ANALYTICAL SOLUTION FOR INVERSE HEAT CONDUCTION PROBLEM

A Thesis Presented to
The Faculty of the
Fritz J. and Dolores H. Russ
College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Kumar Anagurthi
March, 1999
Acknowledgment

I wish to express my deep gratitude and sincere appreciation to my advisor, Professor Hajrudin Pasic, for his exceptional guidance, advice, and personal commitment throughout this research.

I would like to thank professor Khairul Alam, Moss Professor, for his support and assistance in this research and in teaching me numerical methods, heat transfer, and for continuously providing me with financial assistance. I am grateful to Associate Professor Gary Graham for helping me to write the program in FORTRAN. I wish to thank Professor Bhavin V. Mehta for his encouragement and support. I also thank UES, Inc. for providing the scholarship support.

I would also like to express my appreciation to my parents and my friends for their patience, encouragement, and support.
# Table of Contents

## Chapter 1
**Introduction**

1.1 Overview.................................................................1
1.2 Scope and Objectives of Research.................................2
1.3 Organization of Thesis..................................................4

## Chapter 2
**Literature Review**

2.1 Exact Solutions for the IHCP.........................................5
2.2 Approximate Solutions for DHCP.....................................7

## Chapter 3
**Inverse Determination of the Thermal Conductivity**

3.1 Introduction............................................................9
3.2 General 1D Heat Conduction..........................................9
3.3 Inverse Determination of Thermal Conductivity....................12
3.4 Mathematical Formulation............................................13
Chapter 4

Experimental Method

4.1 Experimental Procedure ............................................. 30
4.2 Tools .................................................................... 33
4.3 Experimental Results .................................................. 36

Chapter 5

Inverse Determination of Heat Flux

5.1 Definition of Problem .................................................. 39
5.2 Inverse Solution for Heat Flux ....................................... 41
5.3 Checking the Solution .................................................. 55
5.4 Determination of Coefficients of Heat Flux Polynomial ......... 62
5.5 Example: Comparison with Experiment .......................... 67
Chapter 6

Conclusions and Recommendations for Future Work........................................71

Abstract..............................................................................................................73

References..........................................................................................................75

Appendix

Appendix A.........................................................................................................78

Appendix B.........................................................................................................143
List of Tables

3.1 Heat Conduction Problems .................................................................22

5.1. Table of Heat Flux Coefficients............................................................68
List of Figures

Figure 3.1. Region of one-dimensional heat conduction.............................................10

Figure 3.2(a) Comparison of thermal conductivity with \( \Delta x = 0.1 \) and \( t = 0.2 \), for
Example 1.................................................................23

Figure 3.2(b) Effect of temperature measurement error on thermal conductivity with
\( \Delta x = 0.1 \) and \( t = 0.2 \), Example 2, 3 and 5.................................24

Figure 3.3(a) Comparison of thermal conductivity at various time intervals for
Example 3.................................................................26

Figure 3.3(b) Comparison of thermal conductivity at various time intervals for
Example 5.................................................................27

Figure 4.2.4.1 Probe..........................................................34

Figure 4.2.4.2 Locations of thermocouples .............................................35

Figure 4.2.4.3 Experimental setup.........................................................36
Figure 4.3.1 Experimental curve ................................................................. 38

Figure 5.1 One-dimensional bar ................................................................. 39

Figure 5.5 Analytical temperature curve ....................................................... 69

Figure 5.6 Comparison of analytical and experimental results ...................... 70
Chapter 1

Introduction

1.1 Overview

If the heat flux or temperature histories at the surface of a heated solid are known as a function of time, then the temperature distribution inside the solid body can be found. This method is called "direct problem". Conversely, a surface flux history can be estimated from a set of discretely measured temperatures, and this method is called "inverse technique". Our problem is to estimate the surface heat flux of a complex object given the body's temperature history. The problem is attacked here for a one-dimensional heat conduction problem only.

In high-speed flight (as in spacecraft re-entry), shock wave interaction with the spacecraft can produce enormous fluxes which can damage aerospace vehicles. To study these fluxes, the phenomenon is simulated in hypersonic wind tunnels. Ordinarily in these situations, however, one cannot measure the flux. Therefore, one could measure the temperature and do computations to recover the heat flux.

Theoretically the problems of determining the surface temperature and the surface heat flux histories are equivalent in the sense that if one is known the other can be found in a straightforward fashion. The inverse heat conduction problem, however, is much more difficult to solve, both analytically and numerically, than the direct problem. The process of "backing out" the heat flux; i.e., the inverse heat conduction problem, is
characterized by instabilities in the solution. Furthermore, traditional methods are predominately limited to a single-dimensional conduction model.

1.2 Scope and Objectives of Research

This thesis is concerned with the inverse heat conduction problem (IHCP). The IHCP is the estimation of the surface heat flux history given one or more measured temperature histories inside a heat-conducting body. The word “estimation” is used because in measuring the internal temperatures, errors are always present to some extent and they, in return, affect the accuracy of the heat flux calculation.

The inverse heat conduction problem is difficult because it is extremely sensitive to measurement errors. The difficulties are particularly pronounced as one tries to obtain the maximum amount of information from the data. For the one-dimensional IHCP when discrete values of the heat flux curve are estimated, maximizing the amount of information implies small time steps between heat flux values. However, the use of small time-steps frequently introduces instabilities in the solution of the IHCP unless restrictions are employed.

One of the earliest papers concerned with the calculation of heat transfer rates during quenching of bodies of simple finite shapes was published by Stolz [12] in 1960. For semi-infinite geometries, Mirsepassi [13] maintained that he had used the same approach numerically and graphically in 1960.

There are various approaches to the inverse heat conduction problem. The application of Duhamel’s theorem was restricted to linear problems. Numerical
procedures, such as the finite difference and the finite element methods, have also been
employed due to their inherent ability to treat nonlinear problems. Exact solutions were
proposed by Burggraf [14], Imber and Khan [15], Langford [16], and others. The IHCP is
one of many ill-posed problems, and these are extremely sensitive to measurement errors.
In 1963, Tikhonov [17] introduced regularization method to reduce those sensitivities of
ill-posed problems to measurement errors.

The purpose of this thesis is to provide a general description of various aspects of
IHCP, as well as to introduce a different approach to inverse heat conduction and related
problems.

Criteria for evaluation of IHCP methods [18] are as follows:

1. The predicted temperature and heat fluxes should be accurate if the
   measured data are of high accuracy.

2. The method should be insensitive to measurement errors.

3. The method should be stable for small time-steps. This permits the
   extraction of more information regarding the time variation of surface
   conditions than is permitted by large time steps.

4. Temperature measurements from one or more sensors should be permitted.

5. The method should not require continuous first-time derivatives of the
   surface heat flux. Furthermore, step changes or even more abrupt changes
   in the surface heat fluxes should be permitted.

6. Knowledge of the precise starting time of the application of the surface
   heat flux should not be required.
7. The method should not be restricted to a fixed number of observations.
8. Composite solids should be permitted.
9. Temperature-variable properties should be permitted.
10. Contact conductance should not be excluded.
11. The method should be easy to program.
12. The computer cost should be moderate.
13. The user should not have to be highly skilled in mathematics in order to use the method or to adapt it to other geometry.
14. The method should permit extension to more than one heating surface.
15. The method should have a statistical basis and permit various statistical assumptions for the measurement errors.

1.3 Organization of Thesis

The scope of the thesis is limited to the inverse heat conduction problem. Chapter 1 is an introduction to the subject. Chapter 2 presents a literature review regarding direct and inverse heat conduction problems. Chapter 3 is concerned with the inverse determination of the thermal conductivity from measured temperature profiles. Chapter 4 explains the experimental setup, the procedure, and the experimental results conducted at Ohio University. Chapter 5 discusses analytical inverse heat conduction solution and its comparison with the experimental results, and Chapter 6 presents the conclusions and suggestions for future work.
Chapter 2

Literature Review

2.1 Exact Solutions for the IHCP

This chapter is an overview of the literature related to DHCP and IHCP. It mainly explains the difference between exact and approximate solution procedures in heat conduction problems by giving limitations and advantages of each.

Exact solutions for IHCP provide closed-form expressions for the heat flux in terms of temperature measurements. They give considerable insight into the characteristics of the inverse problems and provide standards of comparison for approximate methods. However, only a very few exact solutions of IHCP are available in the literature for which the temperature sensor is at an arbitrary location and for some simple geometries. This is in contrast to the direct problem of heat conduction problems for which a wide range of solutions is available.

Burgraf [14] presented one of the earliest exact solutions for temperature. He approached the problem by assuming that both the temperature \( T(t) \) and heat flux \( q(t) \) were known functions of time at a sensor location. The temperature field was represented in terms of an infinite series of all order derivatives of both \( T(t) \) and \( q(t) \) and was found for some very simple geometries, such as a circular cylinder or a sphere.
However, exact solutions have many practical limitations, such as

1. High-order derivatives of discrete temperature data $T(t)$ and heat flux $q(t)$ must be evaluated numerically.

2. The application to composite bodies is not appropriate for temperature-dependent properties.

3. It may require a numerical procedure to solve the direct problem and determine $q(t)$.

4. This approach is not applicable to the over-specified problem of more than one interior temperature sensor.

5. The method does not lend itself readily to the case of multiple heat flux determination, such as the two-dimensional case.

Although the above exact solution has limited applicability in a practical sense, it is extremely important because of the insights provided.

Kover'yanov [19] developed results for the hollow cylinders and spheres. The heat flux at the exposed surface was determined by differentiating the temperature field. Imber and Khan [15] obtained an exact solution for the temperature field using Laplace transforms when the temperature was known at two distinct interior points. Their temperature solution can be extrapolated in both directions toward the boundaries. The extrapolation distance is limited to the distance between the two temperature sensors. No computational results were presented for the more difficult problem of calculating heat flux at the exposed surface.
2.2 **Approximate Solutions for DHCP**

The direct techniques are the first stage of solution procedures for the IHCP. The second stage in this iterative improvement involves the application of specific algorithms to find the final solution for the IHCP. Partial differential equations describing the transient heat conduction equation can be solved using a variety of methods, including exact and numerical procedures. The exact methods normally include the classical methods of separation of variables and Laplace transforms. The numerical methods are based on either integral formulation of the mathematical model or the differential form of the model. The transient heat conduction equation can be either linear or nonlinear. For the linear case, the partial differential equation formulation can be equivalently presented by an integral equation. The advantage of the integral form is that the space dependence can be represented exactly, leaving only the time-dependence to be approximated.

**Duhamel’s theorem** [11] and **Green’s functions** [11] employ integral equations for solving transient heat problems. These techniques yield convolution-type integrals. The heat conducting body can be of arbitrary shape and of non-homogeneous material, although the heat applied may be two- or three-dimensional. The technique that utilizes Duhamel’s theorem for solving connected basic geometries is called the unsteady surface element method [1-5]. It also involves solving an inverse problem, but there are no measurement errors, and the matching conditions are at surfaces rather than at interior points of the bodies. The solution for the nonlinear transient heat conduction problem requires an approach that discretizes the partial differential equation. Two of the most
widely used methods for solving the nonlinear transient heat conduction equation are the
finite difference and finite element methods.

To conclude, this chapter has given a brief description of some of the most
frequently used approaches used to find both exact and approximate solutions and their
basic properties. The next chapter will explain the general problem of determination of
thermal conductivity of inverse heat conduction for a known temperature profile.
Chapter 3

Inverse Determination of the Thermal Conductivity

3.1 Introduction

This chapter will describe some of the most important features of the problem of determination of thermal conductivity for inverse heat conduction for a given temperature profile. Analysis and the algorithm developed in this chapter are based on results published in Reference [1]. This study should help obtain a clearer idea about the inverse heat conduction problem, its basic features, and the associated difficulties when one attempts to determine heat flux from a known temperature profile.

Most of the studies on inverse determination of the thermal conductivity from measured temperature profiles assume that the thermal conductivity is only a function of the spatial coordinate. However, thermal conductivity is a temperature-dependent quantity in most practical engineering applications. The paper [1] presents a second order accurate finite difference procedure for the determination of the thermal conductivity that can be a constant or a spatially or temperature varying quantity. The approach is limited to a one-dimensional problem only.

3.2 General 1D Heat Conduction

Consider a one-dimensional \((0 \leq x \leq 1)\), time-dependent, non-homogeneous problem with heat generation. Figure 3.1 depicts the one-dimensional region \(R\) under consideration. The temperature distribution of the medium is initially prescribed over \(R_3\).
For times \( t > 0 \), the boundaries at \( x = 0 \) and \( x = 1 \) are subjected to a set of boundary conditions over \( R_1 \) and \( R_2 \) of the region \( R \), where \( R = \{(x,t) : 0 < x < 1, t > 0\} \). Everything outside of the region is assumed to be at zero temperature. In addition, the product of the material density and heat capacity is considered as unity.

Figure 3.1. Region of one-dimensional heat conduction

Nomenclature:

\[
\begin{align*}
C & \quad \text{arbitrary constant} \\
T & \quad \text{temperature [ } ^\circ\text{C} \text{]} \\
T_m & \quad \text{initial temperature distribution} \\
x & \quad \text{spatial coordinate [m]} \\
k & \quad \text{thermal conductivity [W m}^{-1} \cdot ^\circ\text{C}^{-1}]
\end{align*}
\]
Heat generation \([W \, m^{-3}]\)

Heat flux \([W \, m^{-2}]\)

Time \([s]\)

The general 1D heat conduction equation is

\[
\frac{\partial T(x,t)}{\partial t} - \frac{\partial}{\partial x} \left[ k(x,t) \frac{\partial T(x,t)}{\partial x} \right] = g(x,t) \quad \text{in } 0 < x < 1, \ t > 0. \tag{3.1}
\]

The initial temperature is

\[
T(x,0) = f_m(x) \quad \text{for } 0 \leq x \leq 1. \tag{3.2}
\]

The boundary conditions at \(x = 0\) and/or \(x = 1\) for \(t > 0\) may take various forms

1. The temperature \(T(x,t)\) is prescribed along the boundary surface.
2. The heat flux \(\frac{\partial T(x,t)}{\partial x}\) is applied at the boundary surface.
3. The heat dissipation by convection from a surface to a surrounding environment at zero temperature \(k(x,t) \left( \frac{\partial T(x,t)}{\partial x} \right) + T(x,t)\) is specified on the surface(s).
The main purpose of this study is to determine the thermal conductivity, $k(x,t)$ at any point $x$ within the domain $R = \{(x,t): 0 < x < 1, t > 0\}$, assuming that the temperature $T(x,t)$ is known at discrete grid points. This is called the inverse determination of thermal conductivity.

### 3.3 Inverse Determination of Thermal Conductivity

Here, the condition for the uniqueness of thermal conductivity $k$ is discussed first. Throughout this section, the temperature $T(x,t)$ is assumed to be known and smooth over the entire domain. As a result, derivatives of the temperature $\left(\frac{\partial T}{\partial x}, \frac{\partial^2 T}{\partial x^2}, \frac{\partial T}{\partial t}\right)$ can also be calculated based on the available temperature profile.

The general 1D heat conduction equation is

$$\frac{\partial}{\partial x} \left[ k(x,t) \frac{\partial T(x,t)}{\partial x} \right] = \frac{\partial T(x,t)}{\partial t} - g(x,t) \tag{3.3}$$

At a given time $t = \hat{t}$ Equation (3.3) becomes

$$\frac{\partial}{\partial x} \left[ k(x,\hat{t}) \frac{\partial T(x,\hat{t})}{\partial x} \right] = \frac{\partial T(x,\hat{t})}{\partial t} - g(x,\hat{t}) \tag{3.4}$$
For the solution of the homogeneous equation

$$\frac{\partial}{\partial x} \left[ k(x,t) \frac{\partial T(x,t)}{\partial x} \right] = 0$$

(3.5)

to exist, the following condition must be satisfied:

$$\left[ k(x,t) \frac{\partial T(x,t)}{\partial x} \right] = C$$

(3.6)
i.e.,

$$k(x,t) = C \frac{\frac{\partial T(x,t)}{\partial x}}{\partial x}$$

(3.7)

where $C$ is a constant. Since $C$ is arbitrary, this implies an infinite number of solutions for the homogeneous equation. Therefore, the necessary condition for the unique solution $k(x,t)$ of the ordinary differential Equation (3.5) is: There exists $x_0$ in the interval $[0,1]$, such that

$$\frac{\partial T(x_0,t)}{\partial x} = 0$$

(3.8)
3.4 Mathematical Formulation

First, the entire domain \( \{(x,t): x \in [0,1] \text{ and } t \in [0,\infty]\} \) is discretized with the mesh in the spatial \( x \) and in the time-direction \( t \) with grid points \( x_j = j\Delta x \), where

\[ j = 0,1,2..., N \text{ and } N\Delta x = l \text{ and } t_i = i\Delta t \text{ (} i = 0,1,2... \text{).} \]

The present procedure will assume that the temperature \( T(x,t) \) is known at the grid points \((x_j,t_i)\). Suppose \((x_j,t_i)\) is an interior point and the governing equation is fixed at this particular point. Based on (3.3) there follows

\[
\frac{\partial T^i}{\partial t} - \frac{\partial}{\partial x} \left[ k(x,t) \frac{\partial T(x,t)}{\partial x} \right]_j = g(x,t)_j \quad (3.9)
\]

Applying forward differential to Equation (3.9)

\[
\left[ k_{j-1} (T_{j-1}^i - T_{j+1}^i) + 4k_j (T_{j-1}^i - 2T_j^i + T_{j+1}^i) + k_{j+1} (T_{j+1}^i - T_{j-1}^i) \right] = 4\Delta x^2 \left[ \frac{T_{j+1}^i - T_j^i}{\Delta t} \right] - g_j \quad (3.10)
\]

Discretization of the governing equation at the boundary \( R_1 \) leads to

\[
\left( \frac{\partial T}{\partial x} \right)_0 = \frac{T_0^{i+1} - T_0^i}{\Delta t} \quad (3.11)
\]
Hence, the finite difference equation at the boundary surface \( R_1 \) takes the form

\[
\frac{T_0^{i+1} - T_0^i}{\Delta t} - \frac{1}{\Delta x^2} \left[ k_1^i \left( T_1^i - T_0^i \right) + k_0^i \left( T_2^i - 3T_1^i + 2T_0^i \right) \right] = g_0^i
\]  

(3.13)

Rearranged, Equation (3.13) reads

\[
\left[ k_1^i \left( T_1^i - T_0^i \right) + k_0^i \left( T_2^i - 3T_1^i + 2T_0^i \right) \right] = \Delta x^2 \left( \frac{T_0^{i+1} - T_0^i}{\Delta t} - g_0^i \right)
\]  

(3.14)

Similarly, discretization of the governing equation at the boundary \( R_2 \) produces

\[
\left[ k_1^i \left( T_{N-1}^i - T_N^i \right) + k_N^i \left( T_{N-1}^i - 3T_N^i + 2T_N^i \right) \right] = \Delta x^2 \left( \frac{T_{N}^{i+1} - T_N^i}{\Delta t} - g_N^i \right)
\]  

(3.15)

### 3.5 Algorithm

Suppose we are interested in solving the inverse heat conduction problem at \( t = \hat{t} \) by assuming a temperature \( T(x, t) \), which is known only at the grid points. Using Equations (3.10), (3.14) and (3.15), one arrives at the linear algebraic system
\[ Ax = b \]  

where \( A \) is \((N+1)\)-by-\((N+1)\) and is defined as

\[
A = \begin{bmatrix}
a_{0,0} & a_{0,1} & & \\
a_{1,0} & a_{1,1} & a_{1,2} & \\
& & \ddots & \ddots \\
& & & a_{N-2,N-2} & a_{N-2,N-1} & a_{N-2,N} \\
& & & & a_{N-1,N-1} & a_{N-1,N} \\
& & & & & a_{N,N-1} & a_{N,N}
\end{bmatrix}
\]

while \( x, b \) are \( N+1 \) vectors

\[
x = \begin{bmatrix} x_0 \\ \vdots \\ x_N \end{bmatrix}, \quad b = \begin{bmatrix} b_0 \\ \vdots \\ b_N \end{bmatrix}
\]

Depending on initial and boundary conditions, \( A \) and \( b \) have different forms.

