/(m-1);
r=(double)(K-1)/(l-1);
switch(c)
{
case 1: {
m1=bound(i,n);
/* Collect relevant source diagonal terms from
boundary condition */
source_i=collect_source(t,c,grid_pt,T,X,Th,conv,i,j,k,n
,m,l);
source_j_plus=collect_source(t,c,grid_pt,T,X,Th,conv,i
,j+1,k,n,m,l);
source_j_minus=collect_source(t,c,grid_pt,T,X,Th,con
v,i,j-1,k,n,m,l);
source_k_plus=collect_source(t,c,grid_pt,T,X,Th,conv,
i,j,k+1,n,m,l);
source_k_minus=collect_source(t,c,grid_pt,T,X,Th,co
nv,i,j,k-1,n,m,l);
/* Calculate averages for thermal conductivities */
kavg=0.25*(Th[i][j+1][k]+Th[i][j-1][k]+Th[i][j][k-
1]+Th[i][j][k+1]);
k_eta_avg=0.50*(Th[i][j+1][k]+Th[i][j-1][k]);
k_zei_avg=0.50*(Th[i][j][k+1]+Th[i][j][k-1]);
/* Collect source diagonal terms from double
derivatives and forcing terms */
psi_psi_source=a[0][0]*Th[i][j][k]/pow(jac*h,2.0)*sour
ce_i;
psi_eta_source=2*a[0][1]*k_eta_avg/(4.0*pow(jac,2.0)
*h*g)*m1*(source_j_plus-source_j_minus);
psi_zei_source=2*a[0][2]*k_zei_avg/(4.0*pow(jac,2.0)*
h*r)*m1*(source_k_plus-source_k_minus);
force_psi_source=kavg*P(i,j,k)/(2.0*h)*m1*source_i;
/* Compute boundary temperature using source
terms, source diagonals and coefficients */
source=psi_psi_source+psi_eta_source+psi_zei_sourc
e+force_psi_source;
break;
}
case 2: {
m2=bound(j,m);
/* Collect relevant source diagonal terms from
boundary condition */
source_j=collect_source(t,c,grid_pt,T,X,Th,conv,i,j,k,n
,m,l);
source_i_plus=collect_source(t,c,grid_pt,T,X,Th,conv,i
+1,j,k,n,m,l);
source_i_minus=collect_source(t,c,grid_pt,T,X,Th,con
v,i-1,j,k,n,m,l);
source_k_plus=collect_source(t,c,grid_pt,T,X,Th,conv,
i,j,k+1,n,m,l);
source_k_minus=collect_source(t,c,grid_pt,T,X,Th,co
nv,i,j,k-1,n,m,l);
/* Calculate averages for thermal conductivities */
kavg=0.25*(Th[i+1][j][k]+Th[i-1][j][k]+Th[i][j][k-
1]+Th[i][j][k+1]);
k_psi_avg=0.50*(Th[i+1][j][k]+Th[i-1][j][k]);
k_zei_avg=0.50*(Th[i][j][k+1]+Th[i][j][k-1]);
/* Collect source diagonal terms from double
derivatives and forcing terms */
eta_eta_source=a[1][1]*Th[i][j][k]/pow(jac*g,2.0)*sour
ce_j;
psi_eta_source=2*a[1][0]*k_psi_avg/(4.0*pow(jac,2.0)
*h*g)*m2*(source_i_plus-source_i_minus);
eta_zei_source=2*a[1][2]*k_zei_avg/(4.0*pow(jac,2.0)
*g*r)*m2*(source_k_plus-source_k_minus);
force_eta_source=kavg*Q(i,j,k)/(2.0*g)*m2*source_j;
/* Compute boundary temperature using source
terms, source diagonals and coefficients */
source=eta_eta_source+psi_eta_source+eta_zei_sourc
e+force_eta_source;
break;
}
case 3: {
m3=bound(k,l);
/* Collect relevant source diagonal terms from
boundary condition */
source_k=collect_source(t,c,grid_pt,T,X,Th,conv,i,j,k,
n,m,l);
source_i_plus=collect_source(t,c,grid_pt,T,X,Th,conv,i
+1,j,k,n,m,l);
source_i_minus=collect_source(t,c,grid_pt,T,X,Th,con
v,i-1,j,k,n,m,l);
source_j_plus=collect_source(t,c,grid_pt,T,X,Th,conv,i
,j+1,k,n,m,l);
source_j_minus=collect_source(t,c,grid_pt,T,X,Th,con
v,i,j-1,k,n,m,l);
/* Calculate averages for thermal conductivities */
kavg=0.25*(Th[i+1][j][k]+Th[i-1][j][k]+Th[i][j-
1][k]+Th[i][j+1][k]);
k_psi_avg=0.50*(Th[i+1][j][k]+Th[i-1][j][k]);
```

```
k_eta_avg=0.50*(Th[i][j+1][k]+Th[i][j-1][k]);
/* Collect source diagonal terms from double
derivatives and forcing terms */
zei_zei_source=a[2][2]*Th[i][j][k]/pow(jac*r,2.0)*sour
ce_k;
eta_zei_source=2*a[2][1]*k_eta_avg/(4.0*pow(jac,2.0)
*g*r)*m3*(source_j_plus-source_j_minus);
psi_zei_source=2*a[2][0]*k_psi_avg/(4.0*pow(jac,2.0)*
h*r)*m3*(source_i_plus-source_i_minus);
force_zei_source=kavg*R(i,j,k)/(2.0*r)*m3*source_k;
/* Compute boundary temperature using source
terms, source diagonals and coefficients */
source=zei_zei_source+eta_zei_source+psi_zei_source
+force_zei_source;
break;
}
}
ret_var=-0.50*delt*source;
/* printf("Source (%d,%d,%d) : %6.3lf \n",i,j,k,source);
*/
return(ret_var);
}


double
boundary_operator(t,c,grid_pt,T,X,Th,conv,mul,i,j,k,n
,m,l)
double t;
int c;
struct point *grid_pt;
double ***T,***X,***Th,***conv,***mul;
int i,j,k,n,m,l;
{
double
coeff,off,source,compute_boundary_coefficient(),comp
ute_boundary_off_diagonal(),combined_boundary_sou
rce(),ret_operator;
double
rho,rxn_mw,kinetics_rate(),MW_power(),density();
extern double rho_f,vol_f,delh;
coeff=compute_boundary_coefficient(t,c,grid_pt,T,X,T
h,conv,mul,i,j,k,n,m,l);
off=compute_boundary_off_diagonal(t,c,grid_pt,T,X,T
h,conv,mul,i,j,k,n,m,l);
ret_operator=(mul[i][j][k]+coeff)*T[i][j][k]+off;
/* printf("Operator(%d,%d,%d) : %6.3lf
\n",i,j,k,ret_operator); */
return(ret_operator);
}


double
calculate_boundary_variable(t,c,grid_pt,T,X,Th,conv,
mul,Sf,i,j,k,n,m,l)
double t;
int c;
struct point *grid_pt;
double ***T,***X,***Th,***conv,***mul,***Sf;
int i,j,k,n,m,l;
{
double
coeff,off,source,compute_boundary_coefficient(),comp
ute_boundary_off_diagonal(),combined_boundary_sou
rce(),var;
double
rho,rxn_mw,kinetics_rate(),MW_power(),density();
extern double rho_f,vol_f,delh;
coeff=mul[i][j][k]+compute_boundary_coefficient(t,c,g
rid_pt,T,X,Th,conv,mul,i,j,k,n,m,l);
off=compute_boundary_off_diagonal(t,c,grid_pt,T,X,T
h,conv,mul,i,j,k,n,m,l);
var=(Sf[i][j][k]-off)/coeff;
/* printf("T(%d,%d,%d) : %6.3lf; X(%d,%d,%d) : %6.3lf
\n",i,j,k,T[i][j][k],i,j,k,X[i][j][k]);
printf("Coeff(%d,%d,%d) : %6.3lf \n",i,j,k,coeff);
printf("Off(%d,%d,%d) : %6.3lf \n",i,j,k,off);
printf("Sf(%d,%d,%d) : %6.3lf \n",i,j,k,Sf[i][j][k]);
printf("T(%d,%d,%d) : %6.3lf \n",i,j,k,var);  */
return(var);
}
```

**Program: properties.c**

This code contains all the routines associated with the calculation of the various properties of the material which are assumed to be functions of temperature and cure. The user who wishes to feed in his or her own functionalities into the routines can do so. Remember that the appropriate input prompts that the program currently requires from the input file (PROPERTY.DAT) should also be modified. Further note that the the lines of the code prompting the program to read the various lines associted with the property inputs present in process_in_out.c should also be modified.