**Case (1): Temperature data is available.**

Here the temperature data at all nodal points are available and are not changing with space. Then from (3.10) for \( 0 < M < N \), there follows

\[
T(x_{N-1}, \bar{t}) - T(x_{N+1}, \bar{t}) = 0
\]

\[
T(x_{N+1}, \bar{t}) - 2T(x_N, \bar{t}) + T(x_{N-1}, \bar{t}) \neq 0
\]

(3.17)

(3.18)
The above requirements are equivalent to the necessary condition given by Equation (3.8).

At the boundary surface $R_1$, i.e., for $j = 0$, Equation (3.13) takes the form

\[ a_{0,0} = T(x_2, i) - 3T(x_1, i) + 2T(x_0, i), \]
\[ a_{0,1} = T(x_1, i) - T(x_0, i), \]
\[ b_0 = \Delta x^2 \left[ \frac{T(x_0, i + \Delta t) - T(x_0, i)}{\Delta t} - g(x_0, i) \right] \]

For intermediate points, i.e., $j = 1, 2, ..., N - 1$, Equation (3.10) can be written in the form

\[ a_{j,j-1} = T(x_{j-1}, i) - T(x_{j+1}, i) \]
\[ a_{j,j} = 4 \left[ T(x_{j-1}, i) - 2T(x_j, i) + T(x_{j+1}, i) \right] \]
\[ a_{j,j+1} = T(x_{j+1}, i) - T(x_{j-1}, i) \]
\[ b_j = 4\Delta x^2 \left[ \frac{T(x_j, i + \Delta t) - T(x_j, i)}{\Delta t} - g(x_j, i) \right] \]

At the boundary surface $R_2$, i.e., for $j = N$, Equation (3.15) takes the form

\[ a_{N,N} = T(x_{N-2}, i) - 3T(x_{N-1}, i) + 2T(x_N, i), \]
\[ a_{N,N-1} = T(x_{N-1}, i) - T(x_N, i) \]
\[ b_N = \Delta x^2 \left[ \frac{T(x_N, i + \Delta t) - T(x_N, i)}{\Delta t} - g(x_N, i) \right] \]

Case (2): The temperature data and heat flux at $x = 0$ are available.

Here the temperature and heat flux is given at $x = 0$. Therefore, Equations (3.13), (3.10) and (3.15) change their forms.
At the boundary surface $R_i$, i.e., for $j = 0$, Equation (3.13) takes the form

$$a_{0,0} = 1, \quad (3.29)$$

$$a_{0,1} = 0, \quad (3.30)$$

$$b_0 = \frac{q}{\Delta x} \frac{T(x_1, \bar{t}) - T(x_0, \bar{t})}{\Delta x} \quad (3.31)$$

At the intermediate points, i.e., for $j = 1, 2, \ldots N - 1$

$$a_{j,j-1} = T(x_{j-1}, \bar{t}) - T(x_{j+1}, \bar{t}), \quad (3.32)$$

$$a_{j,j} = 4 \left[ T(x_{j-1}, \bar{t}) - 2T(x_j, \bar{t}) + T(x_{j+1}, \bar{t}) \right], \quad (3.33)$$

$$a_{j,j+1} = T(x_{j+1}, \bar{t}) - T(x_{j-1}, \bar{t}) \quad (3.34)$$

$$b_j = 4 \Delta x^2 \left[ \frac{T(x_j, \bar{t} + \Delta t) - T(x_j, \bar{t})}{\Delta t} - g(x_j, \bar{t}) \right] \quad (3.35)$$

At the boundary surface $R_2$, i.e., for $j = N$, Equation (17) takes the form

$$a_{N,N} = T(x_{N-2}, \bar{t}) - 3T(x_{N-1}, \bar{t}) + 2T(x_N, \bar{t}), \quad (3.36)$$

$$a_{N,N-1} = T(x_{N-1}, \bar{t}) - T(x_N, \bar{t}), \quad (3.37)$$

$$b_N = \Delta x^2 \left[ \frac{T(x_N, \bar{t} + \Delta t) - T(x_N, \bar{t})}{\Delta t} - g(x_N, \bar{t}) \right] \quad (3.38)$$

Case (3): The temperature data and heat flux at $x = 1$ is available.

Here the temperature and heat flux are known at $x = 1$. At the boundary surface $R_I$, i.e., for $j = 0$, Equation (3.15) coefficients take values

$$a_{0,0} = T(x_2, \bar{t}) - 3T(x_1, \bar{t}) + 2T(x_0, \bar{t}), \quad (3.39)$$
\[ a_{0,1} = T(x_1, \bar{t}) - T(x_0, \bar{t}) \quad (3.40) \]

\[ b_0 = \Delta x^2 \left[ \frac{T(x_0, \bar{t} + \Delta t) - T(x_0, \bar{t})}{\Delta t} - g(x_0, \bar{t}) \right] \quad (3.41) \]

At the intermediate points, i.e., for \( j = 1, 2, \ldots, N - 1 \), the coefficients in Equation (3.10) become

\[ a_{j,j-1} = T(x_{j-1}, \bar{t}) - T(x_{j+1}, \bar{t}) \quad (3.42) \]

\[ a_{j,j} = 4 \left[ T(x_{j-1}, \bar{t}) - 2T(x_j, \bar{t}) + T(x_{j+1}, \bar{t}) \right] \quad (3.43) \]

\[ a_{j,j+1} = T(x_{j+1}, \bar{t}) - T(x_{j-1}, \bar{t}) \quad (3.44) \]

\[ b_j = 4\Delta x^2 \left[ \frac{T(x_j, \bar{t} + \Delta t) - T(x_j, \bar{t})}{\Delta t} - g(x_j, \bar{t}) \right] \quad (3.45) \]

At the boundary surface \( R_2 \), i.e., for \( j = N \), the coefficients in Equation (3.15) are

\[ a_{N,N} = 1, \quad (3.46) \]

\[ a_{N,N-1} = 0, \quad (3.47) \]

\[ b_N = \frac{q}{T(x_N, \bar{t}) - T(x_{N-1}, \bar{t})} \quad (3.48) \]

Equation (3.16) is a tridiagonal system of linear algebraic equations. It can be efficiently solved by using Thomas algorithm [22]. The solution \( x \) (thermal conductivity) can be calculated using the program given in Appendix A, which is written in C language for different conditions discussed above.
3.6 Error Analysis

Understanding and controlling the numerical error is essential for a successful solution of the finite difference equation. Here we give the error estimate of the above algorithm. Taylor’s expansion to Equation (3.10) produces

\[ k_{j-1}^{i}(T_{j-1}^{i} - T_{j+1}^{i}) + 4k_{j}^{i}(T_{j+1}^{i} - 2T_{j}^{i} + T_{j-1}^{i}) + k_{j+1}^{i}(T_{j+1}^{i} - T_{j-1}^{i}) - 4\Delta x^{2} \left[ \left( T_{j}^{i+1} - T_{j}^{i} \right)/\Delta t \right] - g_{j}^{i} = O(\Delta x^{3} + \Delta t \Delta x^{2}) \]  

(3.49)

If we let \( \Delta t = \Delta x^{2} \), then the local truncation error (LTE) within the interior points due to the finite differencing is \( O(\Delta x^{3}) \). Similarly, analyzing LTE at left boundary \( R_{1} \) using Equation (3.14) leads to

\[ k_{1}^{i}(T_{1}^{i} - T_{0}^{i}) + k_{0}^{i}(T_{2}^{i} - 3T_{1}^{i} + 2T_{0}^{i}) - \Delta x^{2} \left[ \left( T_{0}^{i+1} - T_{0}^{i} \right)/\Delta t \right] - g_{0}^{i} = O(\Delta x^{3} + \Delta t \Delta x^{2}) \]  

(3.50)

If we let \( \Delta t = \Delta x^{2} \), then the local truncation error (LTE) at the \( R_{1} \) boundary surface due to the finite differencing is \( O(\Delta x^{3}) \). Similarly, at \( R_{2} \) the LTE is \( O(\Delta x^{2}) \).

Consequently, the error of the discretization of the governing equation over the domain \( R \cup R_{1} \cup R_{2} \) is \( \Delta x^{2} \). Therefore, the system of linear Equations (3.16) is different from the original equation by an error \( O(\Delta x^{3}) \), i.e.,

\[ Ax = b + O(\Delta x^{2}) \]  

(3.51)
3.7 Examples

The finite difference procedure described above for the inverse determination of the thermal conductivity in a one-dimensional heat conduction domain was programmed in C and is listed in Appendix 1. To show the applicability of the proposed procedure, five distinct test cases were solved. Test cases include constant, spatially-dependent, or temperature-dependent quantities that are reconstructed from discrete temperature data. The heat conduction problems investigated are tabulated in Table 3.1.

The exact temperature and thermal conductivity used in the test case are pre-selected temperature profiles that satisfy the governing heat conduction equation and the boundary conditions, as well as the initial condition. The simulated temperature data are generated from these pre-selected temperature profiles. The numerical procedure is tested by computing the thermal conductivity from these pre-selected temperatures, and its accuracy is assessed by comparing the calculated results with the pre-selected thermal conductivity profiles.
## Table 3.1. Heat Conduction Problems

<table>
<thead>
<tr>
<th>Example</th>
<th>Boundary conditions ((t &gt; 0)) (x = l)</th>
<th>Initial condition (f(x, t))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(T = 0) (T = 0)</td>
<td>(\sin(\pi x))</td>
</tr>
<tr>
<td>2</td>
<td>(T = 0.36te^{-t}) (T = 0.16te^{-t})</td>
<td>(0)</td>
</tr>
<tr>
<td>3</td>
<td>(\frac{\partial T}{\partial x} = \pi e^{-\pi^2 t}\sin(0.8\pi) + T = e^{-\pi^2 t}[\cos(0.2\pi) - \pi \sin(0.2\pi)])</td>
<td>(\cos \pi (x - 0.8))</td>
</tr>
<tr>
<td>4</td>
<td>(T = 9e^{-t}) (q = -54e^{-t})</td>
<td>((x - 3)^2)</td>
</tr>
<tr>
<td>5</td>
<td>(T = 0) (q = 0.5[1 + e^{-t} \sin(1)]e^{-t}\cos(1))</td>
<td>(T = e^{-t} \sin(1)) (\sin(\pi x))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
<th>Heat generation (g(x, t))</th>
<th>Temperature profile (T(x, t))</th>
<th>Conductivity (k(x, t))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>(e^{-2\pi^2 t}\sin(\pi x))</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>((x-0.6)^2(1-t)e^{-t})</td>
<td>((x-0.6)^2te^{-t})</td>
<td>(1 + 0.25e^{-4(x-0.3)^2})</td>
</tr>
<tr>
<td>3</td>
<td>(-\pi^2 e^{-\pi^2 t}\cos\pi(x-0.8)) (+ \pi^2 e^{-\pi^2 t}\cos\pi(x-0.8)) (- \frac{\pi e^{-\pi^2 t}\sin\pi(x-0.8)}{1-e^{-\pi^2 t}\cos\pi(x-0.8)})^2</td>
<td>(e^{-\pi^2 t}\cos\pi(x-0.8))</td>
<td>(\frac{1}{1-T(x, t)})</td>
</tr>
<tr>
<td>4</td>
<td>(-7(x-3)^2e^{-t})</td>
<td>((x-3)^2e^{-t})</td>
<td>((x-3)^2)</td>
</tr>
<tr>
<td>5</td>
<td>(-0.5e^{-t}\sin(x) - )</td>
<td>(e^{-t}\sin(x))</td>
<td>0.5(1 + (T))</td>
</tr>
</tbody>
</table>
Example 1: Constant thermal conductivity without heat generation

Consider a slab, $0 \leq x \leq 1$, with an initial temperature distribution which varies with the distance. For time $t > 0$, the boundaries at $x = 0$ and $x = l$ are kept at zero temperature.

The thermal conductivity is calculated by dividing the region $0 \leq x \leq 1$ into

![Graph showing thermal conductivity](image)

Figure 3.2(a). Results of thermal conductivity with $\Delta x = 0.1$ and $t = 0.2$ for Example 1 and Example 4.

$N = 10, 20, \text{ and } 40$ intervals having mesh size $\Delta x = 0.1, 0.05, \text{ and } 0.025$, respectively.

The results of the thermal conductivity for mesh size $\Delta x = 0.1$ and $t = 0.2$ are presented in Figure 3.2(a).
It seems that in this case the thermal conductivity is almost constant. These results clearly demonstrate that the numerical error decreased with the mesh size. The results from the present study are in very good agreement with the exact solution [1].

Example 2: Spatially dependent thermal conductivity with heat generation

Consider a plane wall, $0 \leq x \leq 1$, initially at zero temperature. For time $t > 0$, heat is generated in the solid at a variable rate of $g(x,t)$, and the boundaries at $x = 0$ and $x = 1$ are subjected to time-varied temperatures. The thermal conductivity is calculated

![Graph showing thermal conductivity vs. spatial coordinate.](image)

Figure. 3.2(b). Effect of temperature measurement error on thermal conductivity with $\delta x = 0.1$ and $t = 0.2$ for Examples 2, 3 and 5.
by dividing the region $0 \leq x \leq 1$ into $N = 10, 20, \text{ and } 40$ intervals with mesh sizes $
abla x = 0.1, 0.05, \text{ and } 0.025$, respectively. The results of the thermal conductivity for mesh size $
abla x = 0.1$ and $t = 0.2$ are presented in Figure 3.2(b). Note that the accuracy of the prediction increases with decreasing grid size, as expected. The calculated thermal conductivity is compared with the exact function and results are plotted in Figure 3.2(b). Clearly, these numerical results are in excellent agreement with the technical research paper [1].

Example 3 Temperature-dependent thermal conductivity with heat generation and having derivative boundary condition at one side and convection boundary condition on the other side.

Consider a plane wall, $0 \leq x \leq 1$, initially maintained at a temperature which varies with distance. For time $t > 0$, heat is generated in the solid at a variable rate of $g(x,t)$. The derivative of temperature is prescribed at the boundary $x = 0$, while the boundary $x = 1$ dissipates heat by convection into an environment of zero temperature. Both conditions vary with time along the surfaces. The thermal conductivity is calculated by dividing the region $0 \leq x \leq 1$ into $N = 10, 20, \text{ and } 40$ intervals, with mesh sizes $
abla x = 0.1, 0.05, \text{ and } 0.025$, respectively. The results of the thermal conductivity for mesh size $
abla x = 0.1$ and $t = 0.2$ is presented in Figure 3.3(a). For $N > 40$, the variation in thermal conductivity values are negligible when compared with corresponding thermal conductivity values at $N = 40$. So the Figure 3.3(a) shows upto 40 intervals only.
Figure 3.3(a). Comparison of thermal conductivity at various time intervals for Example 3

Example 4: Spatially dependent thermal conductivity with known heat flux at left boundary

Next, consider a plane wall, $0 \leq x \leq 1$, initially maintained at a temperature which varies with distance. For time $t > 0$ the boundaries at $x = 0$ and $x = 1$ are kept at a prescribed temperature, which varies with time. In addition to the temperature measurements, the heat flux is also known at the left boundary, $x = 0$. The thermal
conductivity is calculated by dividing the region $0 \leq x \leq 1$ into $N = 10, 20, \text{ and } 40$
intervals having the mesh sizes $\Delta x = 0.1, 0.05, \text{ and } 0.025$, respectively.

Figure 3.3(b). Comparison of thermal conductivity at various time intervals
for Example 5

The results of the thermal conductivity for mesh size $\Delta x = 0.1$ and $t = 0.2$ are
presented in Figure 3.3(b). Again, as the mesh size decreases, the accuracy of the
approximation increases, and the calculated thermal conductivities from the present study
are in good agreement with the analytical solutions.
Example 5: Temperature-dependent thermal conductivity with known heat flux at right boundary

Finally, consider a plane wall, $0 \leq x \leq 1$, initially maintained at a temperature which varies with distance. For time $t > 0$, the boundary at $x = 0$ is kept at zero temperature, while the boundary surface at $x = 1$ is subjected to a prescribed temperature, which varies with time. In addition to the temperature measurements, the heat flux is known at right boundary, $x = 1$. The thermal conductivity is calculated by dividing the region $0 \leq x \leq 1$ into $N = 10, 20, \text{ and } 40$ intervals having mesh size $\Delta x = 0.1, 0.05, \text{ and } 0.025$, respectively. The results for mesh size $\Delta x = 0.1$ and $t = 0.2$ are presented in Figure 3.3(b).

3.8 Conclusions

Based on the approach used in [1], a second-order finite difference-based algorithm has been developed for the inverse determination of the thermal conductivity of a one-dimensional medium. The unknown thermal conductivity is reconstructed using available temperature data or a combination of temperatures and surface heat flux. The algorithm is useful and attractive for heat transfer inverse analysis due to its simplicity, stability, and high speed. The technique is applicable to linear and nonlinear spatially, as well as temperature-dependent, thermal conductivities.

This algorithm is based on the assumption of known temperature and heat flux profiles. But the problem that needs to be solved is to find the surface heat flux which causes a prescribed temperature profile at some locations, and then, based on that flux,
find the temperature profile over the entire body. This type of determining inverse heat flux is explained in detail in Chapters 4 and 5.

The next chapter will explain the experimental setup and the experimental procedure of quenching used for experimental determination of the temperature as a function of time during quenching.
Chapter 4
Experimental Method

This chapter briefly describes the experimental setup used to determine the heat flux for the inverse heat conduction problem, as well as the results of this experiment performed by D. Jack [25] and M. Javid [26], graduate students in Mechanical Engineering at Ohio University. A basic description of the experimental setup and the main result is presented below. In Chapter 5 (Inverse Determination of Heat Flux) these results are compared to our analytical model.

4.1 Experimental Procedure

A prismatic steel probe 6-by-4-by-1 inch rectangular box, with each plate 0.25 inches thick, was heated to the specified steady-state value, then quenched with water. Finally, running the data acquisition application programs to collect and display sample temperature data and corresponding time were noted during the quenching of the probe. The first data acquisition application was used to monitor the probe’s interior surface temperatures until they converged to their respective steady-state values using TestPoint software package. A second application was used to measure temperature response during the cooling stage of the quenching process. These two programs took analog/digital (A/D) board, converted the measured electric voltages to temperature values, and plotted against time.
Each of the thermocouples were sampled every of 0.01 seconds for 2 minutes. Therefore, 12,000 samples were taken per channel and plotted on a graph with temperature versus time. This experimental curve was then compared with the analytical curve, derived from Chapter 5, for the same (IC and BC) conditions as mentioned above.

This experimental procedure was divided into three phases, as described below:

Phase 1: Heating of the probe up to the given steady-state temperature.

Phase 2: Setting the data acquisition system and removing the probe from the furnace.

Phase 3: Quenching the probe and collecting the data up to a specified period.

4.1.1 Phase 1

In the first step, the probe was thoroughly scrubbed with a wire brush to remove any impurities on the surface and then scrubbed with sandpaper. All connections leading from the probe to the data acquisition system were thoroughly checked for any loose connections.

The muffle furnace was fired up and the various settings for a temperature of 400 degrees centigrade (initial temperature) were set on the temperature controller for the furnace to achieve a steady-state of 400 degrees centigrade. Then the probe was set in the furnace and the lid was closed so that there was minimum transfer of heat from the furnace. The probe was set in the furnace until a steady-state was attained, which took about two hours. After the steady-state was reached, Phase 2 of the experiment was conducted.
4.1.2 **Phase 2**

This was a critical phase in the experimental procedure because of the following difficulties:

1. Determining the time before which the data acquisition should be started.
2. Providing sampling rate for the cooling curve.
3. Determining the point where the actual cooling curve started.