```
#include<stdio.h>
#include<math.h>
/* Calculate density using tempoerature and cure
values at a given point */
double density(Temp,Cure)
double Temp,Cure;
{
extern double vol_f,rho_f,rho_m;
double rf,rm,ret;
rf=rho_f;
rm=rho_m;
ret=(1.0-vol_f)*rm+vol_f*rf;
/* printf("Density : %lf \n",ret); */
return(ret);
}



/* Calculate  specific heat values using temperature
and cure values at a given point */
double specific_heat(Temp,Cure)
double Temp,Cure;
{
extern double rho_f,rho_m,vol_f;
double
den,wf,cpm,cpf,ret,fiber_sp_heat(),matrix_sp_heat();
cpm=matrix_sp_heat(Temp,Cure);
cpf=fiber_sp_heat(Temp,Cure);
den=density(Temp,Cure);
wf=rho_f*vol_f/den;
ret=cpm*(1.0-wf)+wf*cpf;
/* printf("Sp. Heat : %lf \n",ret); */
return(ret);
}



/* These are specific functions depending upon the
property to charactersic correspondence for the
material and can be altered by the user depending
upon his property predictor */
double fiber_sp_heat(T,X)
double T,X;
{
extern double Cp_f;
double ret;
ret=Cp_f;
return(ret);
}

double matrix_sp_heat(T,X)
double T,X;
{
extern double Cp_m;
double ret;
ret=Cp_m;
return(ret);
}



double matrix_conductivity(T,X)
double T,X;
{
extern double km;
double ret;
ret=km;
return(ret);
}



double fiber_conductivity(T,X)
double T,X;
{
extern double kf;
double ret;
ret=kf;
return(ret);
}



double thermal_conductivity(Temp,Cure)
double Temp,Cure;
{
extern double vol_f;
double
thm,thf,matrix_conductivity(),fiber_conductivity(),ret;
thm=matrix_conductivity(Temp,Cure);
thf=fiber_conductivity(Temp,Cure);
ret=thm*(1.0-vol_f)+thf*vol_f;
/* printf("Th. Conductivity : %lf \n",ret); */
return(ret);
}



fill_conductivity(Th,T,X,n,m,l)
double ***Th,***T,***X;
int n,m,l;
{
int i,j,k;
extern double kf,km;
double thermal_conductivity();
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
Th[i][j][k]=thermal_conductivity(T[i][j][k],X[i][j][k]);
}
}
return;
}
```

```
/* This evaluates the product density by specific heat
which is the coefficient of the time derivative */
evaluate_multiplier(ml,T,X,n,m,l)
double ***ml,***T,***X;
int n,m,l;
{
double density(),specific_heat();
int i,j,k;
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
ml[i][j][k]=density(T[i][j][k],X[i][j][k])*specific_heat(T[
i][j][k],X[i][j][k]);
}
}
return;
}
```

**Program: process_in_out.c**

This program reads in all the input data associated with the different process conditions as well as the material parameters for the density, reaction rate equation etc from the appropriate input files.

```
#include<stdio.h>
#include<math.h>
#include<string.h>
#include "grid.h"
/* Read in RXN rate equation data */
read_rxn_data()
{
extern FILE *prop_f;
extern double k1o,k2o,k3o,E1,E2,E3,X_gel,B;
extern double delh;
fscanf(prop_f,"%lf",&k1o);
fscanf(prop_f,"%lf",&E1);
fscanf(prop_f,"%lf",&k2o);
fscanf(prop_f,"%lf",&E2);
fscanf(prop_f,"%lf",&k3o);
fscanf(prop_f,"%lf",&E3);
fscanf(prop_f,"%lf",&X_gel);
fscanf(prop_f,"%lf",&B);
fscanf(prop_f,"%lf",&delh);
return;
}


/* Read in property data */
read_property_data()
{
extern FILE *prop_f;
extern double vol_f;
extern double Cp_f,Cp_m,rho_f,rho_m,kf,km;
fscanf(prop_f,"%lf",&kf);
fscanf(prop_f,"%lf",&km);
fscanf(prop_f,"%lf",&Cp_f);
fscanf(prop_f,"%lf",&Cp_m);
fscanf(prop_f,"%lf",&vol_f);
fscanf(prop_f,"%lf",&rho_f);
fscanf(prop_f,"%lf",&rho_m);
return;
}


/* Read in transfer coefficient data if required */
read_transfer_coeff(n,m,l)
int n,m,l;
{
extern FILE *proc_inp;
extern double ***transfer_coeff;
char bc;
double ret,***mem_alloc_3D();
double val;
int cnt,stp_i,stp_j,stp_k,i,j,k;
transfer_coeff=mem_alloc_3D(n,m,l);
fscanf(proc_inp,"%c",&bc);
for(cnt=0;cnt<3;cnt++)
{
switch(cnt)
{
case 0: {
stp_i=n-1;
stp_j=1;
stp_k=1;
for(i=0;i<n;i=i+stp_i)
{
if(bc=='y')
fscanf(proc_inp,"%lf",&val);
for(j=0;j<m;j=j+stp_j)
{
for(k=0;k<l;k=k+stp_k)
{
if(bc=='y')
transfer_coeff[i][j][k]=val;
else
fscanf(proc_inp,"%lf",&transfer_coeff[i][j][k]);
}
}
}
break;
}
case 1: {
stp_i=1;
stp_j=m-1;
stp_k=1;
for(j=0;j<m;j=j+stp_j)
{
if(bc=='y')
fscanf(proc_inp,"%lf",&val);
for(i=1;i<n-1;i=i+stp_i)
{
for(k=0;k<l;k=k+stp_k)
{
if(bc=='y')
transfer_coeff[i][j][k]=val;
else
fscanf(proc_inp,"%lf",&transfer_coeff[i][j][k]);
}
}
}
break;
}
case 2: {
stp_i=1;
stp_j=1;
```

```
stp_k=l-1;
for(k=0;k<l;k=k+stp_k)
{
if(bc=='y')
fscanf(proc_inp,"%lf",&val);
for(i=1;i<n-1;i=i+stp_i)
{
for(j=1;j<m-1;j=j+stp_j)
{
if(bc=='y')
transfer_coeff[i][j][k]=val;
else
fscanf(proc_inp,"%lf",&transfer_coeff[i][j][k]);
}
}
}
break;
}
}
}
return;
}



/* Read in thermal cure cycle data */
input_thermal_cycle()
{
extern FILE *proc_inp;
extern double **thermal_cycle_data;
double **mem_alloc_2D();
extern int no;
int count;
fscanf(proc_inp,"%d",&no);
thermal_cycle_data=mem_alloc_2D(no+1,3);
count=0;
while(count<=no)
{
fscanf(proc_inp,"%lf",&thermal_cycle_data[count][0]);
fscanf(proc_inp,"%lf",&thermal_cycle_data[count][2]);
if(count==0)
thermal_cycle_data[count][1]=0;
else
thermal_cycle_data[count][1]=(thermal_cycle_data[co
unt][2]-thermal_cycle_data[count-
1][2])/(thermal_cycle_data[count][0]-
thermal_cycle_data[count-1][0]);
count++;
}
return;
}



/* Read in process conditions */
read_process_input(grid_pt,n,m,l)
struct point *grid_pt;
int n,m,l;
{
extern FILE *proc_inp,*prop_f,*numal_input;
extern double final_time,delt;
extern double T_initial,X_initial;
extern int nt0,nt1,nt2,bound_condition;
read_property_data();
read_rxn_data();
fclose(prop_f);
read_transfer_coeff(n,m,l);
input_thermal_cycle();
fscanf(proc_inp,"%lf",&T_initial);
fscanf(proc_inp,"%lf",&X_initial);
fscanf(proc_inp,"%d",&bound_condition);
fclose(proc_inp);
fscanf(numal_input,"%lf",&final_time);
fscanf(numal_input,"%lf",&delt);
fscanf(numal_input,"%d",&nt0);
fscanf(numal_input,"%d",&nt1);
fscanf(numal_input,"%d",&nt2);
fclose(numal_input);
return;
}



/* Read in plotting options specified by user */
read_profile_plot_options(n,m,l)
int n,m,l;
{
extern FILE *disp_f;
extern int psi,eta,zei;
extern char plot_choice[10];
extern double t_display;
extern char map[5],ttx[5];
fscanf(disp_f,"%lf",&t_display);
fscanf(disp_f,"%d",&psi);
fscanf(disp_f,"%d",&eta);
fscanf(disp_f,"%d",&zei);
fscanf(disp_f,"%s",ttx);
fscanf(disp_f,"%s",map);
fscanf(disp_f,"%s",plot_choice);
fclose(disp_f);
return;
}
```

**Program : process_conditions.c**

This program contains all the routines used for calculating the various process parameters such as heat transfer coefficient, the thermal cure cycle etc.

```
#include<stdio.h>
#include<math.h>
double thermal_cycle(t)
double t;
{
extern double **thermal_cycle_data;
extern int no;
double ret;
int i,found;
i=0;
found=0;
while((i<no)&&(!found))
{
if((t<thermal_cycle_data[i+1][0])&&(t>=thermal_cycl
e_data[i][0]))
{
ret=(t-
thermal_cycle_data[i][0])*thermal_cycle_data[i+1][1]
+thermal_cycle_data[i][2];
found=1;
}
i++;
```

```
}
if(found==0)
ret=(t-
thermal_cycle_data[no][0])*thermal_cycle_data[no][1]
+thermal_cycle_data[no][2];
if(ret<=0)
ret=thermal_cycle_data[no][2];
return(ret);
}



Initial_Condition(T,X,n,m,l)
double ***T,***X;
int n,m,l;
{
extern double T_initial,X_initial;
int i,j,k;
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
T[i][j][k]=T_initial;
X[i][j][k]=X_initial;
}
}
}
return;
}



double heat_transfer_coeff(i,j,k)
int i,j,k;
{
extern double ***transfer_coeff;
double ret;
ret=transfer_coeff[i][j][k];
return(ret);
}
```

**Program : power_model.c**

```
/* Function tocalculate MW power given temperature
and cure (currently zero) */
double MW_power(T,X)
double T,X;
{
double ret;
ret=0.00;
return(ret);
}
```

**Program: solve_cure.c**

This program calculates the reactiona rate as a function of temperature and cure and also calculates the extent of cure using an explicit RK method of order 4.

```
#include<stdio.h>
#include<math.h>
#define gas_const 8.314
/* Function which calculates the rate of rxn using a
given rate equation (should be replaced within this
routine when using a different rate equation ) */
double kinetics_rate(T,X)
double T,X; /* Temperature and cure */
{
extern double k1o,k2o,k3o,E1,E2,E3,X_gel,B;
double ret,k1,k2,k3,k4;
if(X>1.000)
ret=0.00;
else
{
k1=k1o*exp(-E1/(gas_const*T));
k2=k2o*exp(-E2/(gas_const*T));
k3=k3o*exp(-E3/(gas_const*T));
if(X<X_gel)
ret=(k1+k2*X)*(1.0-X)*(B-X);
else
ret=k3*(1.0-X);
}
return(ret);
}

/* Solve for the kinetics or a given ODE using RK
method of 4th order */
double solve_kinetics(Temp,Cure,Tg,Xg)
double Temp,Cure,Tg,Xg; /* Temperature, Cure and
the previous time step values of the same */
{
extern double delt;
double ret,k1,k2,k3,k4;
k1=delt*0.50*(kinetics_rate(Temp,Cure)+kinetics_rat
e(Tg,Xg));
k2=delt*0.50*(kinetics_rate(Temp,Cure)+kinetics_rat
e(Tg,Xg+0.5*k1));
k3=delt*0.50*(kinetics_rate(Temp,Cure)+kinetics_rat
e(Tg,Xg+0.5*k2));
k4=delt*0.50*(kinetics_rate(Temp,Cure)+kinetics_rat
e(Tg,Xg+k3));
ret=Xg+1.0/6.0*(k1+2*k2+2*k3+k4);
if(ret>1.0000)
ret=1.0000;
return(ret);
}
```

**Program : shape_in_out.c**

The following set of routines are involved in the input and manipulation of surface coordinate information depending upon the availability of coordinate data (either surface or edge information).

```
#include<stdio.h>
#include<math.h>
#include<string.h>
#include<malloc.h>
/* Read in surface by surface information of the
coordinates of the body */
read_input(ch,fx,fy,fz)
int ch;
double ***fx,***fy,***fz;
{
extern int N,M,K;
extern FILE *inp_f;
extern double scale_x,scale_y,scale_z;
int i,j,k,cnt;
double ***mem_alloc_3D();
if(ch==2)
{
/* Read in surface values for x,y,z*/
/* Psi - constant surfaces in transformed coordinates*/
/* Psi = 0 */
for(j=0;j<M;j++)
{
for(k=0;k<K;k++)
{
fscanf(inp_f,"%lf%lf%lf",&fx[0][j][k],&fy[0][j][k],&fz[0]
[j][k]);
fx[0][j][k]=scale_x*fx[0][j][k];
fy[0][j][k]=scale_y*fy[0][j][k];
fz[0][j][k]=scale_z*fz[0][j][k];
}
}
/* Psi = N-1 */
for(j=0;j<M;j++)
{
for(k=0;k<K;k++)
{
i=N-1;
fscanf(inp_f,"%lf%lf%lf",&fx[N-1][j][k],&fy[N-
1][j][k],&fz[N-1][j][k]);
fx[i][j][k]=scale_x*fx[N-1][j][k];
fy[i][j][k]=scale_y*fy[N-1][j][k];
fz[i][j][k]=scale_z*fz[N-1][j][k];
}
}
/* Eta - constant surfaces in transformed
coordinates*/
/* Eta = 0 */
for(i=1;i<N-1;i++)
{
for(k=0;k<K;k++)
{
j=0;
fscanf(inp_f,"%lf%lf%lf",&fx[i][0][k],&fy[i][0][k],&fz[i][
0][k]);
fx[i][j][k]=scale_x*fx[i][0][k];
fy[i][j][k]=scale_y*fy[i][0][k];
fz[i][j][k]=scale_z*fz[i][0][k];
}
/* Eta = M-1 */
for(i=1;i<N-1;i++)
{
for(k=0;k<K;k++)
{
j=M-1;
fscanf(inp_f,"%lf%lf%lf",&fx[i][M-1][k],&fy[i][M-
1][k],&fz[i][M-1][k]);
fx[i][j][k]=scale_x*fx[i][M-1][k];
fy[i][j][k]=scale_y*fy[i][M-1][k];
fz[i][j][k]=scale_z*fz[i][M-1][k];
}
}
/* Zei - constant surfaces in transformed coordinates*/
/* Zei =0 */
for(i=1;i<N-1;i++)
{
for(j=1;j<M-1;j++)
{
k=0;
fscanf(inp_f,"%lf%lf%lf",&fx[i][j][0],&fy[i][j][0],&fz[i][j]
[0]);
fx[i][j][k]=scale_x*fx[i][j][0];
fy[i][j][k]=scale_y*fy[i][j][0];
fz[i][j][k]=scale_z*fz[i][j][0];
}
}
for(i=1;i<N-1;i++)
/* Zei = K-1*/
{
for(j=1;j<M-1;j++)
{
k=K-1;
fscanf(inp_f,"%lf%lf%lf",&fx[i][j][K-1],&fy[i][j][K-
1],&fz[i][j][K-1]);
fx[i][j][k]=scale_x*fx[i][j][K-1];
fy[i][j][k]=scale_y*fy[i][j][K-1];
fz[i][j][k]=scale_z*fz[i][j][K-1];
}
}
}
else
{
/* Otherwise read in boundary edge values */
/* Psi - constant surfaces in transformed coordinates*/
/* Psi = 0 */
for(j=0;j<M;j=j+M-1)
{
for(k=0;k<K;k++)
{
i=0;
fscanf(inp_f,"%lf%lf%lf",&fx[0][j][k],&fy[0][j][k],&fz[0]
[j][k]);
fx[i][j][k]=scale_x*fx[0][j][k];
fy[i][j][k]=scale_y*fy[0][j][k];
fz[i][j][k]=scale_z*fz[0][j][k];
}
}
for(k=0;k<K;k=k+K-1)
{
for(j=1;j<M-1;j=j++)
{
i=0;
```

```
fscanf(inp_f,"%lf%lf%lf",&fx[0][j][k],&fy[0][j][k],&fz[0]
[j][k]);
fx[i][j][k]=scale_x*fx[0][j][k];
fy[i][j][k]=scale_y*fy[0][j][k];
fz[i][j][k]=scale_z*fz[0][j][k];
}
}
for(j=0;j<M;j=j+M-1)
{
for(k=0;k<K;k++)
{
i=N-1;
fscanf(inp_f,"%lf%lf%lf",&fx[N-1][j][k],&fy[N-
1][j][k],&fz[N-1][j][k]);
fx[i][j][k]=scale_x*fx[N-1][j][k];
fy[i][j][k]=scale_y*fy[N-1][j][k];
fz[i][j][k]=scale_z*fz[N-1][j][k];
}
}
for(k=0;k<K;k=k+K-1)
{
for(j=1;j<M-1;j++)
{
i=N-1;
fscanf(inp_f,"%lf%lf%lf",&fx[N-1][j][k],&fy[N-
1][j][k],&fz[N-1][j][k]);
fx[i][j][k]=scale_x*fx[N-1][j][k];
fy[i][j][k]=scale_y*fy[N-1][j][k];
fz[i][j][k]=scale_z*fz[N-1][j][k];
}
}
/* Eta - constant surfaces in transformed
coordinates*/
/* Eta = 0 */
for(k=0;k<K;k=k+K-1)
{
for(i=1;i<N-1;i++)
{
j=0;
fscanf(inp_f,"%lf%lf%lf",&fx[i][0][k],&fy[i][0][k],&fz[i][
0][k]);
fx[i][j][k]=scale_x*fx[i][0][k];
fy[i][j][k]=scale_y*fy[i][0][k];
fz[i][j][k]=scale_z*fz[i][0][k];
}
}
/* Eta = M-1 */
for(k=0;k<K;k=k+K-1)
{
for(i=1;i<N-1;i++)
{
j=M-1;
fscanf(inp_f,"%lf%lf%lf",&fx[i][M-1][k],&fy[i][M-
1][k],&fz[i][M-1][k]);
fx[i][j][k]=scale_x*fx[i][M-1][k];
fy[i][j][k]=scale_y*fy[i][M-1][k];
fz[i][j][k]=scale_z*fz[i][M-1][k];
}
}
}
return;
}