The data acquisition system was started one second before the probe was actually quenched. This resulted in some confusion regarding where the actual cooling curve started. To determine this, one of the thermocouples was placed at the surface of the probe, which would behave as a switch and show exactly when the probe hit the water. Since the set temperature was 400 degrees centigrade, the voltage shown by the k type thermocouple was in the vicinity of 20mV. Following the recommendation of its producer, a gain of 250 was set on the data acquisition system. A sampling rate of 100Hz was set on the data acquisition system since it was desired to get a sample every 0.01 seconds. The data acquisition system was set to take data up to 120 seconds and, hence, the number of samples was set at 12,000. The sensors monitored were Thermocouples # 3, 4, and 5 at a gain of 250 with a sampling rate of 100Hz and a time of 120 seconds. The time-interval for collection of data was 0.01 seconds.

4.1.3 **Phase 3**

In this phase, the prepared steady-state heated probe and the holding apparatus was immersed into a water tank (Figure 4.2.4.1.) for quenching, running the data acquisition application programs to measure and record the samples from analog/digital
(A/D) board, and converting the measured electric voltages to temperature values. Then a graph was plotted with temperature versus time for a sample data.

4.2 Tools

4.2.1 Data Acquisition System (D.A.S)

Board: D.A.S. 1701 ST: 16 single-ended or 8 differential channels, 12 bit resolution, 166.67 Ksamples/sec, Gains of 1, 5, 50 and 250.

Manufacturer: Keithley Metrabyte

Accessories: EXP 1800 (expansion accessory) contains 16 differential input multiplexes, with the differential inputs under the control of and at the speed of the host D.A.S. board. Gains of 1 to 50 of the accessories combines with gains of the D.A.S. board to give a total gain of 1 to 400. It works with the supporting software of the attached D.A.S. board.

4.2.2 Test Point Software

It has a full set of features for controlling programming flow, accessing custom hardware, and extending the basic package with external code, add-ons, and data exchange links with other programs.

The relevant features of the software are: support for analog input and output (A/D, D/A), digital I/O, and custom I/O port devices high-speed background A/D, strip charts, bar indicators, and numeric displays.

4.2.3 Computer

P5-200 Professional 200 MHz Pentium processor, 32MB SDRAM, 3.0 GB hard drive.
4.2.4 Probe (Figure 4.2.4.1.)

A prismatic probe had dimensions 6-by-4-by-1 inches and was made of six 304 stainless steel plates; each plate was 0.25 inches thick. The plates were welded to each other, except the top plate, which was screwed onto the hollow box. A thin copper plate was provided as a gasket between the top and bottom so that the probe was leak-free. The quench probe was instrumented with five chromel-alumel (type k) thermocouples. Four thermocouples used to measure temperatures were silver-soldered to the base of the probe at locations described in Figure 4.2.4.2.
The fifth thermocouple was mounted outside the front plate to determine the
starting point of the cooling curve. Thermocouple #1 was located approximately 0.75
inches above the geometric center. Thermocouple #2 was located at geometric center.
Thermocouple #3 was positioned 90 degrees away from thermocouple #1 and 0.75 inches
to the right of the geometric center. Thermocouple #4 was positioned 0.5 inches from the
bottom right-hand corner of the back plate.

A circular hole 0.5 inches in diameter was made on one of the sides and a
stainless steel 304 pipe of 0.5 inch diameter was welded to it, through which
thermocouples could be brought out of the probe to be connected to the D.A.S. The
furnace used is a muffle furnace with a temperature control. The experimental setup is
shown in Figure 4.2.4.3.
4.3 Experimental Results

The experimental results of temperature data versus time is given in Figure 4.3.1. for outside thermocouple, which is located at the end (i.e. $x = 0$) of the probe. It describes temperature measurements taken at thermocouples as a function of time. Each of the thermocouples was sampled every 0.01 seconds for 2 minutes. Therefore, 12,000 samples were taken per channel in order to get an efficient, smooth curve; every 100 samples averaged was calculated and plotted on a graph with temperature versus time. This experimental curve was then compared with the analytical curve, which is derived from Chapter 5 for the same (IC and BC) conditions as mentioned above for the experiment.
4.3.1 **Discussion of Cooling Curve [25]**

The experimental heat transfer cooling curve behavior was shown in Figure: 4.3.1. The starting point of the cooling curve is determined by the time at which the outside thermocouple enters the quenching medium. During this cooling process of probe, four stages of heat transfer will take place. These stages characterize the four different cooling mechanisms that occur during quenching. The four cooling stages are known as the initial liquid contact, the vapor blanket, the nucleate boiling stage, and the convective heat transfer stage.

The initial liquid contact stage lasts for an extremely brief time and is not recognizable on the cooling curve. For this reason, the stage was not included in the cooling experimental curve. This cooling mechanism is characterized by intense boiling at the metal quenchant interface and rapid cooling of the metal surface exposed to the quenchant.

The vapor blanket stage is initiated when the unbroken layer of water vapor bubbles along the surfaces of the heated metal. The extreme temperatures may be high enough to induce plastic deformation, which results in a steep curve.

The nucleate boiling stage begins once there is an insufficient amount of heat transferred from the metal surface to completely vaporize the quenchant. During this stage, even though the temperature gradients within the part are highest, the lower temperatures at which nucleate boiling occur generally results in lower plastic deformation.
The convective heat transfer stage is initiated when the temperature of the metal surface drops below the boiling point of the quenchant.

Figure 4.3.1 Experimental curve

The next chapter will explain the analytical solution for this inverse heat conduction problem and compare it with the experimental results described above.
Chapter 5

Inverse Determination of Heat Flux

This chapter describes an analytical approach to the inverse problem, which consists of determining the heat flux that causes a given (measured) temperature profile at one of the ends of a uniform bar and the temperature distribution over that bar as a function of time. The heat flux input at one end of the bar that causes such a temperature profile is assumed to be a polynomial function of time. The unknown polynomial coefficients are found by combining the method of separation of variables and the least-squares method, which minimizes the difference between the analytical prediction and the previously established experimentally-found temperature profiles.

5.1 Definition of the Problem

Consider a one-dimensional heat conduction problem through a uniform bar, (Figure 5.1.) At one end, \( x = 0 \), the probe (bar) is insulated, and at \( x = L \) it is subjected to heat flux \( q(L,t) \), which is an unknown function of time which is found, based on a given temperature distribution at \( x = 0 \).

\[
\frac{\partial T(x,t)}{\partial x} = 0 \quad \quad \quad k \frac{\partial T(x,t)}{\partial x} = q(L,t)
\]

\[ x = 0 \quad \quad \quad L \quad \quad \quad x = L \]

Figure 5.1 One-dimensional bar
The initial temperature of the probe $T(x,0)$ is assumed to be a constant throughout the bar. For simplicity, the product of the material density and heat capacity is considered as unity.

The governing heat conduction equation describing the temperature distribution in the bar is

$$\frac{\partial^2 T(x,t)}{\partial x^2} = \frac{1}{\alpha} \frac{\partial T(x,t)}{\partial t}$$  \hspace{1cm} (5.1)

The initial condition is

$$T(x,0) = T_0$$  \hspace{1cm} (5.2)

The boundary conditions correspond to the isolated left-end of the bar and the heat flux input at the other end (Figure 5.1.).

$$\text{At } x = 0; \; k \frac{\partial T(x,t)}{\partial x} = 0$$  \hspace{1cm} (5.3)

$$\text{At } x = L; \; k \frac{\partial T(x,t)}{\partial x} = q(L,t)$$  \hspace{1cm} (5.4)

where

- $k$ - thermal conductivity
- $q$ - heat flux
- $T$ - temperature
- $t$ - time
- $x$ - space coordinate
- $a_m$ - coefficients of the heat flux polynomial
The problem may be stated as follows. Given a certain, measured temperature \( T(0, t) \) profile as a function of time at \( x = 0 \), find the heat flux \( q(L, t) \) at that end which causes such a distribution. Also, find the temperature distribution as a function of space and time, i.e., \( \tilde{T}(x, t) \) over the entire bar at any instant and obeying the heat conduction Equation (5.1) and the given boundary and initial conditions.

### 5.2 Analytical Solution for Heat Flux

Notice that the boundary condition (5.4) is nonhomogeneous. In order to be able to solve the problem, that boundary condition is first made homogeneous by introducing the following transformation:

\[
T(x, t) = \psi(x, t) + \frac{q(L, t)}{kL^2} (x^3 - Lx^2) \tag{5.5}
\]

Here it is assumed that the heat flux \( q(L, t) \) can be expressed as a polynomial function of time, where \( a_m \) are the corresponding polynomial coefficients, which will be determined later. Hence, to emphasize it can be written as

\[
q(L, t) = q(L, t, a_m) = \sum_{m=0}^{p} a_m t^m \tag{5.6}
\]

From now on, instead of \( q(L, t) \) the notation \( q(L, t, a_m) \) will be used as a remainder that \( q \) is a polynomial whose coefficients \( a_m \) \((m = 0, 1, 2, ..., p)\) are yet to be found.
Differentiating (5.5) with respect to $x$, one finds

$$\frac{\partial T(x, t)}{\partial x} = \frac{\partial \psi(x, t)}{\partial x} + \frac{q(L, t, a_m)}{kL^2} (3x^2 - 2Lx)$$

(5.7)

$$\frac{\partial^2 T(x, t)}{\partial x^2} = \frac{\partial^2 \psi(x, t)}{\partial x^2} + \frac{q(L, t, a_m)}{kL^2} (6x - 2L)$$

(5.8)

while differentiation of (5.5) with respect to $t$ produces

$$\frac{\partial T(x, t)}{\partial t} = \frac{\partial \psi(x, t)}{\partial t} + \frac{(x^3 - Lx^2)}{kL^2} \frac{\partial q(L, t, a_m)}{\partial t}$$

(5.9)

Written in terms of $\psi$, the governing Equation (5.1) using (5.8) and (5.9) becomes nonhomogeneous

$$\frac{\partial^2 \psi(x, t)}{\partial x^2} \frac{1}{\alpha} \frac{\partial \psi(x, t)}{\partial t} = -\frac{q(L, t, a_m)}{kL^2} (6x - 2L) + \frac{1}{\alpha} \frac{(x^3 - Lx^2)}{kL^2} \frac{\partial q(L, t, a_m)}{\partial t}$$

(5.10)

The boundary conditions (5.3) and (5.4) become homogeneous.

At $x = 0$; $\frac{\partial T(x, t)}{\partial x} = 0$, such that $\frac{\partial \psi(x, t)}{\partial x} = 0$

(5.11)

At $x = L$; $k \frac{\partial T(x, t)}{\partial x} = q(L, t, a_m)$, such that $\frac{\partial \psi(x, t)}{\partial x} = 0$

(5.12)

The initial condition (5.2) at $t = 0$ changes to

$$\psi(x, 0) = T_0 - \frac{q(L, 0, a_m)}{kL^2} (x^3 - Lx^2) = f(x)$$

(5.13)
Now one needs to solve (5.10) subjected to the boundary conditions (5.11), (5.12), and the initial conditions (5.13). For that, one attempts to use the separation of variables method and tries

\[ \psi(x,t) = X(x)G(t) \]  

Differentiating (5.14) with respect to \( x \) one gets

\[ \frac{\partial \psi(x,t)}{\partial x} = X'(x)G(t) \]  

while differentiation of (5.14) with respect to time leads to

\[ \frac{\partial \psi(x,t)}{\partial t} = X(x)\dot{G}(t) \]  

Hence, Equation (5.10) becomes

\[ X'(x)G(t) = \frac{1}{\alpha} X(x)\ddot{G}(t) + \frac{g(L,t,a_m)}{kL^2}(6x - 2L) \]  

One first attempts to solve the homogeneous part of (5.18); i.e.,

\[ X'(x)G(t) - \frac{1}{\alpha} X(x)\dot{G}(t) = 0 \]  

such that one arrives at the following boundary value problem to be solved

\[ \frac{X'(x)}{X(x)} = \frac{1}{\alpha} \frac{\dot{G}}{G} = -\lambda^2, \]  

where \( \lambda \) is the eigenvalue to be determined. Hence, one has to solve two equations,
\[ X^* + \lambda^2 X = 0, \quad (5.21) \]
\[ G + \alpha \lambda^2 G = 0, \quad (5.22) \]

where a dot sign represents differentiation with respect to time. Solutions of (5.21) and (5.23) have the general form of

\[ X(x) = A \cos(\lambda x) + B \sin(\lambda x) \quad (5.23) \]
\[ G(t) = C \exp(-\alpha \lambda^2 t) \quad (5.24) \]

Application of the boundary conditions (5.11) and (5.12) yields

\[ -A \lambda \sin(\lambda 0) + B \lambda \cos(\lambda 0) = 0, \quad (5.25a) \]
\[ B = 0, \quad (5.25b) \]

such that

\[ X(x) = A \cos(\lambda x) \quad (5.26) \]

Since at \( x = L, \frac{\partial \psi(x,t)}{\partial x} = 0 \), i.e. \( -A \lambda \sin(\lambda x) = 0 \), one finds that the eigenvalues sought are

\[ \lambda = \frac{n \pi}{L}, \quad n = 0,1,2\ldots, \quad (5.27) \]

while the corresponding eigenfunctions are

\[ X_n(x) = A_n \cos \left( \frac{n \pi}{L} x \right) \quad (5.28) \]

From (5.24) one finds that

\[ G_n(t) = C_n \exp \left( -\alpha \left( \frac{n \pi}{L} \right)^2 t \right) \quad (5.29) \]
where $C_n$ are constants.

Hence, the solution (5.25) sought has the form

$$
\psi_n(x, t) = X_n(x) G_n(t)
$$

$$
= A_n \cos\left(\frac{n\pi}{L} x\right) C_n \exp\left(-\alpha\left(\frac{n\pi}{L}\right)^2 t\right)
$$

$$
= F_n \cos\left(\frac{n\pi}{L} x\right) \exp\left(-\alpha\left(\frac{n\pi}{L}\right)^2 t\right),
$$

(5.31)

From the superposition principle, which holds for the linear problems, one finally arrives at

$$
\psi_n(x, t) = \sum_{n=0}^{\infty} F_n \cos\left(\frac{n\pi}{L} x\right) \exp\left(-\alpha\left(\frac{n\pi}{L}\right)^2 t\right)
$$

(5.32)

In order to solve the initial value problem, we write

$$
\psi_n(x, t) = \sum_{n=0}^{\infty} C_n(t) \cos\left(\frac{n\pi}{L} x\right)
$$

(5.33)

where $C_n(t) = F_n \exp\left(-\alpha\left(\frac{n\pi}{L}\right)^2 t\right)$ and $F_n$ are constants. Differentiating with respect to $x$ one finds

$$
\frac{\partial \psi_n(x, t)}{\partial x} = \sum_{n=0}^{\infty} -C_n(t) \sin\left(\frac{n\pi}{L} x\right) \left(\frac{n\pi}{L}\right)
$$

(5.34)

$$
\frac{\partial \psi_n^2(x, t)}{\partial x^2} = \sum_{n=0}^{\infty} C_n(t) \cos\left(\frac{n\pi}{L} x\right) \left(\frac{n\pi}{L}\right)^2
$$

(5.35)
while differentiation of (5.33) with respect to $t$ produces

$$\frac{\partial \psi(x,t)}{\partial t} = \sum_{n=0}^{\infty} \dot{C}_n(t) \cos\left(\frac{n\pi}{L} x\right)$$

(5.36)

Equation (5.10) then becomes

$$\frac{\partial^2 \psi(x,t)}{\partial x^2} - \frac{1}{\alpha} \frac{\partial \psi(x,t)}{\partial t} = -q(L,t,a_m) \left(6x - 2L\right) + \frac{1}{\alpha} \frac{(x^3 - Lx^2)}{kL^2} \frac{\partial q(L,t,a_m)}{\partial t}$$

(5.37)

Equation (5.37) can be rewritten using (5.34), (5.35), and (5.36) as

$$\sum_{n=0}^{\infty} \dot{C}_n(t) \cos\left(\frac{n\pi}{L} x\right)\left(\frac{n\pi}{L}\right)^2 - \frac{1}{\alpha} \sum_{n=0}^{\infty} \dot{C}_n(t) \cos\left(\frac{n\pi}{L} x\right)$$

$$= -q(L,t,a_m) \left(6x - 2L\right) + \frac{1}{\alpha} \frac{(x^3 - Lx^2)}{kL^2} \frac{\partial q(L,t,a_m)}{\partial t}$$

(5.38)

In order to solve the above equation, one needs to determine $C_n(t)$. It can be done by using the orthogonality properties of the eigen functions (5.26). By multiplying (5.38) by those functions and integrating over the domain, one gets the system of equations to find $C_n(t)$ as follows. At $n = 0$ Equation (5.38) becomes

$$-\frac{1}{\alpha} \dot{C}_0(t) = \frac{1}{L} \int_0^L \frac{q(L,t,a_m)}{kL^2} \left(6x - 2L\right) dx + \frac{1}{L} \int_0^L \frac{1}{\alpha} \frac{(x^3 - Lx^2)}{kL^2} \frac{\partial q(L,t,a_m)}{\partial t} dx$$

(5.39)

$$-\frac{1}{\alpha} \dot{C}_0(t) = -\frac{1}{L} \frac{q(L,t,a_m)}{kL^2} \left(3x^2 - 2Lx\right)_0^L + \frac{1}{L} \frac{1}{\alpha} \frac{1}{kL^2} \frac{\partial q(L,t,a_m)}{\partial t}$$

(5.40)
\[ \dot{C}_0(t) = -\frac{q(L,t,a_m)\alpha}{kL} + \frac{L}{12k} \frac{\partial q(L,t,a_m)}{\partial t} \]  

Integrating (5.41) one finds

\[ C_0(t) = \frac{\alpha}{kL} \sum_{m=0}^{\infty} \frac{a_m t^{m+1}}{m+1} + \frac{L}{12k} q(L,t,a_m) + \text{Const.} \]  

At \( t = 0 \) Equation (5.42) becomes

\[ C_0(0) = \frac{La_0}{12k} + \text{Const.} \]  

The unknown constant can be determined from the initial condition as follows. The Equation (5.32) is

\[ \psi(x,t) = \sum_{n=0}^{\infty} C_n(t) \cos\left(\frac{n\pi}{L} x\right) \]  

such that for \( n = 0 \) and \( t = 0 \)

\[ C_0(0) = \frac{1}{L} \int_0^L f(x) dx = \frac{1}{L} \int_0^L \left( T_0 - \frac{a_0}{kL} \left( x^3 - Lx^2 \right) \right) dx \]

\[ = \frac{1}{L} \left( T_0 - \frac{a_0}{kL} \left( \frac{x^4}{4} - L \frac{x^3}{3} \right) \right)_0^L \]

\[ = T_0 + \frac{a_0 L}{12k} \]  

Equating (5.43) and (5.45), one finds that

\[ T_0 + \frac{a_0 L}{12k} = \frac{La_0}{12k} + \text{Const.} \]

Therefore, the constant is

\[ \text{Const.} = T_0 \]  

\[ (5.46) \]
Equation (5.38) is

$$\sum_{n=0}^{\infty} - C_n(t) \cos\left(\frac{n\pi}{L} x\right) \left(\frac{n\pi}{L}\right)^2 - \frac{1}{\alpha} \sum_{n=0}^{\infty} \dot{C}_n(t) \cos\left(\frac{n\pi}{L} x\right) = -\frac{q(L,t,a_m)}{kL^2} (6x - 2L) + \frac{1}{\alpha} \frac{(x^3 - Lx^2)}{kL^2} \frac{\partial q(L,t,a_m)}{\partial t}$$

(5.47)

Multiplying both sides by $\cos\left(\frac{r\pi x}{L}\right)$ and integrating from 0 to $L$

$$\int_0^L \sum_{n=0}^{\infty} - C_n(t) \cos\left(\frac{n\pi}{L} x\right) \cos\left(\frac{r\pi}{L} x\right) \left(\frac{n\pi}{L}\right)^2 \, dx - \frac{1}{\alpha} \int_0^L \sum_{n=0}^{\infty} \dot{C}_n(t) \cos\left(\frac{n\pi}{L} x\right) \cos\left(\frac{r\pi}{L} x\right) \, dx$$

$$= -\frac{1}{\alpha} \int_0^L \left(\frac{x^3 - Lx^2}{kL^2} \frac{\partial q(L,t,a_m)}{\partial t}\right) \cos\left(\frac{r\pi}{L} x\right) \, dx = \int_0^L \frac{q(L,t,a_m)}{kL^2} (6x - 2L) \cos\left(\frac{r\pi}{L} x\right) \, dx$$

(5.48)

At $n = r$, Equation (5.48) becomes

$$- C_r(t) \left(\frac{r\pi}{L}\right)^2 \frac{L}{2} - \frac{1}{\alpha} \dot{C}_r \frac{L}{2} = -\frac{q(L,t,a_m)}{kL^2} \left(\frac{6L^2}{n^2 \pi^2}\right) (\cos(n\pi) - 1)$$

$$+ \frac{1}{\alpha} \frac{\partial q(L,t,a_m)}{\partial t} \frac{1}{kL^2} \left[\frac{L^4}{n^4 \pi^4} \left(\left(n^2 \pi^2 - 6\right) \cos(n\pi) + 6\right)\right]$$

$$= \int_0^L \cos\left(\frac{n\pi x}{L}\right) \cos\left(\frac{r\pi x}{L}\right) \, dx = \frac{L}{2}$$

(5.49)

Equation (5.49) can be rewritten as
\[
\dot{C}_n - \alpha \left( \frac{n\pi}{L} \right)^2 C_n(t) = \\
\frac{12q(L,t,a_m)\alpha}{kL\pi^2} (\cos(n\pi) - 1) - \frac{2L}{kn^4\pi^4} \frac{\partial q(L,t,a_m)}{\partial t} \left( (n^2\pi^2 - 6)\cos(n\pi) + 6 \right)
\]

(5.50)

Solving this differential equation leads to

\[
C_n(t) = \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \left[ \int_0^t \left( \frac{12q(L,t,a_m)\alpha}{kL\pi^2} (\cos(n\pi) - 1) - \frac{2L}{kn^4\pi^4} \frac{\partial q(L,t,a_m)}{\partial t} \left( (n^2\pi^2 - 6)\cos(n\pi) + 6 \right) \right) \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) dt + E_{1n} \right]
\]

(5.51)

The heat flux is assumed to have the form \( q(L,t,a_m) = \sum_{m=0}^n a_m t^m \). Therefore, Equation (5.51) becomes

\[
C_n(t) = \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \left[ \int_0^t \frac{12\alpha \cos(n\pi - 1)}{kL\pi^2} q(L,t,a_m) \exp \left( \alpha \left( \frac{n\pi}{L} \right)^2 t \right) dt \right]
\]

\[
- \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \left( \frac{2L}{kn^4\pi^4} \left( (n^2\pi^2 - 6)\cos(n\pi) + 6 \right) \int_0^t \frac{dq(L,t,a_m)}{dt} \exp \left( \alpha \left( \frac{n\pi}{L} \right)^2 t \right) dt + E_{1n} \right)
\]

(5.52)
\[ C_n(t) = \begin{bmatrix}
\frac{12\alpha(\cos(n\pi - 1))}{kL^2\pi^2} \left\{ q(L, t, a_m) - \sum_{m=1}^p m a_m t^{m-1} \right\} + \ldots \n
\end{bmatrix} \]
\[
-\frac{2L}{kn^4 \pi^4} \left( n^2 \pi^2 - 6 \cos(n\pi) + 6 \right) \left( \sum_{m=1}^{p} \frac{ma_m t^{m-1}}{\alpha\left( \frac{n\pi}{L} \right)^2} - \sum_{m=2}^{p} \frac{m(m-1)a_m t^{m-2}}{\left( \alpha\left( \frac{n\pi}{L} \right)^2 \right)^2} + \ldots \right)
\]

\[
+ (E_{1n} + E_{2n}) \exp\left( -\alpha\left( \frac{n\pi}{L} \right)^2 t \right)
\]

(5.54)

where

\[
E_{2n} = \left[ \frac{12\alpha (\cos(n\pi - 1))}{kLn^2 \pi^2} \left( \frac{a_0}{\alpha\left( \frac{n\pi}{L} \right)^2} - \frac{a_1}{\left( \alpha\left( \frac{n\pi}{L} \right)^2 \right)^2} + \ldots \right) \right]
\]

\[
+ \frac{2L}{kn^4 \pi^4} \left( n^2 \pi^2 - 6 \cos(n\pi) + 6 \right) \left( \frac{a_1}{\left( \alpha\left( \frac{n\pi}{L} \right)^2 \right)^2} - \frac{2a_2}{\left( \alpha\left( \frac{n\pi}{L} \right)^2 \right)^2} + \ldots \right)
\]

Assume

\[
E_n = E_{1n} + E_{2n}
\]
Therefore, Equation (5.44) can be written as

\[
\psi(x,t) = T_0 + \frac{\alpha}{kL} \left[ \sum_{m=0}^{\infty} a_m \frac{t^{m+1}}{m+1} \right] + \frac{Lq(L,t,a_m)}{12k} \]

\[
+ \sum_{n=1}^{\infty} \frac{12\alpha (n\pi - 1)}{kLn^2 \pi^2} \left[ \frac{dq(L,t,a_m)}{dt} \left( \alpha \frac{n\pi}{L} \right)^2 + \frac{d^2 q(L,t,a_m)}{dt^2} \left( \alpha \frac{n\pi}{L} \right)^3 \right] \cos \left( \frac{n\pi x}{L} \right) \]

\[
- \frac{2L}{kn^4 \pi^4} \left( n^2 \pi^2 - 6 \cos(n\pi) + 6 \right) \left[ \frac{dq(L,t,a_m)}{dt} \left( \alpha \frac{n\pi}{L} \right)^2 - \frac{d^2 q(L,t,a_m)}{dt^2} \left( \alpha \frac{n\pi}{L} \right)^3 \right] + \ldots \cos \left( \frac{n\pi x}{L} \right) \]

\[
+ E_n \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \cos \left( \frac{n\pi x}{L} \right) \]

(5.55)

Applying initial condition on Equation (5.55) to determine \( E_n \) will finally yield to the temperature profile as a function of heat flux. At \( t = 0 \)

\[
\psi(x,0) = T_0 + \frac{\alpha}{kL} [0] + \frac{Lq_0}{12k} \]

\[
+ \sum_{n=1}^{\infty} \frac{12\alpha (n\pi - 1)}{kLn^2 \pi^2} \left[ \frac{dq(L,t,a_m)}{dt} \left( \alpha \frac{n\pi}{L} \right)^2 + \frac{d^2 q(L,t,a_m)}{dt^2} \left( \alpha \frac{n\pi}{L} \right)^3 \right] \cos \left( \frac{n\pi x}{L} \right) \]
After using the orthogonality conditions, i.e., after multiply both sides with \(\cos \left( \frac{n\pi x}{L} \right)\) and integrating from 0 to \(L\) and simplifying, one finds \(E_n\)

\[
E_n = -\sum_{n=1}^{\infty} \frac{2q(L,t,a_m)}{kn^4 \pi^4} \left( n^2 \pi^2 - 6 \cos(n\pi) + 6 \right) + \sum_{n=1}^{\infty} \frac{12\alpha(\cos(n\pi - 1))}{kL \pi^2} \frac{q(L,t,a_m)}{\alpha^2 \left( \frac{n\pi}{L} \right)^2} + \ldots
\]

\[
+ \sum_{n=1}^{\infty} \frac{2L}{kn^4 \pi^4} \left( n^2 \pi^2 - 6 \cos(n\pi) + 6 \right) \left\{ \frac{dq(L,t,a_m)}{dt} + \frac{d^2q(L,t,a_m)}{dt^2} + \ldots \right\} \left( \frac{n\pi}{L} \right)^2
\]
Therefore, the analytical temperature profile is

\[ T(x,t) = T_0 + \frac{\alpha}{kL} \sum_{m=0}^{n} a_m \frac{t^{m+1}}{m+1} + \frac{Lq(L,t,a_m)}{12k} \]

\[ + \sum_{n=1}^{\infty} \frac{12\alpha(\cos(n\pi - 1))}{kLn^2 \pi^2} \left[ \frac{q(L,t,a_m)}{\alpha \left( \frac{n\pi}{L} \right)^2} - \frac{dq(L,t,a_m)}{dt} \left( \alpha \left( \frac{n\pi}{L} \right) \right)^2 + \frac{d^2 q(L,t,a_m)}{dt^2} \left( \alpha \left( \frac{n\pi}{L} \right) \right)^2 \right] \cos \left( \frac{n\pi x}{L} \right) \]

\[ - \sum_{n=1}^{\infty} \frac{2L}{kn^4 \pi^4} \left( n^2 \pi^2 - 6 \right) \cos(n\pi) + 6 \left\{ \frac{dq(L,t,a_m)}{dt} - \frac{d^2 q(L,t,a_m)}{dt^2} \left( \alpha \left( \frac{n\pi}{L} \right) \right)^2 \right\} \cos \left( \frac{n\pi x}{L} \right) \]

\[ - \sum_{n=1}^{\infty} \frac{2q(L,t,a_m),m=0}{kn^4 \pi^4} L \left( n^2 \pi^2 - 6 \right) \cos(n\pi) + 6 \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \cos \left( \frac{n\pi x}{L} \right) \]

\[ + \sum_{n=1}^{\infty} \frac{12\alpha(\cos(n\pi - 1))}{kLn^2 \pi^2} \left[ \frac{q(L,t,a_m)}{\alpha \left( \frac{n\pi}{L} \right)^2} - \frac{dq(L,t,a_m)}{dt} \left( \alpha \left( \frac{n\pi}{L} \right) \right)^2 \right] \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \cos \left( \frac{n\pi x}{L} \right) \]

\[ + \sum_{n=1}^{\infty} \frac{2L}{kn^4 \pi^4} \left( n^2 \pi^2 - 6 \right) \cos(n\pi) + 6 \left\{ \frac{dq(L,t,a_m)}{dt} - \frac{d^2 q(L,t,a_m)}{dt^2} \left( \alpha \left( \frac{n\pi}{L} \right) \right)^2 \right\} \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \cos \left( \frac{n\pi x}{L} \right) \]

\[ + \frac{q(L,t,a_m)}{kL^2} \left( x^3 - Lx^2 \right) \]

\( (5.57) \)
5.3 Checking the Solution

The above analytical solution to determine temperature profile can be checked by satisfying governing Equation (5.1) as

\[ \frac{\partial^2 T(x,t)}{\partial x^2} - \frac{1}{\alpha} \frac{\partial T(x,t)}{\partial t} = 0 \]  

(5.58)

and also satisfying boundary and initial conditions.

Differentiating \( T(x,t) \), i.e., Equation (5.58) with respect to \( x \), one finds

\[
\frac{\partial^2 T(x,t)}{\partial x^2} = -\sum_{n=1}^{\infty} \frac{12\alpha(\cos(n\pi-1))}{kL^2\pi^2} \left\{ \frac{dq(L,t,a_m)}{dt} - \frac{d^2 q(L,t,a_m)}{dt^2} \right\} \cos \left( \frac{n\pi}{L} \right) \left( \frac{n\pi}{L} \right)^2 \\
+ \sum_{n=1}^{\infty} \frac{2L}{kn^4\pi^4} \left( n^2\pi^2 - 6 \right) \cos(n\pi) + 6 \left[ \frac{dq(L,t,a_m)}{dt} - \frac{d^2 q(L,t,a_m)}{dt^2} \right] \cos \left( \frac{n\pi}{L} \right) \left( \frac{n\pi}{L} \right)^2 \\
+ \sum_{n=1}^{\infty} \frac{2q(L,t,a_m)}{kn^4\pi^4} \left( n^2\pi^2 - 6 \right) \cos(n\pi) + 6 \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \sin \left( \frac{n\pi}{L} \right) \left( \frac{n\pi}{L} \right) \\
+ \sum_{n=1}^{\infty} \frac{12\alpha(\cos(n\pi-1))}{kL^2\pi^2} \left\{ \frac{q(L,t,a_m)}{\alpha} - \frac{dq(L,t,a_m)}{\alpha t} \right\} \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \sin \left( \frac{n\pi}{L} \right) \left( \frac{n\pi}{L} \right)
\]
\[-\sum_{n=1}^{\infty} \frac{2L}{kn^4\pi^4} \left( (n^2\pi^2 - 6) \cos(n\pi) + 6 \right) \left\{ \frac{dq(L,t,a_m)}{dt} - \frac{d^2 q(L,t,a_m)}{dt^2} \right\} + \exp\left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \sin\left( \frac{n\pi x}{L} \right) \frac{n\pi}{L} \]

\[+ \frac{q(L,t,a_m)}{kL^2} (3x^2 - 2Lx) \]

**Check for boundary conditions**

At \( x = 0 \), \( \frac{\partial T(x,t)}{\partial x} = 0 \)

At \( x = L \), \( \frac{\partial T(x,t)}{\partial x} = \frac{q(L,t,a_m)}{k} \)

**Check for initial condition**

At \( t = 0 \), Equation (5.57) follows

\[ T(x,0) = T_0 + \frac{L a_0}{12k} - \sum_{n=1}^{\infty} \frac{2a_0 L}{kn^4\pi^4} \left( (n^2\pi^2 - 6) \cos(n\pi) + 6 \right) \cos\left( \frac{n\pi x}{L} \right) + \frac{a_0}{kL^2} (x^3 - Lx^2) \]

\[ \therefore T(x,0) = T_0 \]

where \( G(x) = \frac{a_0}{kL^2} (x^3 - Lx^2) \) can be written in Fourier series as

\[ \frac{a_0}{kL^2} (x^3 - Lx^2) = -\frac{a_0 L}{12k} + \sum_{n=1}^{\infty} \frac{2a_0 L}{kn^4\pi^4} \left( (n^2\pi^2 - 6) \cos(n\pi) + 6 \right) \cos\left( \frac{n\pi x}{L} \right) \]

Writing \( G(x) = \frac{a_0}{kL^2} (x^3 - Lx^2) \) in the form of the Fourier series
Let \( G(x) = \frac{a_0}{kL^2} (x^3 - Lx^2) = \sum_{n=1}^{\infty} G_n \cos \left( \frac{n\pi x}{L} \right) \)

Then, using the orthogonality conditions, at \( n = 0 \)

\[
G_0 = \frac{1}{L} \int_0^L \frac{a_0}{kL^2} (x^3 - Lx^2) \, dx = \frac{a_0}{kL^3} \left[ \frac{L^4}{4} - \frac{L^4}{3} \right] = -\frac{a_0 L}{12k}
\]

while \( n = 1, 2, \ldots \)

\[
G_n = \frac{2}{L} \int_0^L \frac{a_0}{kL^2} (x^3 - Lx^2) \, dx = \frac{2a_0}{k} \left[ \frac{L}{n^4 \pi^4} \left( (n^2 \pi^2 - 6) \cos(n\pi) + 6 \right) \right]
\]

Therefore,

\[
\frac{a_0}{kL^2} (x^3 - Lx^2) = -\frac{a_0 L}{12k} + \sum_{n=1}^{\infty} \frac{2a_0}{k n^4 \pi^4} \left( (n^2 \pi^2 - 6) \cos(n\pi) + 6 \right) \cos \left( \frac{n\pi x}{L} \right)
\]

**Check for governing equation**

Differentiating \( \frac{\partial T}{\partial x} \), with respect to \( x \)

\[
\frac{\partial^2 T(x, t)}{\partial x^2} = -\sum_{n=1}^{\infty} 12 \alpha (\cos(n\pi - 1)) \left\{ \frac{q(L, t, a_m)}{\alpha \left( \frac{n\pi}{L} \right)^2} - \frac{dq}{dt} \left( \alpha \left( \frac{n\pi}{L} \right)^2 \right) + \ldots \right\} \cos \left( \frac{n\pi x}{L} \right) \left( \frac{n\pi}{L} \right)^2
\]

\[
+ \sum_{n=1}^{\infty} \frac{2L}{kn^4 \pi^4} \left( (n^2 \pi^2 - 6) \cos(n\pi) + 6 \right) \left\{ \frac{dq(L, t, a_m)}{dt} \left( \alpha \left( \frac{n\pi}{L} \right)^2 \right)^2 - \frac{d^2 q(L, t, a_m)}{dt^2} \left( \alpha \left( \frac{n\pi}{L} \right)^3 \right)^3 + \ldots \right\} \cos \left( \frac{n\pi x}{L} \right) \left( \frac{n\pi}{L} \right)^2
\]
Differentiating $T(x,t)$, i.e., Equation (5.58) with respect to $t$, one finds

$$\frac{\partial T(x,t)}{\partial t} = \frac{\alpha q(L,t,a_m)}{kL} + \frac{L}{12k} \frac{\partial q(L,t,a_m)}{\partial t}$$

$$+ \sum_{n=1}^{\infty} \frac{12\alpha(n\pi - 1)}{kLn^2\pi^2} \left\{ \frac{q(L,t,a_m)}{\alpha\left(\frac{n\pi}{L}\right)^2} - \frac{d^2 q(L,t,a_m)}{dt^2} \left(\alpha\left(\frac{n\pi}{L}\right)^2\right) \right\} \cos\left(\frac{n\pi x}{L}\right) + ...$$

$$- \sum_{n=1}^{\infty} \frac{2L \left(6n^2\pi^2 - 6\cos(n\pi) + 6\right)}{kn^4\pi^4} \left\{ \frac{d^2 q(L,t,a_m)}{dt^2} - \frac{d^3 q(L,t,a_m)}{dt^3} \left(\alpha\left(\frac{n\pi}{L}\right)^2\right) \right\} \cos\left(\frac{n\pi x}{L}\right) + ...$$

(5.60)
\[ + \sum_{n=1}^{\infty} \frac{2q(L, t, a_m)}{kn^2 \pi^2} L \left( (n^2 \pi^2 - 6) \cos(n\pi) + 6 \right) \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 \right) \left( \alpha \left( \frac{n\pi}{L} \right)^2 \right) \cos \left( \frac{n\pi x}{L} \right) \]

\[ + \sum_{n=1}^{\infty} \frac{12\alpha^2 (\cos(n\pi - 1))}{kLn^2 \pi^2} \left[ \frac{dq(L, t, a_m)}{dt} - \alpha \left( \frac{n\pi}{L} \right)^2 \frac{\cos(n\pi)}{L} \right] + \ldots \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 \right) \left( \frac{n\pi}{L} \right) \cos \left( \frac{n\pi}{L} \right) \]

\[ - \sum_{n=1}^{\infty} \frac{2La}{kn^4 \pi^4} (n^2 \pi^2 - 6) \cos(n\pi) + 6 \left[ \frac{dq(L, t, a_m)}{dt} - \alpha \left( \frac{n\pi}{L} \right)^2 \frac{\cos(n\pi)}{L} \right] + \ldots \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 \right) \left( \frac{n\pi}{L} \right) \cos \left( \frac{n\pi}{L} \right) \]

\[ + \left( \frac{\frac{L^3}{kL} - Lx^2}{kL^2} \right) \frac{\partial q(L, t, a_m)}{\partial t} \]

(5.61)

Writing \( \frac{q(L, t, a_m)}{KL^2} (6x - 2L) \) in the form of the Fourier series

let \( L(x) = \frac{\frac{q(L, t, a_m)}{KL^2} (6x - 2L)}{\cos \left( \frac{n\pi}{L} \right)} = \sum_{n=0}^{\infty} L_n \cos \frac{n\pi x}{L} = \frac{q(L, t, a_m)}{KL^2} (6x - 2L) \)

Then, using the orthogonality conditions, at \( n = 0 \)

\[ L_0 = \frac{1}{L} \int_0^L \frac{q(L, t, a_m)}{KL^2} (6x - 2L) \, dx = \frac{q(L, t, a_m)}{KL^3} \left( \frac{6x^2}{2} - 2Lx \right)_0^L \]

\[ = \frac{q(L, t, a_m)}{KL^3} (L^2) = \frac{q(L, t, a_m)}{KL} \]

while at \( n = 1, 2 \ldots \)
\[ L_n = \frac{2}{L} \int_0^L q(L, t, a_m) (6x - 2L) \cos \frac{n\pi x}{L} \, dx \]

\[ = \frac{2q}{KL^3} \left( (6x - 2t) \frac{\sin \frac{n\pi x}{L}}{n\pi} + 6 \frac{\cos \frac{n\pi x}{L}}{\left( \frac{n\pi}{L} \right)} \right)_0^L \]

\[ = \frac{12q(L, t, a_m)}{KL^3} \left[ \frac{L^2}{n^2\pi^2}(\cos n\pi - 1) \right] \]

\[ = \frac{12q(L, t, a_m)}{KLn^2\pi^2} (\cos n\pi - 1) \]

\[ \therefore \frac{q(L, t, a_m)}{KL^2} (6X - 2L) = \frac{q(L, t, a_m)}{KL} + \sum_{n=1}^{\infty} \frac{12q(L, t, a_m)}{KLn^2\pi^2} (\cos n\pi - 1) \cos \left( \frac{n\pi x}{L} \right) \]  

(5.62)

Writing \( S(x) = \frac{1}{\alpha KL^2} \frac{\partial q(L, t, a_m)}{\partial t} (x^3 - Lx^2) \) in Fourier series, as follows.