/* The following are debugging routines which display
values of the shape coordinates in different formats */
print_out(n_f,fx,fy,fz,n,m,l) /* Print out INTo a
Matlab program file to create a 3D wire plot of the
object */
FILE *n_f;
double ***fx,***fy,***fz;
int n,m,l;
{
int i,j,k;
fprintf(n_f,"clear;\nhold off;\n clg;\n");
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
fprintf(n_f,"matij%d%d=[",i,j);
for(k=0;k<l;k++)
{
fprintf(n_f,"          %6.4lf          %6.4lf
%6.4lf\n",fx[i][j][k],fy[i][j][k],fz[i][j][k]);
}
fprintf(n_f,"];\n");
fprintf(n_f,"plot3(matij%d%d(:,1),matij%d%d(:,2),mati
j%d%d(:,3),'w');\n",i,j,i,j,i,j);
if((i==0)&&(j==0))
fprintf(n_f,"hold;\n");
}
}
/* for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
fprintf(n_f,"matjk%d%d=[",j,k);
for(i=0;i<n;i++)
{
fprintf(n_f,"          %6.4lf          %6.4lf
%6.4lf\n",fx[i][j][k],fy[i][j][k],fz[i][j][k]);
}
fprintf(n_f,"];\n");
fprintf(n_f,"plot3(matjk%d%d(:,1),matjk%d%d(:,2),ma
tjk%d%d(:,3),'m');\n",j,k,j,k,j,k);
}
}
for(i=0;i<n;i++)
{
for(k=0;k<l;k++)
{
fprintf(n_f,"matik%d%d=[",i,k);
for(j=0;j<m;j++)
{
fprintf(n_f,"          %6.4lf          %6.4lf
%6.4lf\n",fx[i][j][k],fy[i][j][k],fz[i][j][k]);
}
fprintf(n_f,"];\n");
fprintf(n_f,"plot3(matik%d%d(:,1),matik%d%d(:,2),ma
tik%d%d(:,3),'y');\n",i,k,i,k,i,k);
}
}
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
fprintf(n_f,"matij%d%d=[",i,j);
for(k=0;k<l;k++)
```

```
fprintf(n_f,"                %6.4lf                %6.4lf
%6.4lf\n",fx[i][j][k],fy[i][j][k],fz[i][j][k]);
fprintf(n_f,"];\n");
fprintf(n_f,"plot3(matij%d%d(:,1),matij%d%d(:,2),mati
j%d%d(:,3),'b');\n",i,j,i,j,i,j);
}
}
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
fprintf(n_f,"matjk%d%d=[",j,k);
for(i=0;i<n;i++)
fprintf(n_f,"                %6.4lf                %6.4lf
%6.4lf\n",fx[i][j][k],fy[i][j][k],fz[i][j][k]);
fprintf(n_f,"];\n");
fprintf(n_f,"plot3(matjk%d%d(:,1),matjk%d%d(:,2),ma
tjk%d%d(:,3),'g');\n",j,k,j,k,j,k);
}
}*/
return;
}



print_input(v_f,fx,fy,fz,n,m,l)
FILE *v_f;
double ***fx,***fy,***fz;
int n,m,l;
{
int i,j,k;
for(j=0;j<m;j++)
{
fprintf(v_f,"mat0i%d=[",j);
for(k=0;k<l;k++)
fprintf(v_f,"%lf                %lf                %lf
\n",fx[0][j][k],fy[0][j][k],fz[0][j][k]);
fprintf(v_f,"];\n");
fprintf(v_f,"plot3(mat0i%d(:,1),mat0i%d(:,2),mat0i%d(:
,3),'y');\n",j,j,j);
if(j==0)
fprintf(v_f,"hold on;\n");
}
/* Psi = n-1 */
for(j=0;j<m;j++)
{
fprintf(v_f,"mat%di%d=[",n-1,j);
for(k=0;k<l;k++)
fprintf(v_f,"%lf    %lf    %lf    \n",fx[n-1][j][k],fy[n-
1][j][k],fz[n-1][j][k]);
fprintf(v_f,"];\n");
fprintf(v_f,"plot3(mat%di%d(:,1),mat%di%d(:,2),mat%
di%d(:,3),'w');\n",n-1,j,n-1,j,n-1,j);
}
/*  Eta  -  constant  surfaces  in  transformed
coordinates*/
/* Eta = 0 */
for(i=0;i<n;i++)
{
fprintf(v_f,"mat0j%d=[",i);
for(k=0;k<l;k++)
fprintf(v_f,"%lf                %lf                %lf
\n",fx[i][0][k],fy[i][0][k],fz[i][0][k]);
fprintf(v_f,"];\n");
fprintf(v_f,"plot3(mat0j%d(:,1),mat0j%d(:,2),mat0j%d(:
,3),'m');\n",i,i,i);
}
/* Eta = m-1 */
for(i=0;i<n;i++)
{
fprintf(v_f,"mat%dj%d=[",m-1,i);
for(k=0;k<l;k++)
fprintf(v_f,"%lf    %lf    %lf    \n",fx[i][m-1][k],fy[i][m-
1][k],fz[i][m-1][k]);
fprintf(v_f,"];\n");
fprintf(v_f,"plot3(mat%dj%d(:,1),mat%dj%d(:,2),mat%
dj%d(:,3),'w');\n",m-1,i,m-1,i,m-1,i);
}
/* Zei - constant surfaces in transformed coordinates*/
/* Zei =0 */
for(i=0;i<n;i++)
{
fprintf(v_f,"mat0k%d=[",i);
for(j=0;j<m;j++)
fprintf(v_f,"%lf                %lf                %lf
\n",fx[i][j][0],fy[i][j][0],fz[i][j][0]);
fprintf(v_f,"];\n");
fprintf(v_f,"plot3(mat0k%d(:,1),mat0k%d(:,2),mat0k%
d(:,3),'m');\n",i,i,i);
}
for(i=0;i<n;i++)
/* Zei = l-1*/
{
fprintf(v_f,"mat%dk%d=[",l-1,i);
for(j=0;j<m;j++)
fprintf(v_f,"%lf    %lf    %lf    \n",fx[i][j][l-1],fy[i][j][l-
1],fz[i][j][l-1]);
fprintf(v_f,"];\n");
fprintf(v_f,"plot3(mat%dk%d(:,1),mat%dk%d(:,2),mat
%dk%d(:,3),'m');\n",l-1,i,l-1,i,l-1,i);
}
return;
}
```

The following set of programs are for the conversion of C data structures into Matlab matrices as well as using inbuilt matlab functions to plot data, display surfaces and color maps of various profiles.

**Program: plot_surfaces.c**

```
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include<string.h>
#include<malloc.h>
#include "engine.h"
/* Plot the shape of the object and additionally
provide the color map using dependent variable
values if available */
plot_surf(p,xp,yp,zp,c,n,m,l)
int p;
double ***xp,***yp,***zp,***c;
int n,m,l;
{
Matrix *x,*y,*z,*d,*df;
double *dr;
extern Engine *eptr;
```

```
double *rl,*img;
int i,j,k,s,p1,total;
double
di,dj,dk,x_min,y_min,z_min,x_max,y_max,z_max;
double
*tempc,*tempx,*tempy,*tempz,find_max(),find_min();
char
*erg,str[50],str1[50],con[50],err[50],mat_comm[200],fi
g_name[75];
total=n*m*l;
if((p==1)||(p==2))
{
di=(double) n;
dj=(double) m;
dk=(double) l;
tempz=(double *)malloc(total*sizeof(double));
tempx=(double *)malloc(total*sizeof(double));
tempy=(double *)malloc(total*sizeof(double));
mat_set_full(xp,tempx,n,m,l);
mat_set_full(yp,tempy,n,m,l);
mat_set_full(zp,tempz,n,m,l);
x_min=find_min(xp,n,m,l);
y_min=find_min(yp,n,m,l);
z_min=find_min(zp,n,m,l);
x_max=find_max(xp,n,m,l);
y_max=find_max(yp,n,m,l);
z_max=find_max(zp,n,m,l);
engPutFull(eptr,"n",1,1,&di,NULL);
engPutFull(eptr,"m",1,1,&dj,NULL);
engPutFull(eptr,"l",1,1,&dk,NULL);
engPutFull(eptr,"xmx",1,1,&x_max,NULL);
engPutFull(eptr,"ymx",1,1,&y_max,NULL);
engPutFull(eptr,"zmx",1,1,&z_max,NULL);
engPutFull(eptr,"xmn",1,1,&x_min,NULL);
engPutFull(eptr,"ymn",1,1,&y_min,NULL);
engPutFull(eptr,"zmn",1,1,&z_min,NULL);
engPutFull(eptr,"x",total,1,tempx,NULL);
engPutFull(eptr,"y",total,1,tempy,NULL);
engPutFull(eptr,"z",total,1,tempz,NULL);
}
if(p==1)
engEvalString(eptr,"figure(1);");
if((p==1)||(p==2))
{
if(p==2)
engEvalString(eptr,"figure(find_pic('Shape
Modelling'));");
sprintf(con,"subplot(1,2,%d);",p);
engEvalString(eptr,con);
engEvalString(eptr,"hold off;");
engEvalString(eptr,"shape_plot(x,y,z,n,m,l,1);");
if(p==1)
{
engEvalString(eptr,"title('Initial Guess');");
sprintf(mat_comm,"set(gcf,'name','Shape
Modelling');");
engEvalString(eptr,mat_comm);
}
else
engEvalString(eptr,"title('Boundary Fitted Object');");
free((char *)tempx);
free((char *)tempy);
free((char *)tempz);
}
else
{
tempc=(double *)malloc(n*m*l*sizeof(double));
mat_set_full(c,tempc,n,m,l);
engPutFull(eptr,"c",n*m*l,1,tempc,NULL);
engEvalString(eptr,"shape_plot(x,y,z,n,m,l,1,c);");
engEvalString(eptr,"shading interp;");
free((char *)tempc);
}
return;
}