Next, let

\[ S(x) = \frac{1}{\alpha KL^2} \frac{\partial q(L, t, a_m)}{\partial t} (x^3 - Lx^2) = \sum_{n=0}^{\infty} S_n \cos \frac{n\pi x}{L} = \frac{1}{\alpha KL^2} \frac{\partial q(L, t, a_m)}{\partial t} (x^3 - Lx^2) \]

Then, at \( n = 0 \)

\[ S_0 = \frac{1}{L} \int_0^L \frac{1}{\alpha KL^2} \frac{\partial q(L, t, a_m)}{\partial t} (x^3 - Lx^2) \, dx \]
\[
\frac{\partial q(L, t, a_m)}{\partial t} = -\frac{\partial q(L, t, a_m)}{L} = \frac{\partial q(L, t, a_m)}{12K\alpha}
\]

while \(n = 1, 2, 3, \ldots\)

\[
S_n = \frac{2}{L} \int_0^L \frac{1}{\alpha K L^2} \frac{\partial q(L, t, a_m)}{\partial t} (x^4 - Lx^3) \cos \frac{n\pi x}{L} dx
\]

\[
= \frac{2}{K\alpha L^3} \frac{\partial q(L, t, a_m)}{\partial t} \left[ \frac{L^4}{n^4 \pi^4} \right] (n^2 \pi^2 \cos n\pi - 6 \cos n\pi + 6)
\]

\[
= \frac{2L}{K\alpha n^4 \pi^4} \frac{\partial q(L, t, a_m)}{\partial t} (\cos n\pi (n^2 \pi^2 - 6)\cos n\pi + 6)
\]

\[
= \frac{1}{\alpha K L^2} \frac{\partial q(L, t, a_m)}{\partial t} (x^3 - Lx^2) = \frac{\partial q(L, t, a_m)}{12K\alpha} + \sum_{n=1}^{\infty} \frac{2L}{K\alpha n^4 \pi^4} \frac{\partial q(L, t, a_m)}{\partial t} (\cos (n\pi (n^2 \pi^2 - 6)\cos n\pi + 6)) \cos \left( \frac{n\pi x}{L} \right)
\]

(5.63)

Therefore, the L.H.S of (5.59) is

\[
\frac{\partial^2 T(x, t)}{\partial x^2} - \frac{1}{\alpha} \frac{\partial T(x, t)}{\partial t}
\]

\[
= -\sum_{n=1}^{\infty} \frac{12\alpha (\cos (n\pi) - 1)}{kLn^2 \pi^2} \left\{ \frac{q(L, t, a_m)}{\alpha} \right\} \cos \left( \frac{n\pi x}{L} \right)
\]

\[
+ \sum_{n=1}^{\infty} \frac{2L}{k\alpha n^4 \pi^4} \left( n^2 \pi^2 - 6 \right) \cos (n\pi) + 6 \left( \frac{dq(L, t, a_m)}{\partial t} \right) \cos \left( \frac{n\pi x}{L} \right)
\]

\[
+ \frac{q(L, t, a_m)}{kL^2} (6x - 2L) - \frac{1}{\alpha} \left[ \frac{\alpha q(L, t, a_m)}{kL} + \frac{L}{12k} \frac{\partial q(L, t, a_m)}{\partial t} \right]
\]
= 0 \quad \text{(using Equations (5.62) and (5.63))}

This proves that the analytical solution found is the exact one.

5.4 Determination of Coefficients of Heat Flux Polynomial

In Section 5.3, the temperature profile was found in terms of the heat flux. The heat flux is assumed to be well represented as a polynomial function of time having the unknown coefficients $a_0, a_1, \ldots$, i.e.,

$$q(L, t, a_m) = \sum_{m=0}^{p} a_m t^n$$

These coefficients must now be found in order to find the temperature distribution. That can be accomplished by applying the least-squares method to the temperature profile. According to the least-squares method, at $x = 0$

$$\frac{\partial}{\partial a_i} \sum (T - T_e)^2 = 0$$

where $T$ is the analytically determined temperature profile given by Equation (5.58), while $T_e$ is the experimentally determined temperature. Hence, Equation (5.64) requires that

$$2 \sum_{i=0}^{s} (T - T_e(t_i)) \frac{\partial T}{\partial a_m} = 0$$

where $i = 0, 1, 2, \ldots$ and $s$ is an integer
From Equation (5.58) one finds that

\[
T(0, t) - T_e = T_0 + \frac{\alpha}{kL} \sum_{m=0}^{\infty} a_m \frac{t^{m+1}}{m+1} + \frac{Lq(L, t, a_m)}{12k} + \ldots
\]

\[
+ 12\alpha \frac{\cos(n\pi) - 1}{kL^n \pi^2} \left[ \frac{q(L, t, a_m)}{\alpha \left( \frac{n\pi}{L} \right)^2} - \frac{dq(L, t, a_m)}{dt} \left( \alpha \left( \frac{n\pi}{L} \right)^2 \right) + \frac{d^2 q(L, t, a_m)}{dt^2} \left( \alpha \left( \frac{n\pi}{L} \right)^3 \right) - \ldots \right] \cos \left( \frac{n\pi x}{L} \right)
\]

\[
- \frac{2L}{kn^4 \pi^4} \left[ \left( n^2 \pi^2 - 6 \right) \cos(n\pi) + 6 \right] \left[ \frac{dq(L, t, a_m)}{dt} \left( \alpha \left( \frac{n\pi}{L} \right)^2 \right) - \frac{d^2 q(L, t, a_m)}{dt^2} \left( \alpha \left( \frac{n\pi}{L} \right)^3 \right) + \ldots \right] \cos \left( \frac{n\pi x}{L} \right)
\]

\[
- \frac{2q(L, t, a_m)}{kn^4 \pi^4} \left( n^2 \pi^2 - 6 \cos(n\pi) + 6 \right) \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \cos \left( \frac{n\pi x}{L} \right)
\]

\[
- \frac{12\alpha \cos(n\pi) - 1}{kL^n \pi^2} \left[ \frac{q(L, t, a_m)}{\alpha \left( \frac{n\pi}{L} \right)^2} - \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \cos \left( \frac{n\pi}{L} \right) \right]_{n=0}
\]

\[
+ \frac{2L}{kn^4 \pi^4} \left( n^2 \pi^2 - 6 \cos(n\pi) + 6 \right) \left[ \frac{dq(L, t, a_m)}{dt} - \frac{d^2 q(L, t, a_m)}{dt^2} \left[ \alpha \left( \frac{n\pi}{L} \right)^2 \right] + \ldots \right] \right] \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \cos \left( \frac{n\pi x}{L} \right)
\]

\[
\frac{q(L, t, a_m)}{kL^2} \left( x^3 - Lx^2 \right) - T_e
\]

(5.66)

Therefore, differentiating Equation (5.58) with respect to \( a_0 \) at \( x = 0 \), one finds
\[
\frac{\partial T(0,t)}{\partial a_0} = \frac{L}{12k} + \frac{\alpha}{kL} + \sum_{n=1}^{\infty} \frac{12L(\cos(n\pi - 1))}{kn^4\pi^4} \\
- \sum_{n=1}^{\infty} \frac{2L}{kn^4\pi^4} \left( (n^2\pi^2 - 6) \cos(n\pi) + 6 \right) \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \\
- \sum_{n=1}^{\infty} \frac{12L\alpha(\cos(n\pi - 1))}{kLn^4\pi^4} \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) = r_0(t) [\text{assume}] 
\]

(5.67)

Similarly,

\[
\frac{\partial T(0,t)}{\partial a_1} = \frac{Lt}{12k} + \frac{\alpha^2}{2kL} + \sum_{n=1}^{\infty} \frac{12\alpha(\cos(n\pi - 1))}{kLn^2\pi^2} \left[ \frac{t}{\alpha \left( \frac{n\pi}{L} \right)^2} - \frac{1}{\left( \alpha \left( \frac{n\pi}{L} \right)^2 \right)^2} \right] \\
- \sum_{n=1}^{\infty} \frac{2L}{kn^4\pi^4} \left( (n^2\pi^2 - 6) \cos(n\pi) + 6 \right) \left( \frac{1}{\alpha \left( \frac{n\pi}{L} \right)^2} \right) \\
- \sum_{n=1}^{\infty} \frac{12\alpha(\cos(n\pi - 1))}{kLn^2\pi^2} \left[ -\frac{1}{\left( \alpha \left( \frac{n\pi}{L} \right)^2 \right)^2} \right] \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \\
+ \sum_{n=1}^{\infty} \frac{2L}{kn^4\pi^4} \left( (n^2\pi^2 - 6) \cos(n\pi) + 6 \right) \left( \frac{1}{\alpha \left( \frac{n\pi}{L} \right)^2} \right) \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) = r_1(t) [\text{assume}] 
\]

(5.68)
\[
\frac{\partial T(0,t)}{\partial a_2} = \frac{Lt^2}{12k} + \frac{\alpha t^3}{3kL} + \sum_{n=1}^{\infty} \frac{12\alpha (\cos(n\pi - 1))}{kL n^2 \pi^2} \left\{ \frac{t^2}{\alpha \left( \frac{n\pi}{L} \right)^2} - \frac{2t}{\left( \alpha \left( \frac{n\pi}{L} \right) \right)^2} + \frac{2}{\left( \alpha \left( \frac{n\pi}{L} \right) \right)^3} \right\}
\]

\[
- \sum_{n=1}^{\infty} \frac{2L}{kn^4 \pi^4} \left( (n^2 \pi^2 - 6) \cos(n\pi) + 6 \right) \left\{ \frac{2t}{\left( \alpha \left( \frac{n\pi}{L} \right) \right)^3} - \frac{2}{\left( \alpha \left( \frac{n\pi}{L} \right) \right)^2} \right\}
\]

\[
- \sum_{n=1}^{\infty} \frac{12\alpha (\cos(n\pi - 1))}{kL n^2 \pi^2} \left\{ \frac{2}{\left( \alpha \left( \frac{n\pi}{L} \right) \right)^3} \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) \right\}
\]

\[
+ \sum_{n=1}^{\infty} \frac{2L}{kn^4 \pi^4} \left( (n^2 \pi^2 - 6) \cos(n\pi) + 6 \right) \left\{ \frac{-2}{\left( \alpha \left( \frac{n\pi}{L} \right) \right)^3} \right\} \exp \left( -\alpha \left( \frac{n\pi}{L} \right)^2 t \right) = r_2(t) \text{[assume]}
\]

(5.69)

The procedure can be repeated to find \( \frac{\partial T(L,t)}{\partial a_3}, \frac{\partial T(L,t)}{\partial a_4}, \ldots \) correspondingly assume

\( r_3(t), r_4(t), \ldots \)
Equation (5.65) for \( a_0 \) is written as

\[
\begin{align*}
&\left[a_0 r_0(t_0) + a_1 r_1(t_0) + a_2 r_2(t_0) + \ldots + a_p r_p(t_0) + T_0 - T_e(t_0)\right] f_0(t_0) + \\
&\left[a_0 r_0(t_1) + a_1 r_1(t_1) + a_2 r_2(t_1) + \ldots + a_p r_p(t_1) + T_0 - T_e(t_1)\right] f_0(t_1) + \\
&\left[a_0 r_0(t_2) + a_1 r_1(t_2) + a_2 r_2(t_2) + \ldots + a_p r_p(t_2) + T_0 - T_e(t_2)\right] f_0(t_2) + \\
&+ \left[a_0 r_0(t_3) + a_1 r_1(t_3) + a_2 r_2(t_3) + \ldots + a_p r_p(t_3) + T_0 - T_e(t_3)\right] f_0(t_3) = 0
\end{align*}
\]

\[ (5.70) \]

Similarly for \( a_1, a_2, \ldots, a_p \) follows

\[
\begin{align*}
&a_0 [r_0^2(t_0) + r_1^2(t_0) + r_2^2(t_0) + \ldots + r_3^2(t_0)] \\
&+ a_1 [r_0^2(t_1) + r_1^2(t_1) + r_2^2(t_1) + \ldots + r_3^2(t_1)] \\
&+ \ldots + a_p [r_0^2(t_p) + r_1^2(t_p) + r_2^2(t_p) + \ldots + r_3^2(t_p)] \\
= & \left(T_e(t_0) - T_0 \gamma_0(t_0) + (T_e(t_1) - T_0) \gamma_1(t_1) + \ldots + (T_e(t_s) - T_0) \gamma_s(t_s)\right)
\end{align*}
\]

\[ (5.71) \]

\[ \ldots \]

\[
\begin{align*}
&a_0 [r_0^2(t_0) + r_1^2(t_0) + r_2^2(t_0) + \ldots + r_3^2(t_0)] \\
&+ a_1 [r_0^2(t_1) + r_1^2(t_1) + r_2^2(t_1) + \ldots + r_3^2(t_1)] \\
&+ \ldots + a_p [r_0^2(t_p) + r_1^2(t_p) + r_2^2(t_p) + \ldots + r_3^2(t_p)] \\
= & \left(T_e(t_0) - T_0 \gamma_p(t_0) + (T_e(t_1) - T_0) \gamma_p(t_1) + \ldots + (T_e(t_s) - T_0) \gamma_p(t_s)\right)
\end{align*}
\]

\[ (5.72) \]
Solving the above linear algebraic system in the coefficients by the Gauss elimination method, one can find the polynomial coefficients $a_0, a_1, \ldots, a_m$ that are necessary for describing the heat flux (as a function of time), which produces the given temperature profile at $x = 0$.

The above procedure is programmed using C language and is given in Appendix B.

5.5 Example: Comparison with Experiment

The above solution is next compared with the experimentally obtained results described in Chapter 4. In the experiment, the probe $0 \leq x \leq L$ is initially maintained at a fixed temperature $T_0$. At $x = 0$, the probe is insulated, while the boundary $x = L$ dissipates heat due to quenching in water. The experimental temperature $T_e(t)$ at $x = 0$ can be noted using the experimental procedure explained in Chapter 4. The simplified form of the probe represents a bar having one-dimensional heat conduction, which is similar to a bar having the same boundary and initial conditions explained in this chapter. So, the above solution is suitable to find the analytical temperature profile and heat flux for this experiment. Using the C program (Appendix B) of this analytical solution and the experimental data (from experiment), inverse heat flux can be determined.

As we discussed above, the analytical solution will find the constants $a_0, a_1, \ldots, a_m$. These constants are coefficients of heat flux polynomial (Equation 5.6). This means we
are able to find the heat flux as a function of time at a fixed space (the end of the bar), which is not possible through the experiment. This is the main advantage of the analytical solution over the experiment.

For this example the heat flux was assumed as a six degree polynomial function of time at $x=0$, i.e., Equation (5.6):

$$q(L, t, a_6) = \sum_{m=0}^{6} a_m t^n$$

Using the analytical solution described above, the coefficients $a_0, a_1, a_2, \ldots, a_6$ of heat flux are calculated and shown in Table 5.1.

5.1. Table of Heat Flux Coefficients

<table>
<thead>
<tr>
<th>Coefficients</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>-1126101.012</td>
</tr>
<tr>
<td>$a_1$</td>
<td>132610.2469</td>
</tr>
<tr>
<td>$a_2$</td>
<td>-6381.266959</td>
</tr>
<tr>
<td>$a_3$</td>
<td>155.455066</td>
</tr>
<tr>
<td>$a_4$</td>
<td>-2.020459</td>
</tr>
<tr>
<td>$a_5$</td>
<td>0.013351</td>
</tr>
<tr>
<td>$a_6$</td>
<td>-0.000035</td>
</tr>
</tbody>
</table>
An analytical temperature curve (5.5) is plotted by developing an analytical temperature profile for different time steps at $x = 0$.

![Analytical temperature curve](image)

**Figure 5.5** Analytical curve

The experimental data consists of 10,000 temperature data points given in the time period of interest. In order to get a more efficient result and a smooth curve, the average of every 100 points is calculated; the corresponding results are presented in Figure (5.5) for the probe located 10 degrees inclined in furnace. The analytical curve from Figure (5.5) is next compared with the experimental curve from Figure 4.1 described in Chapter 4. Figure (5.6) shows the comparison. From this curve we can observe that both analytical and experimental curves are matching each other very well.
Figure 5.6  Comparison of analytical and experimental results
Chapter 6
Conclusions and Recommendations for Future Work

This thesis first described differences between direct and inverse heat conduction (IHCP) illustrated by several numerical examples. The criteria for evaluation of IHCP methods were discussed as well. The overview of the relevant literature regarding research of both exact and approximate solutions was also presented.

Next, the problem of determination of the thermal conductivity from measured temperature profiles was discussed and the corresponding mathematical model for the inverse heat conduction problem established. The second order accurate finite-difference-based solution algorithm for problems in which a thermal flux can be either a constant or spatially or temperature varying quantity was then developed and presented for one-dimensional problems.

Then, a detailed description of the quench system developed earlier at Ohio University to collect heat transfer data that will be used in future studies to design quench processes for optimum heat treating of materials was described. This apparatus had been used for establishing a correlation between the temperature profile and the corresponding heat flux input. During a quenching experiment process, the part temperatures and the cooling rates need to be controlled within certain limits to ensure the formation of desirable metallurgical properties. Uniform and predictable heat transfer is critical during a quenching process to minimize the potential for distortion, cracking, and nonuniform property distribution.
The experiments on quenching resulted in the specimen's temperature profile as a function of time. It was then assumed that the heat flux causing such a temperature distribution could be suitably represented as an algebraic polynomial function of time and that the unknown polynomial coefficients could be found by the least squares procedure.

Next, a one-dimensional analytical model was established and solved by the separation-of-variables method for the inverse problem of heat conduction through a bar with one end isolated and the other subjected to the heat flux described, which would cause a given temperature distribution. The analytical temperature curve was finally favorably compared with the experimental results and the agreement was very satisfactory.

Future work could be possibly extended to two- and three-dimensional analytical models and their experimental verifications. However, because of more involved boundary-value problems in those situations, solution procedures would require implementation of the finite element or the finite difference methods. The procedure similar to the one described in this thesis to establish basic properties of those elements (or finite-difference cells) may still be used.
This thesis describes a new technique for solving the inverse heat conduction problem in a one-dimensional bar. It consists of determination of a heat flux, which is a function of time and which causes a given, experimentally determined temperature profile measured at one end of the bar while the other end is kept insulated. Based on the found heat flux, the temperature profile is then found over the entire bar as a function of time.

The algorithm combines the separation of the variables method in conjunction with the least-squares method.