/* Set up the three dimensional structures into
matlab vectors for plotting */
mat_set_full(p,temp,n,m,l)
double ***p;
double *temp;
int n,m,l;
{
int r,i,j,k;
r=0;
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
temp[r]=p[i][j][k];
r++;
}
}
}
return;
}


/* Set up viewing options for the user. Also provides
section wise cutting options for given shape along
Psi, Eta and Zei specified within the grid point
limits. Provides options to use matlab commands to
manipulate views for the expert matlab user */
view_options(x,y,z,c,n,m,l,b)
double ***x,***y,***z,***c;
int n,m,l,b;
{
extern Engine *eptr;
char
yn[5],*err,mat_comm[200],fig_name[75],pass[20],psi_
ar[30],eta_ar[30],zei_ar[30];
int surf,ch,cnti,cntj,cntk,fig,count;
fig=1;
count=1;
cnti=cntj=cntk=1;
printf("Which surface do you want to view
<1.Psi/2.Eta/3.Zei/4.None> : ");
scanf("%d",&ch);
while(ch!=4)
{
if(count==1)
{
engEvalString(eptr,"figure;");
engEvalString(eptr,"hi=gcf;");
```

```
if(c==(double ***)NULL)
{
if(b==1)
sprintf(mat_comm,"set(gcf,'name','Surface Plots-1');");
else
sprintf(mat_comm,"set(gcf,'name','Surface Plots-2');");
}
else
{
if(b==1)
sprintf(mat_comm,"set(gcf,'name','Surface
Temperature Map');");
else
sprintf(mat_comm,"set(gcf,'name','Surface Cure
Map');");
}
engEvalString(eptr,mat_comm);
count++;
}
switch(ch)
{
case 1:{
printf("Psi = ");
scanf("%d",&surf);
engEvalString(eptr,"figure(hi);");
engEvalString(eptr,"subplot(1,3,1);");
engEvalString(eptr,"hold off;");
plot_surface(1,x,y,z,c,n,m,l,surf);
engEvalString(eptr,"view(90,20);");
engEvalString(eptr,"axis off;");
sprintf(psi_ar,"title('u = %d plane');",surf);
engEvalString(eptr,psi_ar);
engEvalString(eptr,"end;");
engEvalString(eptr,"pause;");
cnti++;
break;
}
case 2: {
printf("Eta = ");
scanf("%d",&surf);
engEvalString(eptr,"figure(hi);");
engEvalString(eptr,"subplot(1,3,2);");
engEvalString(eptr,"hold off;");
plot_surface(2,x,y,z,c,n,m,l,surf);
engEvalString(eptr,"view(100,30);");
engEvalString(eptr,"axis off;");
sprintf(eta_ar,"title('v = %d plane');",surf);
engEvalString(eptr,eta_ar);
engEvalString(eptr,"end;");
engEvalString(eptr,"pause;");
cntj++;
break;
}
case 3: {
printf("Zei = ");
scanf("%d",&surf);
engEvalString(eptr,"figure(hi);");
engEvalString(eptr,"subplot(1,3,3);");
engEvalString(eptr,"hold off;");
plot_surface(3,x,y,z,c,n,m,l,surf);
engEvalString(eptr,"view(3);");
engEvalString(eptr,"axis off;");
sprintf(zei_ar,"title('w = %d plane');",surf);
engEvalString(eptr,zei_ar);
engEvalString(eptr,"end;");
engEvalString(eptr,"pause;");
cntk++;
break;
}
}
printf("Which surface do you want to view
<1.Psi/2.Eta/3.Zei/4.None> : ");
scanf("%d",&ch);
}
mat_lab_options();
return;
}



/* Plot a single surface/coordinate-section of the shape
*/
plot_surface(p,x,y,z,c,n,m,l,i) /* To plot the section
requested */
int p; /* Psi, Eta or Zei surface index (1, 2 or 3 resptly)
*/
double ***x,***y,***z,***c; /* Shape and
Temperature matrices */
int n,m,l,i; /* Size of plane and corresponding index
for 3rd coordinate */
{
extern Engine *eptr; /* Matlab Workspace */
double dp,di,*tempx,*tempy,*tempz,*tempc;
int j,k;
dp=(double)p;
di=(double)(i+1);
engPutFull(eptr,"index",1,1,&dp,NULL);
engPutFull(eptr,"i",1,1,&di,NULL);
if(c==(double ***) NULL)
engEvalString(eptr,"surface_plot(x,y,z,n,m,l,i,index);"
);
else
{
tempc=(double *)malloc(n*m*l*sizeof(double));
mat_set_full(c,tempc,n,m,l);
engPutFull(eptr,"c",n*m*l,1,tempc,NULL);
engEvalString(eptr,"surface_plot(x,y,z,n,m,l,i,index,c)
;");
engEvalString(eptr,"shading interp;");
free((char *)tempc);
}
engEvalString(eptr,"clear c;");
return;
}



/* Matlab options routine. Acts as conduit between
the user input at terminal to the matlab software
workspace */
mat_lab_options()
{
extern Engine *eptr;
char yn[3],*err;
err=(char *)malloc(5*sizeof(char));
printf("Do you want to execute Matlab commands
(y/n) :");
scanf("%s",err);
```

```
while((strcmp(err,"quit")!=0)&&(strcmp(err,"n")!=0)&
&(strcmp(err,"quit;")!=0))
{
free((char *)err);
err=(char *)malloc(75*sizeof(char));
printf("Command : ");
scanf("%s",err);
if((strcmp(err,"quit")!=0)&&(strcmp(err,"quit;")!=0))
{
engEvalString(eptr,err);
engEvalString(eptr,"figure(gcf);");
}
}
free((char *)err);
return;
}


print_plot_options()
{
char yn[3],*mat_comm1;
int fig_no;
mat_comm1=(char *)malloc(60*sizeof(char));
printf("Do you want any plots to be saved ? <y/n>:");
scanf("%s",yn);
while(strcmp(yn,"y")==0)
{
printf("Figure No. for which hard copy is required ?
");
scanf("%d",&fig_no);
sprintf(mat_comm1,"figure(%d);print
Figure%d",fig_no,fig_no);
engEvalString(eptr,mat_comm1);
printf("Do you want any other figure to be saved ?
<y/n>:");
scanf("%s",yn);
}
free(mat_comm1);
return;
}
```

**Program : plot_profiles.c**