A brief description of the differences between direct and inverse heat conduction problems are described first, and criteria that must be met by potentially useful IHCP methods are explained. This is followed by the corresponding literature survey. A general problem of experimental verification of the theory is discussed next. For illustration, a problem of determination of thermal conductivity by the finite difference method, based on the measured temperature profile, is explained in detail.

Next, an experimental setup developed at Ohio University and used to study a quenching process is described. The time change of temperatures during this process had been recorded earlier and is regarded as a known function.

In the analytical treatment, the heat flux at the end of the bar is assumed as the polynomial function of time with the unknown polynomial coefficients. These coefficients were then calculated by employing the least-squares method to fit the
analytically-found solution for the temperature, based on such a flux, and minimizing the error of such a distribution by comparing it with the experimental data. Finally, this heat flux is used to find the analytical solution of the temperature profile over the bar as a function of time. The analytically found temperature profile at the end of the bar was agreed very well with the experimental curve.
References


9. Tervola, P., “A method to determine the thermal conductivity from measured


11. James, V. Beck, Ben Blackwell, and Charles R. St. Clair, Jr., Inverse Heat


15. Imber, M. and Khan, J., “Prediction of transient temperature distributions with

16. Langford, D., “New analytic solutions of the one-dimensional heat equation for
temperature and heat flow rate both prescribed at the same fixed boundary (with

15. Tikhonov, A. N. and Arsenin, V. Y., Solutions of Ill-Posed Problems, V. H. Winston


APPENDIX A

Programs: Inverse determination of thermal conductivity for a given temperature profile (Chapter 3) using finite difference method.

The finite difference procedure described in Chapter 3 for the inverse determination of the thermal conductivity in a one-dimensional heat conduction domain was programmed in C language below. To show the applicability of the proposed procedure, five distinct test cases (Chapter 3) were solved. Test cases include constant, spatially dependent, or temperature dependent quantities that are reconstructed from discrete temperature data. The heat conduction problems investigated are tabulated in Table 3.1.

/********************
CONSTANT THERMAL CONDUCTIVITY WITHOUT HEAT GENERATION (EXAMPLE: 1)
/********************
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

double deltax, deltat; /* Declaration of input datatypes deltax and deltat */

double  l,add;
int     n, i, j; /* n = No. of intervals, i & j are position variables in t(time) and x-directions */

int     m,k;
double  x[100], t[100], T[100][100], a[100][100], b[100],g[100][100];
void title();          /* Title of the program */
void input();          /* Entering deltax, deltat and n */
void xlocations(); /* Calculating nodal points in x-direction i.e., X[j] */
void tlocations(); /* Calculating nodal points in t(time)-direction i.e., t[j] */
void temperaturevalues(); /* Calculating nodal temperature values */
void heatgenerationvalues(); /* Calculating nodal Heat Generation values */
void AandBmatrices(); /* Calculating "A" and "B" matrices at different x-locations keeping time is constant. Here deltax = 0.1, deltat = 0.1, and time = 0.2 which means i=2*/
void ThomasElimination(); /* Applying Thomas Elimination Algorithm */
void AmatrixAfterElimination(); /* Matrix "A" after Elimination */
void BmatrixAfterElimination(); /* Matrix "B" after Elimination */
void ProcessToGetKmatrix(); /* Back substitution method to get Thermal conductivity values at different x-locations */
void thermalconductivityvalues();

void main(void)
{
    title();
    input();
    xlocations();
    tlocations();
    temperaturevalues();
    heatgenerationvalues();
    AandBmatrices();
    ThomasElimination();
    AmatrixAfterElimination();
    BmatrixAfterElimination();
    ProcessToGetKmatrix();
    thermalconductivityvalues();
}

/* Title of the program */
void title()
{
    printf("CONSTANT THERMAL CONDUCTIVITY WITHOUT HEAT GENERATION (EXAMPLE: 1)\n");
    printf("---------------------------------------------------------- _
---- --)
    printf("\n\n");
}

/* Entering deltax, deltat and n */
void input(void)
{
    printf("Please enter deltaX: "); /* Enter mesh width deltax in spatial direction */
    scanf("%lf", &deltax);
    printf("Please enter deltaT: "); /* Enter mesh width deltat in time direction */
    scanf("%lf", &deltat);
    printf("Please enter no. of grid points : ");
}
scanf("%d", &n);
/* Enter number of grid points. Assume mesh size is same in x and time
directions i.e. equal to n (which means it is square mesh) */
printf("\n\n");
}

/* Calculating and Printing nodal points in x-direction i.e X[j] */
void xlocations(void)
{
    printf("x=\t")
    for (j = 0; j <=n; j++) { /* Calculating position values in x-
direction */
        x[j] = j * deltax;
    }
    for (j = 0; j <=n; j++) { /* Printing position values in x-
direction */
        printf("%4.2lf\t", x[j]);
    }
    printf("\n");
}

/* Calculating and Printing nodal points in t(time)-direction i.e t[j] */
void tlocations(void)
{
    printf("t=\t")
    for (i = 0; i <=n; i++) { /* Calculating position values in time-direction */
        t[i] = i * deltat;
    }
    for (i = 0; i <=n; i++) { /* Printing position values in time-
direction */
        printf("%4.2lf\t", t[i]);
    }
    printf("\n\n");
}

/* Calculating and Printing nodal temperature values */
void temperaturevalues(void)
{
    printf("Temperature matrix T[x][t]:\n");
    printf("\n\n");
    for (j = 0; j <=n; j++) { /* Calculating temperature
values at grid locations */
        for (i = 0; i <=n; i++) {
    }
\[ T[j][i] = (\exp(-2\cdot M\_PI\cdot M\_PI \cdot t[i])) \cdot \sin(M\_PI \cdot x[j]) \] /* Temperature Profile */

printf("%10f", T[j][i]); /* Printing temperature values at all the grid points */
}
printf("\n");
printf("\n\n");

void heatgenerationvalues()
{
    printf("Heat generation matrix g[x][t]:\n");
    printf("\n\n");
     for(j=0;j<=n;j++) /* Calculating heat generated values at grid locations*/
    {
        g[j][i]=0;
        printf("%8.4lf g[j][i]; /* Printing heat generated values at grid locations*/

        }
    printf("\n");
}

void AandBmatrices()
{
    printf("\n");
    i=2; /* Here assume deltat = 0.1 which means calculating matrices at time = 0.2 */
    /* Calculating "A" and "B" matrices */
    a[0][0] = T[2][i] - (3 \cdot T[1][i]) + (2 \cdot T[0][i]);
    a[0][1] = T[1][i] - T[0][i];
    b[0] = (deltax \cdot deltay) * (\frac{(T[0][i] - T[1][i])}{deltat}) - g[0][i];

    for (j = 1; j <= n; j++)
    {
        a[j][j - 1] = T[j - 1][i] - T[j + 1][i];
        a[j][j] = 4 * (T[j - 1][i] - (2 \cdot T[j][i]) + T[j + 1][i]);
        a[j][j + 1] = T[j + 1][i] - T[j - 1][i];
\[b[j] = (4 \times \text{deltax} \times \text{deltax}) \times ((T[j][i + 1] - T[j][i]) / (\text{deltat})) - g[j][i];\]

\[a[n][n - 1] = T[n - 1][i] - T[n][i];\]
\[a[n][n] = T[n - 2][i] - (3 \times T[n - 1][i]) + (2 \times T[n][i]);\]

\[T[n][i];\]
\[b[n] = (\text{deltax} \times \text{deltax}) \times ((T[n][i + 1] - T[n][i]) / (\text{deltat})) - g[n][i];\]

```c
printf("Matrix A:\n\n");
/* Printing "A" matrix */
for (j = 0; j <= n; j++) {
    for (k = 0; k <= n; k++) {
        printf(" %8.4lf", a[j][k]);
    }
    printf("\n");
}
printf("\n\n");
/* Printing "B" matrix */
printf("Matrix B:\n\n");
for (j = 0; j <= n; j++) {
    printf("%f\n", b[j]);
}
```

/* Applying Thomas Elimination Algorithm */

```c
void ThomasElimination()
{
    for (k = 0; k <= n; k++){
        for (i = k + 1; i <= n; i++) {
            a[i][k] = a[i][k] / a[k][k];
            for (j = k + 1; j <= n; j++)
                a[i][j] = a[i][j] - a[k][j] * a[i][k];
            b[i] = b[i] - a[i][k] * b[k];
        }
    }
}
```

/* Matrix "A" after Elimination */

```c
void AmatrixAfterElimination()
{
    printf("\n\nA Matrix after elimination:\n\n");
    for (i = 0; i <= n; i++) {
        for (j = 0; j <= n; j++) {
            if (i <= j)
                printf("%f", a[i][j]);
        }
    }
    printf("\n");
}```
else
    printf("0.000000 ");
}
printf("\n");
}

/* Matrix *B* after Elimination */
void BmatrixAfterElimination()
{
    printf("\n\n B Matrix after elimination: \n\n");
    for (i = 0; i <= n; i++) {
        printf("%f\t\n", b[i]);
    }
}

/* Back substitution method to get Thermal conductivity values at different x-locations */
void ProcessToGetKmatrix()
{
    b[n] = b[n] / a[n][n];
    for (i = n - 1; i >= 0; i--) {
        add = b[i];
        for (k = i + 1; k <= n; k++)
            add = add - b[k] * a[i][k];
        b[i] = add / a[i][i];
    }
}

/* Thermal conductivity is a function of space only. Assume deltax = 0.1 and time = 0.2 */
void thermalconductivityvalues()
{
    printf("\n\nThermal conductivity values: \n\n");
    for (i = 0; i <= n; i++) {
        printf("%f\t\n", b[i]);
    }
}
SPATIALLY-DEPENDENT THERMAL CONDUCTIVITY BECAUSE OF HEAT GENERATION

(EXAMPLE: 2)

*************

#include<stdio.h>
#include<stdlib.h>
#include<math.h>

double deltax, deltat; /* Declaration of input datatypes deltax and deltat */
double l,add;
int n, i, j; /* n = No. of intervals, i & j are position variables in t(time) and x-directions */
int m,k;
double x[100], t[100], T[100][100], a[100][100], b[100],g[100][100];

void title(); /* Title of the program */
void input(); /* Entering deltax, deltat and n */
void xlocations();.../* Calculating nodal points in x-direction i.e. X[j] */
void tlocations(); /* Calculating nodal points in t(time)-direction i.e. t[j] */
void temperaturevalues(); /* Calculating nodal temperature values */
void heatgenerationvalues(); /* Calculating nodal Heat Generation values */
void AandBmatrices(); /* Calculating "A" and " B " matrices as a function of time, Take deltax = 0.1, deltat = 0.1 and time = 0.2, which means i=2*/
void ThomasElimination(); /* Applying Thomas Elimination Algorithm */
void AmatrixAfterElimination(); /* Matrix "A" after Elimination */
void BmatrixAfterElimination(); /* Matrix "B" after Elimination */
void ProcessToGetKmatrix(); /* Back substitution method to get Thermal conductivity values at different x-locations */
void thermalconductivityvalues();

void main(void)
{
title();
input();
xlocations();
tlocations();
temperaturevalues();
heatgenerationvalues();
AandBmatrices();
ThomasElimination();
AmatrixAfterElimination();
BmatrixAfterElimination();
ProcessToGetKmatrix();
thermalconductivityvalues();
/* Title of the program */
void title()
{
printf("n\n\nSPATIALLY-DEPENDENT THERMAL CONDUCTIVITY BECAUSE OF HEAT
GENERATION (EXAMPLE: 2)\n\n");
printf("---------------------------------------------------
----");
printf("n\n\n");
}

/* Entering deltax, deltat and n */
void input(void)
{
    printf("Please enter deltaX: "); /* Enter mesh width deltax in
    spatial direction */
    scanf("%lf", &deltax);
    printf("Please enter deltaT: "); /* Enter mesh width deltat in
time direction */
    scanf("%lf", &deltat);
    printf("Please enter no. of grid points : "); /* Enter number of
grid points. */
    scanf("%d", &n);
    printf("n\n\n");
}

/* Calculating and Printing nodal points in x-direction i.e X[j] */
void xlocations(void)
{
    printf("x=\n");
    for (j = 0; j <= n; j++) { /* Calculating position values in x-
direction */
        x[j] = j * deltax;
    }
    for (j = 0; j <= n; j++) { /* Printing position values in x-
direction */
        printf("%4.2lf\t", x[j]);
    }
    printf("\n");
}

/* Calculating and Printing nodal points in t(time)-direction i.e t[j] */
void tlocations(void)
{
    printf("t=\n");
    for (i = 0; i <= n; i++) { /* Calculating position values in
time-direction */
    
    
}
t[i] = i * deltat;
}
for (i = 0; i <= nj; i++) { /* Printing position values in time-direction */
    printf("%4.2f\t", t[i]);
}
printf("\n\n");

/* Calculating and Printing nodal temperature values */

void temperaturevalues(void)
{
    printf("Temperature matrix T[x][t]:\n");
    printf("\n\n");
    for (j = 0; j <= nj; j++) { /* Calculating temperature values at grid locations*/
        for (i = 0; i <= nj; i++) {
            T[j][i] = pow((x[j]-0.6),2)*t[j]*exp(-t[i]); /* Temperature Profile */
            printf("%10f \n, T[j][i]; /* Printing temperature values at all the grid points */
        }
    printf("\n\n");
}

/* Calculating and Printing nodal heat generation values */

void heatgenerationvalues()
{
    printf("Heat generation matrix g[x][t]:\n");
    printf("\n\n");
    for(j=0;j<=nj;j++) /* Calculating heat generated values at grid locations*/
    {
        for(i=0;i<=nj;i++)
        {
            g[j][i]=pow((x[j]-0.6),2)*(1-t[i])*exp(-t[i])-(2+(0.5-4*(x[j]-0.3))*(x[j]-0.6)*exp(-4*pow((x[j]-0.3),2)))*t[i]*exp(-t[i]);
            printf("%8.4lf",g[j][i]); /* Printing heat generated values at grid locations*/
        }
    printf("\n");
/* Calculating "A" and "B" matrices at deltah = 0.1, deltat = 0.1 and time = 0.2 which means i=2*/

void AandBmatrices()
{
    printf("\n");
    /* time = 0.2 */
    /* Calculating "A" and "B" matrices */
    a[0][0] = T[2][i] - (3 * T[1][i]) + ( 2 * T[0][i]);
    a[0][1] = T[1][i] - T[0][i];
    b[0] = (deltax * deltah) * ( ((T[0][i+1] - T[0][i]) / (deltat)) - g[0][i]);
    for (j = 1; j <= n; j++) {
        a[j][j - 1] = T[j - 1][i] - T[j + 1][i];
        a[j][j] = 4 * (T[j - 1][i] - (2 * T[j][i]) + T[j + 1][i]);
        a[j][j + 1] = T[j + 1][i] - T[j - 1][i];
        b[j] = (4 * deltah * deltax) * ( ((T[j][i + 1] - T[j][i]) / (deltat)) - g[j][i]);
    }
    a[n][n - 1] = T[n - 1][i] - T[n][i];
    a[n][n] = T[n - 2][i] - (3 * T[n - 1][i]) + ( 2 * T[n][i]);
    b[n] = ( deltax * deltax) * ( ((T[n][i + 1] - T[n][i]) / (deltat)) - g[n][i]);
    printf("Matrix A:\n\n");
    /* Printing "A" matrix */
    for (j = 0; j <= n; j++) {
            for (k = 0; k <= n; k++) {
                printf(" %8.4lf", a[j][k]);
            }
            printf("\n");
    }
    printf("\n");
    /* Printing "B" matrix */
    printf("Matrix B:\n\n");
    for(j=0; j<=n; j++)
    {
        printf("%f\n", b[j]);
    }
}
/* Applying Thomas Elimination Algorithm */

void ThomasElimination()
{
    for (k = 0; k <= n;k++){
        for (i = k + 1; i <= n; i++) {
            a[i][k] = a[i][k] / a[k][k];
            for (j = k + 1; j <= n; j++)
                a[i][j] = a[i][j] - a[k][j] * a[i][k];
            b[i] = b[i] - a[i][k] * b[k];
        }
    }
}

/* Matrix "A" after Elimination */

void AmatrixAfterElimination()
{
    printf("n
n A Matrix after elimination:\n\n\n");
    for (i = 0; i <= n; i++) {
        for (j = 0; j <= n; j++) {
            if (i <= j)
                printf("%f ", a[i][j]);
            else
                printf("0.000000 ");
        }
        printf("n");
    }
}

/* Matrix "B" after Elimination */

void BmatrixAfterElimination()
{
    printf("n
n B Matrix after elimination:\n\n\n R");
    for (i = 0; i <= n; i++) {
        printf("%f	
\n", b[i]);
    }
}

/* Back substitution method to get Thermal conductivity values at different x-locations */

void ProcessToGetKmatrix()
{
    b[n] = b[n] / a[n][n];
    for (i = n - 1; i >= 0; i--){

add = b[i];
for (k = i + 1; k <= n; k++)
    add = add - b[k] * a[i][k];
b[i] = add / a[i][i];
}
}

void thermalconductivityvalues()
{
    printf("\n\nThermal conductivity values: \n\n");
    for (i = 0; i <= n; i++) {
        printf("%f	\n", b[i]);
    }
    printf("\n");
}
TEMPERATURE-DEPENDENT THERMAL CONDUCTIVITY HAVING DERIVATIVE BOUNDARY CONDITION

AT x=0 DISSIPATING HEAT BY CONVECTION and AT x=1 HEAT GENERATION
(EXAMPLE: 3)

#include<stdio.h>
#include<stdlib.h>
#include<math.h>

double deltax, deltati; /* Declaration of input datatypes deltax and deltat */
double l,addi;
int n, i, j; /* n = No. of intervals, i & j are position variables in t(time) and x-directions */
int m,k;
double x[100], t[100], T[100][100], a[100][100],
b[100],g[100][100];
void title(); /* Title of the program */
void input(); /* Entering deltax, deltat and n */
void xlocations(); /* Calculating nodal points in x-direction i.e. X[j] */
void tlocations(); /* Calculating nodal points in t(time)-direction i.e. t[j] */
void temperaturevalues(); /* Calculating nodal temperature values */
void heatgenerationvalues(); /* Calculating nodal Heat Generation values */
void AandBmatrices(); /* Calculating "A" and "B" matrices */
void ThomasElimination(); /* Applying Thomas Elimination Algorithm */
void AmatrixAfterElimination(); /* Matrix "A" after Elimination */
void BmatrixAfterElimination(); /* Matrix "B" after Elimination */
void ProcessToGetKmatrix(); /* Back substitution method */
void thermalconductivityvalues(); /* Thermal conductivity values*/

void main(void)
{
    title();
    input();
    xlocations();
    tlocations();
    temperaturevalues();
    heatgenerationvalues();
    AandBmatrices();
    ThomasElimination();
    AmatrixAfterElimination();
BmatrixAfterElimination();
ProcessToGetKmatrix();
thermalconductivityvalues();
}

/* Title of the program */
void title()
{
printf("\n\nTEMPERATURE-DEPENDENT THERMAL CONDUCTIVITY (EXAMPLE: 3)\n\n");
printf("---------------------------------------------------------------
---------------
");
printf("\n\n");
}

/* Entering deltax, deltat and n */
void input(void)
{
    printf("Please enter deltaX:"); /* Enter mesh width deltax in spatial direction */
    scanf("%lf", &deltax);
    printf("Please enter deltaT:"); /* Enter mesh width deltat in time direction */
    scanf("%lf", &deltat);
    printf("Please enter no. of grid points :"); /* Enter number of grid points. */
    scanf("%d", &n);
    printf("\n\n");
}

/* Calculating and Printing nodal points in x-direction i.e X[j] */
void xlocations(void)
{
    printf("x=\t");
    for (j = 0; j <=n; j++) { /* Calculating position values in x-direction */
        x[j] = j * deltax;
    }
    for (j = 0; j <=n; j++) { /* Printing position values in x-direction */
        printf("%4.2lf\t", x[j]);
    }
    printf("\n\n");
}
/* Calculating and Printing nodal points in \( t \)(time)-direction i.e \( t[j] \) */
void tlocations(void)
{
    printf("t=t\n");
    for (i = 0; i <=n; i++) { /* Calculating position values in time-direction */
        t[i] = i * deltat;
    }
    for (i = 0; i <=n; i++) { /* Printing position values in time-direction */
        printf("%4.2lf \n", t[i]);
    }
    printf("\n\n");
}

/* Calculating and Printing nodal temperature values */
void temperaturevalues(void)
{
    printf("Temperature matrix \( T[x][t] \):\n");
    printf("\n\n");
    for (j = 0; j <=n; j++) { /* Calculating temperature values at grid locations*/
        for (i = 0; i <=n; i++) {
            T[j][i] = exp(-M_PI*M_PI*t[i])*cos(M_PI*(x[j]-0.8));
        }
    }
    /* Temperature Profile */
    printf("%10f", T[j][i]); /* Printing temperature values at all the grid points */
    printf("\n");
    printf("\n\n");
}

/* Calculating and Printing nodal Heat Generation values */
void heatgenerationvalues()
{
    printf("Heat generation matrix \( g[x][t] \):\n");
    printf("\n\n");
    for(j=0;j<=n;j++) /* Calculating heat generated values at grid locations*/
    {
        for(i=0;i<=n;i++)
        {

\[ g[j][i] = (-M_PI*M_PI*T[j][i]) + ((M_PI*M_PI*T[j][i])/(1-T[j][i])) - \]
\[ \text{pow}((M_PI*\exp(-M_PI*M_PI*t[i])\sin(M_PI*(x[j]-0.8))/(1-T[j][i])), 2); \]

\[ \text{printf}(%8.4lf, g[j][i]); /* Printing heat generated values at grid}
\[ \text{locations*/}
\]
\[ } \]
\[ } \]

\[ /* Calculating "A" and "B" matrices at different x-locations at}
\[ \text{deltax = 0.1, deltat = 0.1 and calculating matrices at time = 0.2,}
\[ \text{which means i=2*/} \]
\[ \text{void AandBmatrices()}
\[ \{ \]
\[ \text{printf("\n");}
\[ \text{i=1; /* Here deltat deltat = 0.1 which means I am calculating matrices}
\[ \text{at time = 0.2 */}
\[ /* Calculating "A" and "B" matrices */}
\[ a[0][0] = T[2][i] - (3 * T[1][i]) + (2 * T[0][i]);}
\[ a[0][1] = T[1][i] - T[0][i];}
\[ b[0] = (\text{deltax} * \text{deltax}) * (((T[0][i+1] - T[0][i]) / (}
\[ \text{deltat)})-g[0][i]); }
\[ \text{for (j = 1; j <= n; j++) {}
\[ a[j][j - 1] = T[j - 1][i] - T[j + 1][i];}
\[ a[j][j] = 4 * (T[j - 1][i] - (2 * T[j][i]) + T[j +}
\[ 1][i]);}
\[ a[j][j + 1] = T[j + 1][i] - T[j - 1][i];}
\[ b[j] = (4 * \text{deltax} * \text{deltax}) * (((T[j][i+1] -}
\[ T[j][i]) / (\text{deltat}))-g[j][i]);}
\[ }
\[ a[n][n - 1] = T[n - 1][i] - T[n][i];}
\[ a[n][n] = T[n - 2][i] - (3 * T[n - 1][i]) + (2 *}
\[ T[n][i]);}
\[ b[n] = (\text{deltax} * \text{deltax}) * (((T[n][i+1] - T[n][i])}
\[ /(\text{deltat}))-g[n][i]); }
\[ \text{printf("Matrix A:\n\n");}
\[ /* Printing "A" matrix */}
\[ \text{for (j = 0; j <= n; j++) {}
\[ \text{for (k = 0; k <= n; k++) {}
\[ \text{printf(* %8.4lf*, a[j][k]);}
\[ }
\[ \text{printf("\n");}
\[ }
\[ \text{printf("\n\n");}
/ * Printing "B" matrix */
printf("Matrix B:\n\n");
for(j=0;j<=n;j++)
{
printf("%f
",b[j]);
}

/* Applying Thomas Elimination Algorithm */
void ThomasElimination()
{
    for (k = 0; k <=n;k++){
        for (i = k + 1; i <= n; i++) {
            a[i][k] = a[i][k] / a[k][k];
            for (j = k + 1; j <= n; j++)
                a[i][j] = a[i][j] - a[k][j] * a[i][k];
            b[i] = b[i] - a[i][k] * b[k];
        }
    }
}

/* Matrix "A" after Elimination */
void AmatrixAfterElimination()
{
    printf("\n\n A Matrix after elimination:\n\n");
    for (i = 0; i <= n; i++) {
        for (j = 0; j <= n; j++) {
            if (i <= j)
                printf("%f", a[i][j]);
            else
                printf("0.000000 ");
        }
    }
}

/* Matrix "B" after Elimination */
void BmatrixAfterElimination()
{
    printf("\n\n B Matrix after elimination:\n\n");
    for (i = 0; i <= n; i++) {
        printf("%f\t\n", b[i]);
    }
}
/* Back substitution method to get Thermal conductivity values at different x-locations */

void ProcessToGetKmatrix()
{
    b[n] = b[n] / a[n][n];
    for (i = n - 1; i >= 0; i--) {
        add = b[i];
        for (k = i + 1; k <= n; k++)
            add = add - b[k] * a[i][k];
        b[i] = add / a[i][i];
    }
}

/* Thermal conductivity values at deltax = 0.1 and at time = 0.2 */

void thermalcon ductivityvalues()
{
    printf("Thermal conductivity values:
    for (i = 0; i <= n; i++) {
        printf("%f\t\n", b[i]);
    }


SPATIALLY-DEPENDENT THERMAL CONDUCTIVITY FOR KNOWN HEATFLUX AT LEFT BOUNDARY

X=0 (EXAMPLE: 4)

#include<stdio.h>
#include<stdlib.h>
#include<math.h>

double deltax, deltat /* Declaration of input datatypes deltax and deltat */
double q, l, addi /* q is a heatflux*/
int n, i, j; /* n = No. of intervals, i & j are position variables in t(time) and x-directions */
int m, k;
double x[100], t[100], T[100][100], a[100][100], b[100], g[100][100];
void title(); /* Title of the program */
void input(); /* Entering deltax, deltat and n */
void xlocations(); /* Calculating nodal points in x-direction i.e. X[j] */
void tlocations(); /* Calculating nodal points in t(time)-direction i.e. t[j] */
void temperaturevalues(); /* Calculating nodal points in t(time)-direction i.e. t[j] */
void heatgenerationvalues(); /* Calculating nodal Heat Generation values */
void AandBmatrices(); /* Calculating "A" and " B " matrices at deltax = 0.1, deltat = 0.1 time = 0.2 which means i=2*/
void ThomasElimination(); /* Applying Thomas Elimination Algorithm */
void AmatrixAfterElimination(); /* Matrix "A" after Elimination */
void BmatrixAfterElimination(); /* Matrix "B" after Elimination */
void ProcessToGetKmatrix(); /* Back substitution method to get Thermal conductivit values at differant x-locations */
void thermalconductivityvalues();

void main(void)
{
    title();
    input();
    xlocations();
    tlocations();
    temperaturevalues();
    heatgenerationvalues();
    AandBmatrices();
    ThomasElimination();
}
ArnatrixAfterElimination();
BmatrixAfterElimination();
ProcessToGetKmatrix();
thermalconductivityvalues();
}

/* Title of the program */
void title()
{
printf("\n\nSPATIALLY-DEPENDENT THERMAL CONDUCTIVITY FOR KNOWN HEATFLUX
AT LEFT BOUNDARY X=0 (EXAMPLE: 4)\n\n");
printf("---------------------------------------------------------------
---------------------------------------------------------------
\n\n");
printf("\n\n");
}

/* Entering deltax, deltat and n */
void input(void)
{
printf("Please enter deltaX: "); /* Enter mesh width deltax in
spatial direction */
scanf("%lf", &deltax);
printf("Please enter deltaT: "); /* Enter mesh width deltat in
time direction */
scanf("%lf", &deltat);
printf("Please enter no. of grid points : ");
scanf("%d", &n);
printf("\n\n");
}

/* Calculating and Printing nodal points in x-direction i.e X[j] */
void xlocations(void)
{
printf("x=t");
for (j = 0; j <=n; j++) { /* Calculating position values in x-
direction */
x[j] = j * deltax;
}
for (j = 0; j <=n; j++) { /* Printing position values in x-
direction */
printf("%4.2lf\t", x[j]);
}
printf("\n");
/* Calculating and Printing nodal points in t(time)-direction i.e. t[j] */

void tlocations(void)
{
    printf("t=\t\n");
    for (i = 0; i <= n; i++) {
        /* Calculating position values in time-direction */
        t[i] = i * deltat;
    }
    for (i = 0; i <= n; i++) {
        /* Printing position values in time-direction */
        printf("%4.2f\t\n", t[i]);
    }
    printf("\n\n");
}

/* Calculating and Printing nodal temperature values */

void temperaturevalues(void)
{
    printf("Temperature matrix T[x][t]:\n");
    printf("\n")
    for (j = 0; j <= n; j++) {
        /* Calculating temperature values at grid locations*/
        for (i = 0; i <= n; i++) {
            T[j][i] = pow((x[j]-3),2)*exp(-t[i]); /* Temperature Profile */
            printf("%10f", T[j][i]); /* Printing temperature values at all the grid points */
        }
        printf("\n");
    }
    printf("\n\n");
}

/* Calculating and Printing nodal Heat Generation values */

void heatgenerationvalues()
{
    printf("Heat generation matrix g[x][t]:\n");
    printf("\n")
    for(j=0;j<=n;j++)  /* Calculating heat generated values at grid locations*/
    {
        for(i=0;i<=n;i++)
        {
            g[j][i]=-7*pow((x[j]-3),2)*exp(-t[i]);
        }
    }
}
printf("%8.4lf",g[j][i]); /* Printing heat generated values at grid locations*/
}
    printf("\n");
}

void AandBmatrices()
{
    printf("\n");
    l=2; /* at deltat = 0.1, at time = 0.2 */
    /* Calculating "A" and "B" matrices */
    a[0][0] = 1;
    a[0][1] = 0;
    q=(-54)*exp(-t[i]);
    b[0] = (q) / ((T[1][i] - T[0][i]) / (deltax));

    for (j = 1; j <= n; j++) {
        a[j][j - 1] = T[j - 1][i] - T[j + 1][i];
        a[j][j] = 4 * (T[j - 1][i] - (2 * T[j][i]) + T[j + 1][i]);
        a[j][j + 1] = T[j + 1][i] - T[j - 1][i];
        b[j] = (4 * deltax * deltax) * (((T[j][i + 1] - T[j][i]) / (deltat)) - g[j][i]);
    }
    a[n][n - 1] = T[n - 1][i] - T[n][i];
    a[n][n] = T[n - 2][i] - (3 * T[n - 1][i]) + (2 * T[n][i]);
    b[n] = (deltax * deltax) * (((T[n][i + 1] - T[n][i]) / (deltat)) - g[n][i]);

    printf("Matrix A:\n\n");
    /* Printing "A" matrix */
    for (j = 0; j <= n; j++) {
        for (k = 0; k <= n; k++) {
            printf(" %8.4lf", a[j][k]);
        }
    }
    printf("\n");

    printf("Matrix B:\n\n");
    /* Printing "B" matrix */
    printf("Matrix B:\n\n");
    for (j = 0; j <= n; j++) {
        printf("%f\n", b[j]);
    }
}
/* Applying Thomas Elimination Algorithm */

void ThomasElimination()
{
    for (k = 0; k <=n;k++){
        for (i = k + 1; i <= n; i++) {
            a[i][k] = a[i][k] / a[k][k];
            for (j = k + 1; j <= n; j++)
                a[i][j] = a[i][j] - a[k][j] * a[i][k];
            b[i] = b[i] - a[i][k] * b[k];
        }
    }
}

/* Matrix "A" after Elimination */

void AmatrixAfterElimination()
{
    printf("\n\n A Matrix after elimination:\n\n");
    for (i = 0; i <= n; i++) {
        for (j = 0; j <= n; j++) {
            if (i <= j)
                printf("%f ", a[i][j]);
            else
                printf("0.000000 ");
        }
    }
    printf("\n");
}

/* Matrix "B" after Elimination */

void BmatrixAfterElimination()
{
    printf("\n\n B Matrix after elimination:\n\n");
    for (i = 0; i <= n; i++) {
        printf("%f\n", b[i]);
    }
}

/* Back substitution method to get Thermal conductivity values at different x-locations */

void ProcessToGetKmatrix()
{
    b[n] = b[n] / a[n][n];
    for (i = n - 1; i >= 0; i--) {
add = b[i];
for (k = i + 1; k <= n; k++)
    add = add - b[k] * a[i][k];
    b[i] = add / a[i][i];
}

/* Thermal conductivity values at deltax = 0.1 and time = 0.2 */
void thermalconductivityvalues()
{
    printf("\n\nThermal conductivity values:\n\n");
    for (i = 0; i <= n; i++) {
        printf("%f\t\n", b[i]);
    }
}
# include <stdio.h>  
# include <stdlib.h>  
# include <math.h>  

double deltax, deltat; /* Declaration of input datatypes deltax and deltat */  
double q, l, add; /* q is a heatflux */  
int n, i, j; /* n = No. of intervals, i & j are position variables in t(time) and x-directions */  
int m, k;  
double x[100], t[100], T[100][100], a[100][100], b[100], g[100][100];  
void title(); /* Title of the program */  
void input(); /* Entering deltax, deltat and n */  
void xlocations(); /* Calculating nodal points in x-direction i.e. X[j] */  
void tlocations(); /* Calculating nodal points in t(time)-direction i.e. t[j] */  
void temperaturevalues(); /* Calculating nodal temperature values */  
void heatgenerationvalues(); /* Calculating nodal Heat Generation values */  
void AandBmatrices();  
void ThomasElimination(); /* Applying Thomas Elimination Algorithm */  
void AmatrixAfterElimination(); /* Matrix "A" after Elimination */  
void BmatrixAfterElimination(); /* Matrix "B" after Elimination */  
void ProcessToGetKmatrix(); /* Back substitution method to get Thermal conductivity values at different x-locations */  
void thermalconductivityvalues();  

void main(void)  
{  
title();  
input();  
xlocations();  
tlocations();  
temperaturevalues();  
heatgenerationvalues();  
AandBmatrices();  
ThomasElimination();  
AmatrixAfterElimination();  
BmatrixAfterElimination();  
ProcessToGetKmatrix();
thermalconductivityvalues();
}

/* Title of the program */
void title()
{
printf("TEMPERATURE-DEPENDENT THERMAL CONDUCTIVITY FOR KNOWN HEAT
FLUX AT X=1 (EXAMPLE: 5)\n");
printf("-----------------------------------------------------------------
--------------------------------");
printf("\n\n");
}

/* Entering deltax, deltat and n */
void input(void)
{
    printf("Please enter deltaX: "); /* Enter mesh width deltax in
spatial direction */
    scanf("%lf ", &deltax);
    printf("Please enter deltaT: "); /* Enter mesh width deltat in
time direction */
    scanf("%lf ", &deltat);
    scanf("%d ", &n);
    printf("Please enter no. of grid points ");
}

/* Calculating and Printing nodal points in x-direction i.e X[j] */
void xlocations(void)
{
    printf("X=\t");
    for (j = 0; j <=n; j++) { /* Calculating position values in x-
direction */
        x[j] = j * deltax;
    }
    for (j = 0; j <=n; j++) { /* Printing position values in x-
direction */
        printf("%4.2lf\t", x[j]);
    }
    printf("\n");
}

/* Calculating and Printing nodal points in t(time)-direction i.e t[j] */
void tlocations(void)
{
    printf("t=\t");
    for (i = 0; i <=n; i++) { /* Calculating position values in
time-direction */
        printf("%4.2lf\t", t[i]);
    }
    printf("\n");
}
\[ t[i] = i * \text{deltat}; \]

for (i = 0; i <= n; i++) { /* Printing position values in time-direction */
    printf("%4.2lf\t", t[i]);
}
printf("\n\n");

/* Calculating and Printing nodal temperature values */

void temperaturevalues(void)
{
    printf("Temperature matrix T[x][t]:\n");
    printf("\n\n");
    for (j = 0; j <= n; j++) { /* Calculating temperature values at grid locations*/
        for (i = 0; i <= n; i++) { /* Temperature Profile */
            T[j][i] = exp(-t[i]) * sin(x[j]); /* Temperature Profile */
            printf("%10f", T[j][i]); /* Printing temperature values at all the grid points */
        }
    printf("\n\n");
}

/* Calculating and Printing nodal Heat Generation values */

void heatgenerationvalues()
{
    printf("Heat generation matrix g[x][t]:\n");
    printf("\n\n");
    for (j=0; j<=n; j++) /* Calculating heat generated values at grid locations*/
    {
        for (i=0; i<=n; i++)
        {
            g[j][i]=(-0.5*exp(-t[i])*sin(x[j]))-(0.5*(exp(-2*t[i]))*(cos(x[j])*cos(x[j])-sin(x[j])*sin(x[j])));
            printf("%8.4lf",g[j][i]); /* Printing heat generated values at grid locations*/
        }
    printf("\n");
}
/* Calculating "A" and "B" matrices */
void AandBmatrices()
{
    printf("\n");
    i=1; /* at deltat = 0.1 which means time = 0.2 */
    /* Calculating "A" and "B" matrices */
    a[0][0] = T[2][i] - (3 * T[1][i]) + (2 * T[0][i]);
    a[0][1] = T[1][i] - T[0][i];
    b[0] = (deltax * deltax) * (((T[0][i+1] - T[0][i]) / (deltat)) - g[0][i]);
    for (j = 1; j <= n; j++) {
        a[j][j - 1] = T[j - 1][i] - T[j + 1][i];
        a[j][j] = 4 * (T[j - 1][i] - (2 * T[j][i]) + T[j + 1][i]);
        a[j][j + 1] = T[j + 1][i] - T[j - 1][i];
        b[j] = (4 * deltax * deltax) * (((T[j][i+1] - T[j][i]) / (deltat)) - g[j][i]);
    }
    a[n][n - 1] = 0;
    a[n][n] = 1;
    q=0.5*(1+exp(-t[i])*sin(1))*exp(-t[i])*cos(1);
    b[n] = (q / ((T[n][i] - T[n-1][i]) / (deltax)));

    printf("Matrix A:\n\n");
    /* Printing "A" matrix */
    for (j = 0; j <= n; j++) {
        for (k = 0; k <= n; k++) {
            printf(" %8.4lf", a[j][k]);
        }
        printf("\n");
    }
    printf("\n\n");

    /* Printing "B" matrix */
    printf("Matrix B:\n\n");
    for(j=0;j<=n;j++)
    {
        printf("%f",b[j]);
    }
}
/ * Applying Thomas Elimination Algorithm */

void ThomasElimination()
{
    for (k = 0; k <= n; k++) {
        for (i = k + 1; i <= n; i++) {
            a[i][k] = a[i][k] / a[k][k];
            for (j = k + 1; j <= n; j++)
                a[i][j] = a[i][j] - a[k][j] * a[i][k];
            b[i] = b[i] - a[i][k] * b[k];
        }
    }
}

/* Matrix *A* after Elimination */

void AMatrixAfterElimination()
{
    printf("\n\n A Matrix after elimination:\n\n");
    for (i = 0; i <= n; i++) {
        for (j = 0; j <= n; j++) {
            if (i <= j)
                printf("%f ", a[i][j]);
            else
                printf("0.000000 ");
        }
        printf("\n");
    }
}

/* Matrix *B* after Elimination */

void BMatrixAfterElimination()
{
    printf("\n\n B Matrix after elimination:\n\n");
    for (i = 0; i <= n; i++) {
        printf("%f \n ", b[i]);
    }
}

/* Back substitution method to get Thermal conductivity values at differant x-locations */

void ProcessToGetKmatrix()
{
    b[n] = b[n] / a[n][n];
    for (i = n - 1; i >= 0; i--) {
add = b[i];
for (k = i + 1; k <= n; k++)
    add = add - b[k] * a[i][k];
b[i] = add / a[i][i];
}

/* Thermal conductivity values time at deltax = 0.1 and time = 0.2 */
void thermalconductivityvalues()
{
    printf("\n\nThermal conductivity values:\n\n");
    for (i = 0; i <= n; i++) {
        printf("%f\t\n", b[i]);
    }
}
Program: This is the ANALYTICAL SOLUTION based on

/* Header files */
#include<stdio.h>
#include<math.h>
define tol 0.000001 /* Tolerance */

/* Global Variables */
double t,x,tplus,xplus,sum,sum1,series;
int n,tsteps,xsteps,i,j;
double l,qc,alpha,length;
double deltat,deltax;
double T0,k,temperature;

main()
{
    /* No. of Time Steps */
    printf("please enter No. of tsteps: ");
    scanf("%d", &tsteps);

    /* Time Interval */
    printf("please enter deltat: ");
    scanf("%lf", &deltat);

    /* space interval */
    printf("please enter deltax: ");
    scanf("%lf", &deltax);

    /* Length of the beam */
    length=0.01;

    /* Heat Flux */
    qc=-200000.0;

    /* Alpha */
    alpha=0.005522;

    /* Thermal Conductivity */
    k=20.0;

    /* Initial temperature */
T0=400.0;

/* Location */
xplus=deltax/length;

/* Specific time */
tplus=(alpha*deltat)/(length*length);
printf("\n\n\n\n·)i
printf(·			x+=%lf

·,xplus)i
printf("t+		 T+		Temperature\n\n\n";

/* Main Loop */
for(i=lii<=tstep sii++)
{
    sum1=0.0;
    n=1;
    while(n<10000){
        t=i*tplus;
        x=xplus;
        sum=t+(1.0/3.0)-x+((0.5)*x*x);
        l = (double) n;
        series=(-2.0/(M_PI*M_PI))*(1.0/(l*l))*exp(-
(l*l*M_PI*M_PI*t))*cos(l*M_PI*x);
        sum1=sum1+series;
        if(_ABS(series)<=tol) /* check */
            break;
        else
            n++;
    }
    sum=sum+sum1;
    temperature=T0+sum*(qc*length/k);

    /********************************************************************************
    ********************************************************************************
    OUTPUT DATA
    ********************************************************************************
    ********************************************************************************
    /* Printing Result */
    printf("%lf\t %lf\t %lf\n",t,sum,temperature);
    }
}
}
Name of the program: Numerical Solution

Description: This program is based on Crank Nicolson and Gauss Elimination Method.

Here one end of bar is insulated and the other end subjected to heat flux, and the initial condition throughout the bar is 400 degree centigrade. This program finds temperature profile as a function of time.

/*********************************************************
***********************************************************/

/* Header files */
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

/* Global Variables */
double deltax, deltat, cp, row, d, add;
double x[500], t[2500], T[2500][500], q[500][2500],
k[2500][500],
    kl[2500][500], Tl[2500][500];
double a[500][500], b[500], b1[500], b2[500];
int ln, n, m, check, xyz;
int s;

/* Function Prototypes */
void input();  /* Input data */
void xvalues(); /* Calculating space values */
void tvalues(); /* Calculating time values */
void heatflux(); /* Function for Heat flux */
void initialtemperature(); /* Function for Initial Temperature */
void test();
void aandbmatrix(); /* Forming A and B matrix */
void guass(); /* Applying Gauss Elimination */
void loop(); /* Loop process */
void thermal(); /* Finding temperature */
void tt();
static int maxiter = 0;
MAIN

DESCRIPTIN : It calls all the functions the and finds the results. Here time(xyz) is zero at first and then it goes into loop computing temperature at fixed space after one time step. This process continues until number of time steps reaches m.

main()
{
    int iter;
    xyz=0;
    input(); /* Input asking from user */
    xvalues(); /* Calculating space valus */
    tvalues(); /* Calculating time valus */
    heatflux(); /* Calculating Heat Flux */
    initialtemperature(); /* Initial Temperature */
    test(); /* Thermal Conductivity */
    for(xyz=0;xyz<m;xyz++)
    {
        aandbmatrix(xyz);
        guass();
        loop();
        maxiter = 0;
    }
}

INPUT DATA

void input()
{
    printf("Please enter deltax:");
    scanf("%lf", &deltax);

    printf("Please enter deltat:");
    scanf("%lf", &deltat);

    printf("Please enter no. of x steps:");
    scanf("%d", &n);

    printf("Please enter no. of time steps:");
    scanf("%d", &m);
}
SPACE VALUES
DESCRIPTION: The complete beam is divided into
Number of space steps(n). Then it calculates x value at each node.