```
#include<stdio.h>
#include<math.h>
#include<string.h>
#include<malloc.h>
#include "grid.h"
#include "engine.h"
/* Graphical display of the required variables as 3D
plots along two directions, color maps indicating
trends and time plots at different points as specified
in the 'init_display.m' matlab file */
Display_variables(t,grid_pt,T,X,n,m,l)
double t; /* Time */
struct point *grid_pt; /* Grid point structure
containing physical coordinate information */
double ***T,***X; /* Temperature and Cure variables
*/
int n,m,l; /* Mesh size */
{
extern int psi,eta,zei;
int no_fig,cnt;
extern char plot_choice[10];
extern char map[5],ttx[5];
extern Engine *eptr;
extern double t_display;
double *tempT,*tempX,***x,***y,***z;
cnt=0;
if(psi!=-1)
cnt++;
if(eta!=-1)
cnt++;
if(zei!=-1)
cnt++;
no_fig=cnt;
/* Plot Temperature maps and profiles */
if((no_fig!=0)&&((strcmp(plot_choice,"Temp")==0)||(
strcmp(plot_choice,"Both")==0)))
{
if(strcmp(map,"yes")==0)
{
if(t<=1.0e-8)
{
engEvalString(eptr,"figure;set(gcf,'name','Temperatu
re Profiles');");
plot_graph(t,no_fig,grid_pt,T,n,m,l);
engEvalString(eptr,"figure;set(gcf,'name','Temperatu
re Map');");
plot_map(t,grid_pt,T,n,m,l,1);
}
else
{
engEvalString(eptr,"figure(find_pic('Temperature
Profiles'));");
plot_graph(t,no_fig,grid_pt,T,n,m,l);
engEvalString(eptr,"figure(find_pic('Temperature
Map'));");
plot_map(t,grid_pt,T,n,m,l,1);
}
}
/* Update time-temperature plot */
if(strcmp(ttx,"yes")==0)
update_time_plot(t,T,n,m,l,"T");
}
/* Plot Cure maps and profiles */
if((no_fig!=0)&&((strcmp(plot_choice,"Cure")==0)||(s
trcmp(plot_choice,"Both")==0)))
{
if(strcmp(map,"yes")==0)
{
if(t<=1.0e-8)
{
engEvalString(eptr,"figure(gcf+1);set(gcf,'name','Cure
Profiles');");
plot_graph(t,no_fig,grid_pt,X,n,m,l);
engEvalString(eptr,"figure;set(gcf,'name','Cure
Map');");
plot_map(t,grid_pt,X,n,m,l,2);
}
else
{
engEvalString(eptr,"figure(find_pic('Cure
Profiles'));");
plot_graph(t,no_fig,grid_pt,X,n,m,l);
engEvalString(eptr,"figure(find_pic('Cure Map'));");
plot_map(t,grid_pt,X,n,m,l,2);
}
}
/* Update time-cure plots */
```

```
if(strcmp(ttx,"yes")==0)
update_time_plot(t,X,n,m,l,"X");
}
return;
}



/* Graphical output of Temperature or Cure Vs
(Psi,Eta),(Eta,Zei) and (Zei,Psi) directions */
plot_graph(t,no_fig,grid_pt,f,n,m,l)
double t;
int no_fig;
struct point *grid_pt;
double ***f;
int n,m,l;
{
extern int psi,eta,zei;
extern Engine *eptr;
double *temp;
char str[50];
int j;
j=1;
if((psi!=-1))
{
sprintf(str,"subplot(1,%d,%d);",no_fig,j);
engEvalString(eptr,str);
plot_prof(t,j,psi,f,n,m,l);
engEvalString(eptr,"pause;");
j++;
}
if(eta!=-1)
{
sprintf(str,"subplot(1,%d,%d);",no_fig,j);
engEvalString(eptr,str);
plot_prof(t,j,eta,f,n,m,l);
engEvalString(eptr,"pause;");
j++;
}
if(zei!=-1)
{
sprintf(str,"subplot(1,%d,%d);",no_fig,j);
engEvalString(eptr,str);
plot_prof(t,j,zei,f,n,m,l);
engEvalString(eptr,"pause;");
}
return;
}



plot_prof(t,ch,i,f,n,m,l)
double t;
int ch,i;
double ***f;
int n,m,l;
{
extern Engine *eptr;
int s,j,k;
char str[50],com[50];
double *temp,***mem_alloc_3D(),di;
di=(double) i;
temp=(double *)malloc(n*m*l*sizeof(double));
mat_set_full(f,temp,n,m,l);
engPutFull(eptr,"P",n*m*l,1,temp,NULL);
engPutFull(eptr,"I",1,1,&di,NULL);
switch(ch)
{
case 1: {
engEvalString(eptr,"plot_xy(P,1,I,n,m,l);");
sprintf(str,"title('Psi=%d;Time:%6.1lf units');",i,t);
engEvalString(eptr,"xlabel('Eta');ylabel('Zei');zlabel('T
');");
break;
}
case 2: {
engEvalString(eptr,"plot_xy(P,2,I,n,m,l);");
sprintf(str,"title('Eta=%d;Time:%6.1lf units');",i,t);
engEvalString(eptr,"xlabel('Psi');ylabel('Zei');zlabel('T'
);");
break;
}
case 3: {
engEvalString(eptr,"plot_xy(P,3,I,n,m,l);");
sprintf(str,"title('Zei=%d;Time:%6.1lf units');",i,t);
engEvalString(eptr,"xlabel('Psi');ylabel('Eta');zlabel('T
');");
break;
}
}
engEvalString(eptr,str);
engEvalString(eptr,"clear P;");
free((char *)temp);
return;
}



/* Plot the color map of Temperature and or Cure for
entire shape */
plot_map(t,grid_pt,T,n,m,l,b)
double t;
struct point *grid_pt;
double ***T;
int n,m,l,b;
{
extern Engine *eptr;
char str[100];
engEvalString(eptr,"hold off;");
engEvalString(eptr,"set(gca,'Visible','off');");
engEvalString(eptr,"view(3);");
engEvalString(eptr,"hold on;");
sprintf(str,"title(Temperature Map: Time=%6.1lf
units);",t);
engEvalString(eptr,str);
plot_surf(3,grid_pt->x,grid_pt->y,grid_pt->z,T,n,m,l);
engEvalString(eptr,"hold off;");
/*          view_options(grid_pt->x,grid_pt->y,grid_pt-
>z,T,n,m,l,b); */
return;
}



/* Update the time-temperature and/or time-cure
plots */
update_time_plot(t,f,n,m,l,str)
double t,***f;
int n,m,l;
char *str;
{
```

```
double *temp;
extern Engine *eptr;
temp=(double *)malloc(n*m*l*sizeof(double));
mat_set_full(f,temp,n,m,l);
engPutFull(eptr,"t",1,1,&t,NULL);
engPutFull(eptr,"c",n*m*l,1,temp,NULL);
if(t<=1.0e-8)
{
engEvalString(eptr,"figure;");
if(strcmp(str,"T")==0)
{
engEvalString(eptr,"set(gcf,'name','Time-
Temperature Plots');");
engEvalString(eptr,"time_tplots(t,c,n,m,l);");
engEvalString(eptr,"title('Temperature Vs Time');");
engEvalString(eptr,"xlabel('Time ---->');");
engEvalString(eptr,"ylabel('Temperature ----->');");
}
else
{
engEvalString(eptr,"set(gcf,'name','Time-Cure
Plots');");
engEvalString(eptr,"time_xplots(t,c,n,m,l);");
engEvalString(eptr,"title('Cure Vs Time');");
engEvalString(eptr,"xlabel('Time ---->');");
engEvalString(eptr,"ylabel('Cure ----->');");
engEvalString(eptr,"axis([0 t 0 1]);");
}
}
else
{
if(strcmp(str,"T")==0)
{
engEvalString(eptr,"figure(find_pic('Time-
Temperature Plots'));");
engEvalString(eptr,"time_tplots(t,c,n,m,l);");
engEvalString(eptr,"axis([0 t Tin max(c)]);");
}
else
{
engEvalString(eptr,"figure(find_pic('Time-Cure
Plots'));");
engEvalString(eptr,"time_xplots(t,c,n,m,l);");
engEvalString(eptr,"axis([0 t 0 1]);");
}
}
engEvalString(eptr,"clear c;");
free(temp);
return;
}



initialize_workspace(grid_pt,n,m,l)
struct point *grid_pt;
int n,m,l;
{
extern Engine *eptr;
extern double final_time,t_display;
extern double T_initial,X_initial;
double *temp;
engEvalString(eptr,"init_display;");
temp=(double *)malloc(n*m*l*sizeof(double));
mat_set_full(grid_pt->x,temp,n,m,l);
engPutFull(eptr,"x",n*m*l,1,temp,NULL);
mat_set_full(grid_pt->y,temp,n,m,l);
engPutFull(eptr,"y",n*m*l,1,temp,NULL);
mat_set_full(grid_pt->z,temp,n,m,l);
engPutFull(eptr,"z",n*m*l,1,temp,NULL);
engPutFull(eptr,"tfinal",1,1,&final_time,NULL);
engPutFull(eptr,"tdisp",1,1,&t_display,NULL);
engPutFull(eptr,"Tin",1,1,&T_initial,NULL);
engPutFull(eptr,"Xin",1,1,&X_initial,NULL);
free(temp);
return;
}
```

The following set of programs contain general purpose routines on memory management, matrix operations and the various derivative operations performed numerically in space w.r.t natural coordinates.

**Program ; memory.c**

```
#include<stdio.h>
#include<math.h>
#include<malloc.h>
#include "grid.h"
/* Transfinite Interpolation Routine */
transfinite_3D(f,n,m,l)
double ***f;
int n,m,l;
{
double ***F1,***mem_alloc_3D(),int_plt();
double ***F2;
double ***F3;
int i,j,k;
F1=mem_alloc_3D(l,n,m);
/* Correction along j(eta) direction */
for(k=0;k<l;k++)
{
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
F1[k][i][j]=int_plt(k,l,f[i][j][0],f[i][j][l-1]);
}
}
F2=mem_alloc_3D(m,l,n);
/* Correction along i(psi) direction */
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
for(i=0;i<n;i++)
F2[j][k][i]=int_plt(j,m,f[i][0][k]-F1[k][i][0],f[i][m-1][k]-
F1[k][i][m-1]);
}
}
F3=mem_alloc_3D(n,m,l);
/* Correction along k(zei) direction */
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
F3[i][j][k]=int_plt(i,n,f[0][j][k]-F1[k][0][j]-
F2[j][k][0],f[n-1][j][k]-F1[k][n-1][j]-F2[j][k][n-1]);
}
}
/* Algebraic grid is sum of all corrections */
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
f[i][j][k]=F1[k][i][j]+F2[j][k][i]+F3[i][j][k];
}
}
free_3D(F1,l,n,m);
free_3D(F2,m,l,n);
free_3D(F3,n,m,l);
return;
}


/* Cubic spline interpolation (will be linear if only two
end points are given) */
double int_plt(i,n,r1,r2) /* Cubic Spline (Linear if only
6 surfaces are specified in all )
 interpolation */
int i,n;
double r1,r2;
{
double ret;
/* if((i==0)||(i==n-1))   */
ret=r1*(n-1-i)/(double)(n-1)+r2*i/(double)(n-1);
/* else
ret=2.3*((n-1-i)*r1+i*r2)/((double)(i*(n-1)));  */
return(ret);
}


/* Transfinite interpolation along two dimensions .
Will be used if only edge information is given */
transfinite_2D(f,n,m)
double **f;
int n,m;
{
double **f1,**f2,**mem_alloc_2D(),int_plt();
int i,j;
f1=mem_alloc_2D(n,m);
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f1[i][j]=int_plt(i,n,f[0][j],f[n-1][j]);
}
f2=mem_alloc_2D(m,n);
for(j=0;j<m;j++)
{
for(i=0;i<n;i++)
f2[j][i]=int_plt(j,m,f[i][0]-f1[i][0],f[i][m-1]-f1[i][m-1]);
}
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f[i][j]=f1[i][j]+f2[j][i];
}
free_2D(f1,n,m);
free_2D(f2,m,n);
return;
}


/* Approximation of surfaces by successive calling of
transfinite two dimensional routines */
surface_approx(ch,f,n,m,l)
int ch;
double ***f;
int n,m,l;
{
int i,j,k,d;
double **two_d,**mem_alloc_2D();
d=0;
if(ch==1)
```

```
{
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
if(((i==0)&&(j==0))||((i==0)&&(j==m-1))||((i==n-
1)&&(j==0))||((i==n-1)&&(j==m-
1))||((i==0)&&(k==0))||((i==0)&&(k==l-1))||((i==n-
1)&&(k==0))||((i==n-1)&&(k==l-
1))||((j==0)&&(k==0))||((j==m-
1)&&(k==0))||((j==0)&&(k==l-1))||((j==m-
1)&&(k==l-1)))
d++;
else
f[i][j][k]=0.00;
}
}
}
for(i=0;i<3;i++)
{
switch(i)
{
case 0: {
two_d=mem_alloc_2D(m,l);
for(j=0;j<n;j=j+n-1)
{
D3_to_D2(1,f,two_d,j,m,l);
transfinite_2D(two_d,m,l);
D2_to_D3(1,two_d,f,j,m,l);
}
free_2D(two_d,m,l);
break;
}
case 1: {
two_d=mem_alloc_2D(n,l);
for(j=0;j<m;j=j+m-1)
{
D3_to_D2(2,f,two_d,j,n,l);
transfinite_2D(two_d,n,l);
D2_to_D3(2,two_d,f,j,n,l);
}
free_2D(two_d,n,l);
break;
}
case 2: {
two_d=mem_alloc_2D(n,m);
for(j=0;j<l;j=j+l-1)
{
D3_to_D2(3,f,two_d,j,n,m);
transfinite_2D(two_d,n,m);
D2_to_D3(3,two_d,f,j,n,m);
}
free_2D(two_d,n,m);
break;
}
}
}
}
transfinite_3D(f,n,m,l);
return;
}
```

```
/* Convert a three dimensional matrix into two
dimensional matrix containing only surface
information*/
D3_to_D2(c,f1,f2,p,n,m)
int c;
double ***f1,**f2;
int p,n,m;
{
int i,j;
switch(c)
{
case 1: {
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f2[i][j]=f1[p][i][j];
}
break;
}
case 2: {
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f2[i][j]=f1[i][p][j];
}
break;
}
case 3: {
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f2[i][j]=f1[i][j][p];
}
break;
}
}
return;
}


/* Reading back surface information into 3D matrix */
D2_to_D3(c,f1,f2,p,n,m)
int c;
double **f1,***f2;
int p,n,m;
{
int i,j;
switch(c)
{
case 1: {
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f2[p][i][j]=f1[i][j];
}
break;
}
case 2: {
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f2[i][p][j]=f1[i][j];
}
break;
}
```

```
case 3: {
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
f2[i][j][p]=f1[i][j];
}
break;
}
}
return;
}
```

**Program : matrix_operations.c**

```
/* This program contains all operations related to
matrices such as addition, subtraction , initializing a
matrix , copying one matrix into another, finding
minimum and maximum of a given matrix and
finding the norm of a matrix. All these matrices refer
to three dimensional matrices. The matrix_solve
routine in this program refers to a two dimensional
matrix and solves a linear set of simultameous
equations using the row reduction method. The
equation is of the form Ax=b; The matrix supplied is
the augmented matrix A|b.n this case. This routine is
not currently used but if there is  a need to change
the relaxation method from Gauss siedel to solution
of simultaneous equations this routine could be used
provided the variables
are supplied in the right format  */
#include<stdio.h>
#include<math.h>
mat_add(c,a,b,n,m,l)
double ***c,***a,***b;
int n,m,l;
{
int i,j,k;
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
c[i][j][k]=a[i][j][k]+b[i][j][k];
}
}
}
return;
}



mat_sub(c,a,b,n,m,l)
double ***c,***a,***b;
int n,m,l;
{
int i,j,k;
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
c[i][j][k]=a[i][j][k]-b[i][j][k];
}
}
}
return;
}



set_mat_zero(a,n,m,l)
double ***a;
int n,m,l;
{
int i,j,k;
i=0;
while(i<n)
{
j=0;
while(j<m)
{
k=0;
while(k<l)
{
a[i][j][k]=0;
k++;
}
j++;
}
i++;
}
return;
}



mat_copy(f1,f2,n,m,l)
double ***f1,***f2;
int n,m,l;
{
int i,j,k;
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
f1[i][j][k]=f2[i][j][k];
}
}
return;
}



double norm_xyz(fx,fy,fz,n,m,l)
double ***fx,***fy,***fz;
int n,m,l;
{
double norm;
int i,j,k;
norm=0.0;
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
norm=norm+sqrt(pow(fx[i][j][k],2.0)+pow(fy[i][j][k],2.
0)+pow(fz[i][j][k],2.0));
```

```
}
}
norm=norm/(n*m*l);
return(norm);
}



double find_max(f,n,m,l)
double ***f;
int n,m,l;
{
double ret;
int i,j,k;
ret=f[0][0][0];
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
if(ret<=f[i][j][k])
ret=f[i][j][k];
}
}
}
return(ret);
}



double find_abs_max(f,n,m,l)
double ***f;
int n,m,l;
{
double ret;
int i,j,k;
ret=0.00;
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
if(ret<=fabs(f[i][j][k]))
ret=fabs(f[i][j][k]);
}
}
}
return(ret);
}



double find_min(f,n,m,l)
double ***f;
int n,m,l;
{
double ret;
int i,j,k;
ret=f[0][0][0];
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
for(k=0;k<l;k++)
{
if(ret>=f[i][j][k])
ret=f[i][j][k];
}
}
}
return(ret);
}



double norm(f,n,m,l)
double ***f;
int n,m,l;
{
double ret,find_abs_max();
int i,j,k;
ret=find_abs_max(f,n,m,l);
return(ret);
}



/*Matrix Solution*/
matrix_solve(a,m)
double **a;
int m;
{
int i,j,k,l,r;
double temp,t,s;
r=0;
while(r<m)
{
t=a[r][r];
if(abs(t)<=1.0e-4)
{
for(k=r+1;k<=m;k++)
{
if(abs(a[k][r])>1.0e-4)
{
for(l=0;l<=m;l++)
{
temp=a[r][l];
a[r][l]=a[k][l];
a[k][l]=temp;
}
}
}
}
for(k=0;k<m+1;k++)
{
a[r][k]=a[r][k]/t;
}
for(l=0;l<m;l++)
{
if(l!=r)
{
s=a[l][r];
for(k=0;k<m+1;k++)
{
a[l][k]=a[l][k]-s*a[r][k];
}
}
```

```
}
r++;
}
return;
}
```

**Program : derivatives.c**

```
/* All the following routines (from f_psi to f_zei_psi)
  evaluate the appropriate derivates of a 3D function
'f'*/
#include<stdio.h>
#include<math.h>
double f_psi(f,i,j,k,n,m,l,bc) /* Psi derivative */
double ***f;
int i,j,k,n,m,l;
int bc;
{
extern int N;
double fp,h;
h=(double)(N-1.0)/(n-1.0);
if((i>0)&&(i<n-1))
fp=0.50/h*(f[i+1][j][k]-f[i-1][j][k]);
else
{
if(i==0)
fp=1.00/h*(f[i+1][j][k]-f[i][j][k]);
else
fp=1.00/h*(f[i][j][k]-f[i-1][j][k]);
}
return(fp);
}



double f_eta(f,i,j,k,n,m,l,bc) /* Eta Derivative */
double ***f;
int i,j,k,n,m,l;
int bc;
{
double fp,g;
extern int M;
g=(double)(M-1.0)/(m-1.0);
if((j>0)&&(j<m-1))
fp=0.50/g*(f[i][j+1][k]-f[i][j-1][k]);
else
{
if(j==0)
fp=1.00/g*(f[i][j+1][k]-f[i][j][k]);
else
fp=1.00/g*(f[i][j][k]-f[i][j-1][k]);
}
return(fp);
}



double f_zei(f,i,j,k,n,m,l,bc) /* Zei derivative */
double ***f;
int i,j,k,n,m,l;
int bc;
{
double fp,r;
extern int K;
r=(double)(K-1.0)/(l-1.0);
if((k>0)&&(k<l-1))
fp=0.50/r*(f[i][j][k+1]-f[i][j][k-1]);
else
{
if(k==0)
fp=1.00/r*(f[i][j][k+1]-f[i][j][k]);
else
fp=1.00/r*(f[i][j][k]-f[i][j][k-1]);
}
return(fp);
}



double f_psi_psi(u,f,i,j,k,n) /* Psi-Psi double derivative
*/
double ***u,***f;
int i,j,k,n;
{
double fpp,h,kav;
extern int N;
h=(double)(N-1)/(n-1);
if(u==(double ***)NULL)
fpp=1.0/pow(h,2.0)*(f[i+1][j][k]-2.0*f[i][j][k]+f[i-
1][j][k]);
else
{
kav=0.50*(u[i+1][j][k]+u[i-1][j][k]);
fpp=kav/pow(h,2.0)*(f[i+1][j][k]-2.0*f[i][j][k]+f[i-
1][j][k]);
}
return(fpp);
}



double f_eta_eta(u,f,i,j,k,m) /* Eta-Eta double
derivative */
double ***u,***f;
int i,j,k,m;
{
double kav;
double fpp,g;
extern int M;
g=(double)(M-1)/(m-1);
if(u==(double ***)NULL)
fpp=1.0/pow(g,2.0)*(f[i][j+1][k]-2.0*f[i][j][k]+f[i][j-
1][k]);
else
{
kav=0.50*(u[i][j+1][k]+u[i][j-1][k]);
fpp=kav/pow(g,2.0)*(f[i][j+1][k]-2.0*f[i][j][k]+f[i][j-
1][k]);
}
return(fpp);
}



double f_zei_zei(u,f,i,j,k,l) /* Zei-Zei double derivative
*/
double ***u,***f;
int i,j,k,l;
{
extern int K;
```

```
double fpp,kav,r;
r=(double)(K-1)/(l-1);
if(u==(double ***)NULL)
fpp=1.0/pow(r,2.0)*(f[i][j][k+1]-2.0*f[i][j][k]+f[i][j][k-
1]);
else
{
kav=0.50*(u[i][j][k+1]+u[i][j][k-1]);
fpp=kav/pow(r,2.0)*(f[i][j][k+1]-2.0*f[i][j][k]+f[i][j][k-
1]);
}
return(fpp);
}



double f_psi_eta(u,f,i,j,k,n,m) /* Psi-Eta double
derivative */
double ***u;
double ***f;
int i,j,k,n,m;
{
extern int N,M;
double kav,fpp,h,g;
h=(double)(N-1)/(n-1);
g=(double)(M-1)/(m-1);
if(u==(double ***)NULL)
fpp=0.25/(h*g)*(f[i+1][j+1][k]-f[i+1][j-1][k]-f[i-
1][j+1][k]+f[i-1][j-1][k]);
else
{
kav=0.25*(u[i+1][j+1][k]+u[i+1][j-1][k]+u[i-
1][j+1][k]+u[i-1][j-1][k]);
fpp=0.25*kav/(h*g)*(f[i+1][j+1][k]-f[i+1][j-1][k]-f[i-
1][j+1][k]+f[i-1][j-1][k]);
}
return(fpp);
}



double f_eta_psi(u,f,i,j,k,n,m) /* Eta-Psi double
derivative */
double ***u;
double ***f;
int i,j,k,n,m;
{
extern int N,M;
double fpp,h,g,kav;
h=(N-1)/(n-1);
g=(M-1)/(m-1);
if(u==(double ***)NULL)
fpp=0.25/(h*g)*(f[i+1][j+1][k]-f[i+1][j-1][k]-f[i-
1][j+1][k]+f[i-1][j-1][k]);
else
{
kav=0.25*(u[i+1][j+1][k]+u[i+1][j-1][k]+u[i-
1][j+1][k]+u[i-1][j-1][k]);
fpp=0.25*kav/(h*g)*(f[i+1][j+1][k]-f[i-1][j+1][k]-
f[i+1][j-1][k]+f[i-1][j-1][k]);
}
return(fpp);
}



double f_eta_zei(u,f,i,j,k,m,l) /* Eta-Zei double
derivative */
double ***u;
double ***f;
int i,j,k,m,l;
{
extern int M,K;
double fpp,g,r,kav;
g=(M-1)/(m-1);
r=(K-1)/(l-1);
if(u==(double ***)NULL)
fpp=0.25/(g*r)*(f[i][j+1][k+1]-f[i][j+1][k-1]-f[i][j-
1][k+1]+f[i][j-1][k-1]);
else
{
kav=0.25*(u[i][j+1][k+1]+u[i][j-1][k+1]+u[i][j+1][k-
1]+u[i][j-1][k-1]);
fpp=0.25*kav/(g*r)*(f[i][j+1][k+1]-f[i][j+1][k-1]-f[i][j-
1][k+1]+f[i][j-1][k-1]);
}
return(fpp);
}



double f_zei_eta(u,f,i,j,k,m,l) /* Zei-Eta double
derivative */
double ***u;
double ***f;
int i,j,k,m,l;
{
extern int M,K;
double fpp,g,r,kav;
g=(double)(M-1)/(m-1);
r=(double)(K-1)/(l-1);
if(u==(double ***)NULL)
fpp=0.25/(g*r)*(f[i][j+1][k+1]-f[i][j+1][k-1]-f[i][j-
1][k+1]+f[i][j-1][k-1]);
else
{
kav=0.25*(u[i][j+1][k+1]+u[i][j-1][k+1]+u[i][j+1][k-
1]+u[i][j-1][k-1]);
fpp=0.25*kav/(g*r)*(f[i][j+1][k+1]-f[i][j-1][k+1]-
f[i][j+1][k-1]+f[i][j-1][k-1]);
}
return(fpp);
}



double f_zei_psi(u,f,i,j,k,l,n) /* Zei-Psi double
derivative */
double ***u;
double ***f;
int i,j,k,l,n;
{
extern int N,K;
double fpp,r,h,kav;
r=(double)(K-1)/(l-1);
h=(double)(N-1)/(n-1);
if(u==(double ***)NULL)
fpp=0.25/(r*h)*(f[i+1][j][k+1]-f[i-1][j][k+1]-f[i+1][j][k-
1]+f[i-1][j][k-1]);
else
```

```
{
kav=0.25*(u[i+1][j][k+1]+u[i-1][j][k+1]+u[i+1][j][k-
1]+u[i-1][j][k-1]);
fpp=0.25*kav/(r*h)*(f[i+1][j][k+1]-f[i-1][j][k+1]-
f[i+1][j][k-1]+f[i-1][j][k-1]);
}
return(fpp);
}



double f_psi_zei(u,f,i,j,k,l,n) /* Psi-Zei double
derivative */
double ***u;
double ***f;
int i,j,k,l,n;
{
extern int N,K;
double fpp,r,h,kav;
r=(double)(K-1)/(l-1);
h=(double)(N-1)/(n-1);
if(u==(double ***)NULL)
fpp=0.25/(r*h)*(f[i+1][j][k+1]-f[i-1][j][k+1]-f[i+1][j][k-
1]+f[i-1][j][k-1]);
else
{
kav=0.25*(u[i+1][j][k+1]+u[i-1][j][k+1]+u[i+1][j][k-
1]+u[i-1][j][k-1]);
fpp=0.25*kav/(r*h)*(f[i+1][j][k+1]-f[i+1][j][k-1]-f[i-
1][j][k+1]+f[i-1][j][k-1]);
}
return(fpp);
}



int normal(i,n)
int i,n;
{
int ret;
if(i==n-1)
ret=1;
if(i==0)
ret=-1;
return(ret);
}
```