```c
void xvalues()
{
    int j;
    for (j = 1; j <= n; j++) {
        x[j] = j * deltax;
    }
}
```

TIME VALUES
DESCRIPTION: Each x value subjected to Number of time
steps(m). Then it calculates t value.

```c
void tvalues()
{
    int j;
    for (j = 0; j <= m; j++) {
        t[j] = j * deltat;
    }
}
```

HEAT FLUX
DESCRIPTION: This is a function of time.

```c
void heatflux()
{
    int j;
    for (j = 0; j <= m; j++) {
        q[n][j] = 483.939 - (332.324 * t[j]) -
                  (456.814 * t[j] * t[j]) + (179.939 * t[j] * t[j] * t[j]) -
                  (22.736 * pow(t[j], 4)) + (0.958 * pow(t[j], 5));
    }
}
```
INITIAL TEMPERATURE
DESCRIPTION: This is function of space at time = 0.
*****************************************************************************

```c
void initialtemperature()
{
    int j;
    for (j = 1; j <= n; j++) {
        T[j][0] = 400.0;
    }
}

void test()
{
    int j, i;
    for (j = 1; j <= n; j++) {
        for (i = 0; i <= n; i++)
            b1[i] = T[i][xyz];
    }
```
/**A and B Matrices**

**DESCRIPTION:** Crank Nikholsen method used to find A and B matrices.

```c
void aandbmatrix()
{
    int j, i;
    row = 7817.0; /* Row */
    cp = 0.46; /* cp */
    d = (deltat) / (2.0 * row * cp * deltay * deltay);
    j = 0;
    a[j][j] = 1;
    a[j][j + 2] = -1;
    b[j] = -2.0 * deltay * 0.0;
    for (j = 1; j <= n; j++) {
        a[j][j - 1] = -d * k[j - 1][xyz][j - 1];
        a[j][j] = 1.0 + d * (k[j][j] + k[j][j - 1]);
        a[j][j + 1] = -d * k[j][j];
        b[j] = d * k[j][j - 1] * T[j - 1][xyz] + (1 - d * (k[j][j] + k[j][j - 1])).
    }
    j = n + 1;
    a[j][j] = 1.0;
    a[j][j - 2] = -1.0;
    a[j][j - 1] = 0.0;
    b[j] = 2.0 * deltay * (q[j - 1][xyz] / k[xyz][j - 1]);
}
```

/**Applying Gauss Elimination**

**DESCRIPTION:** Applying Gauss Elimination method to above formed A and B matrices.

```c
void guass()
{
    int i, j, k, p, max;
    double large, temp, ramp;
    for (i = 0; i <= n + 1; i++) {
        max = i;
        large = a[i][i];
        for (p = 0; p <= n + 1; p++) {
            if (a[p + 1][i] > large) {
                max = p + 1;
            }
        }
    }
}
large = a[p + 1][i];
} else {
    max = max;
    large = large;
}
}
ramp = b[i];
b[i] = b[max];
b[max] = ramp;
for (j = 1; j <= n; j++) {
    temp = a[i][j];
    a[i][j] = a[max][j];
    a[max][j] = temp;
}

/* RIGHT SIDE MATRIX AFTER PIVOTING */

for (k = 0; k <= n + 1; k++) {
    for (i = k + 1; i <= n + 1; i++) {
        a[i][k] = a[i][k] / a[k][k];
        for (j = k + 1; j <= n + 1; j++)
            a[i][j] = a[i][j] - a[k][j] * a[i][k];
        b[i] = b[i] - a[i][k] * b[k];
    }
}

b[n + 1] = b[n + 1] / a[n + 1][n + 1];
for (i = n; i >= 0; i--) {
    add = b[i];
    for (k = i + 1; k <= n + 1; k++)
        add = add - b[k] * a[i][k];
    b[i] = add / a[i][i];
}
LOOP

DESCRIPTION: Thermal conductivity is a function of Temperature. unknown. So assume thermal conductivity at unknown temperature is equal to thermal conductivity at previous temperature. And then temperature will be calculated in loop.

```c
void loop()
{
    int i, j;
    int yes = 0;

    for(i=0; i<=n+1; i++)
    {
        if(_ABS((bl[i]-b[i])/bl[i]) > 0.00001)
        {
            yes = 1;
            break;
        }
    }

    if (yes == 1 && maxiter < 200)
    {
        maxiter++;
        for(i=0; i<=n+1; i++)
        {
            b1[i] = b[i];
        }
        thermal();
        aandbmatrix();
        guass();
        loop();
    }
    else{
        i = 99; /* calculation at n = i */
        printf("%lf\t%lf\n", t[xyz+1], b[i]);
    }
    tt();
}
```
AFTER EACH TIME STEP

DESCRIPTION: After each time step the calculated temperature becomes the initial one.

****************************************************
****************************************************
void tt()
    {
    int i;
    for (i = 0; i <= n + 1; i++) {
        ln=xyz+li
        T [i][ln]=b[i];
        k[ln][i]=k[xyz][i];
    }
    for (i = 0; i <= n; i++) {
        kl[ln][i]=kl[xyz][i];
    }
}

****************************************************
****************************************************
THERMAL

DESCRIPTION: The loop process continues at the same temperature after some iterations. So for each iteration, after temperature and thermal conductivity will be calculated by using this function.

****************************************************
****************************************************
void thermal()
    {
    int i;
    for (i = 0; i <= n+1; i++) {
        b2[i] = (b1[i] + b1[i + 1]) / 2.0;
    }
    for (i = 0; i <= n; i++) {
        k1[xyz][i] = 20.0; /*-4E-11*pow(b2[i],4) + 7E-08*pow(b2[i],3) -
                        3E-05*pow(b2[i],2) + 0.0102*b2[i] + 16.226*/
    }
    for (i = 0; i <= n + 1; i++) {
        k[xyz][i] = 20.0; /*-4E-11*pow(b1[i],4) + 7E-08*pow(b1[i],3) -
                           3E-05*pow(b1[i],2) + 0.0102*b1[i] + 16.226*/
    }
}
PROGRAM : MINIMIZATION

DESCRIPTION : This program is used to find temperature function using Minimization approach.

************************************************************************************************************************************************
/* Header Files */
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
/* Globals */
#define ftol 1.0e-10
#define sign(a,b) ((b) >=0.0 ? fabs(a) : -fabs(a))
float ax,xx,cx;
float time[100],temp[100];
float xl[100];
int nun,ndp;
float dff[100];
float fret;
float xmin;
/* Prototypes */
void dfpmin(float p[],int , int );
float func(float p[]);
void dffunc(float p[],float dff[]);
void linmin(float p[],float xi[],int );
void mnbrak(float ,float ,float,float pcom[],float xicom[]);
float fdlidim(float x,float p[],float xi[]);
float FMAX(float c,float d);
float dbrent(float pcom[],float xicom[]);
float dfldidim(float x, float pcom[], float xicom[]);
main()
{    
    int i;
    int iter;
    /*****************************************************************************/
    /*******************************************************************************/
    INPUT
   *******************************************************************************/
    printf("\nNo of Unknowns:");
    scanf("%d",&nun);
    printf("\nNo of datapoints:");
    scanf("%d",&ndp);
for(i=1;i<=ndp;i++)
{
    printf("Enter time[%d]:",i);
    scanf("%f",&time[i]);
    printf("Enter temp [%d] ":,i);
    scanf("%f",&temp[i]);
}

for(i=1;i<=nun;i++)
{
    printf("Unknown [%d] =",i);
    scanf("%f",&x1[i]);
}
dfpmin(x1,nun,iter);
/*Printing number of iterations*/
printf("Total iterations: %d\n",iter);
i=1;
printf("Unknown [%d] =%f\n",i,x1[1]);
i=2;
printf("Unknown [%d] =%f\n",i,x1[2]);
}

/********************************************
********************************************
dfpmin

DESCRIPTION:
The dfpmin function is used to find the location of minimum and the minimum value of the function. This is the main function which calls other function linmin to perform the above task. It calls func and dfunc to get the temperature function and to calculate dff w.r.t constants. Then it calls the linmin() to find the minimum.

Given a starting point p[1 ... n] that is a vector of length n, the Broyden Flether-Goldfarb-Shanno variant of Davidon-Fletcher-Powell minimization is performed on a function func(), using its gradient as calculated by a routine dfunc(). The convergence requirement on zeroing the gradient is input as ftol. Returned quantities are p[1 ... n] ( the location of the minimum), iter (the number of iterations that were performed), and fret (the minimum value of the function). The routine linmin() is called to perform approximate line minimizations.)

 ********************************************
 ********************************************

void dfpmin(float p[], int n, int iter)
{
#define nmax 50  
#define itmax 2000  
#define eps 1.0e-10  

float hessin[nmax][nmax], xi[nmax], g[nmax], dg[nmax];  
float hdg[nmax];  
float fp;  
int i, j;  
int its;  
float fac, fad, fae;

/* p is an array of unknown constants is the same as  
   xi as mentioned in main */

/* Calling function func */
fp=func(p);
/* Calculate starting gradient value*/
dfunc(p,dff);

/*Initialize the inverse Hessian to the unit matrix*/
for(i=1;i<=nun;i++)
{
    for(j=1;j<=nun;j++)
    {
        hessin[i][j]=0.0;
    }
    hessin[i][i]=1;
/* Initial line direction */
    xi[i]=-dff[i];
}

/* Main loop over the iterations*/
for(its=1;its<=itmax;its++)
{
    iter=its;

    /* The new function evaluation occurs in linmin();  
save the function value fp for the next line search.  
It is usually safe to ignore the value of check */
    linmin(p,xi,nun);
    if(2.*fabs(fret-fp)<=ftol*(fabs(fret)+fabs(fp)+eps))
        return;
    fp=fret;
/* save the old gradient */
    for(i=1;i<=nun;i++)
    {
        dg[i]=g[i];
    }
    fret=func(p);  
dfunc(p,g);
/* Get the new gradient */
/* Compute difference of gradients, and difference times
current matrix */
for (i = 1; i <= nun; i++)
{
  dg[i] = g[i] - dg[i];
}
for (i = 1; i <= nun; i++)
{
  hdg[i] = 0.;
  for (j = 1; j <= nun; j++)
  {
    hdg[i] = hdg[i] + hessin[i][j] * dg[j];
  }
}
/* Calculate dot products for the denominators */
fac = 0.;
fae = 0.;
for (i = 1; i <= nun; i++)
{
  fac = fac + dg[i] * xi[i];
  fae = fae + dg[i] * hdg[i];
}
/* Skip update if fac not sufficiently positive */
fac = 1. / fac;
fae = 1. / fae;
for (i = 1; i <= nun; i++)
{
  dg[i] = fac * xi[i] - fae * hdg[i];
}
/* The vector that makes BFGS different from DFP:
   The BFGS updating formula: */
for (i = 1; i <= nun; i++)
{
  for (j = 1; j <= nun; j++)
  {
    hessin[i][j] = hessin[i][j] + fac * xi[i] * xi[j] -
                   fae * hdg[i] * hdg[j] + fae * dg[i] * dg[j];
  }
}
/* Now calculate the next direction to go and go back
   for another iteration */
for (i = 1; i <= nun; i++)
{
  xi[i] = 0.;
  for (j = 1; j <= nun; j++)
  {
    xi[i] = xi[i] - hessin[i][j] * g[j];
  }
}
return;
/*********************************************************************************/

FUNCTION DESCRIPTION:
The function func() finds the function value
p is an array of unknown constants
same as xl as mentioned in main
**********************************************************************************/

float func(float p[])
{
    float a,b,g1[100],func1;
    float f[100];
    int i;
    /* Assigning p values to a,b... */
    /* g1 is an array of analytical function values*/
    a=p[1];
    b=p[2];

    /* Finding analytical function g1 values using
     Initial guess */
    for(i=1;i<=ndp;i++)
        g1[i]=a+b*time[i];

    /* initialize function f to zero */
    for(i=1;i<=ndp;i++)
        f[i]=0.0;
    /* calculating f */
    for(i=1;i<=ndp;i++)
        f[i]=(temp[i]-g1[i])*(temp[i]-g1[i]);
    /* Calculating func */
    func1=0.0;

    for(i=1;i<=ndp;i++)
        func1=func1+f[i];

    return func1 ;
}
DFUNC

DESCRIPTION:
Calculates the gradient value of the function
gl is a array of analytical function
p is a array of unknown constants same as x1

void dfunc(float p[],float dff[])
{
float a,b,gl[100];
int i;
/* Assigning array p to a,b... */
a=p[1];
b=p[2];

/* Calculating analytical function gl */
for(i=1;i<=ndpii;i++)
    gl[i]=a+b*time[i];

/* Initialize derivative of analytical function
  w.r.t all constants equal to zero */
for(i=1;i<=nun;i++)
    dff[i]=0.;

/* Calculation of derivatives w.r.t unknown constants */
for(i=1;i<=ndpii;i++)
{
    dff[1]=dff[1]-2.*(temp[i]-gl[i])
    dff[2]=dff[2]-2.*(temp[i]-gl[i])*time[i];
}

}

LINMIN

DESCRIPTION:
Given an n-dimensional point p[1..n] and an n-dimensional
direction xi[1...n], moves and resets p to where
the function func(p) takes on a minimum along the
direction xi and p, and replaces xi by the actual vector
displacement that p was moved. Also returns as fret the
value of func at the returned location p. This is actually
accomplished by calling the routines mnbrak and brent.

void linmin(float p[],float xi[],int nun)
#define nmax 50
#define tol 1.e-10
float pcom[nmax], xicom[nmax];
int ncom;
float fa, fb, fx;
int j;
ncom=nun;
for(j=1;j<=nun;j++)
{
    pcom[j]=p[j];
    xicom[j]=xi[j];
}
/* Initial guess for brackets */
ax=0.;
xx=1.;
/* Calling mnbrak function */
mnbrak(fa, fx, fb, pcom, xicom);
/* Calling dbrent function */
fret=dbrent(pcom, xicom);
printf("fret=%5.20lf\n", fret);
printf("xmin=%5.20lf\n", xmin);
/* Construct the vector results to return */
for(j=1;j<=nun;j++)
{
    xi[j]=xi[j]*xmin;
    p[j]=p[j]+xi[j];
}
return;
}
fb = fldim(xx,pcom,xicom);
/* Switch roles of a and b so that we can go downhill
in the direction from ax to xx */
if(fb>fa){
    dum=ax;
    ax=xx;
    xx=dum;
    dum=fb;
    fb=fa;
    fa=dum;
}
/* First guess for cx */
    cx=xx+gold*(xx-ax);
    fc=fldim(cx,pcom,xicom);
    /* Keep returning here until we bracket
Compute u by parabolic extrapolation from ax,xx,cx.
Tiny is used to prevent any possible division by zero.
*/
    if(fb>fc){
        r=(xx-ax)*(fb-fc);
        q=(xx-cx)*(fb-fa);
        u=xx-((xx-cx)*q-(xx-ax)*r)/(2.*sign(FMAX(fabs(q-r),tiny),q-r));
        ulim=xx+glimit*(cx-xx);
        /* We won't go farther than this. Test various possibilities
Parabolic u is between xx and cx : try it. */
        if((xx-u)*(u-cx)>0.){
            fu=fldim(u,pcom,xicom);
            /* Got a minimum between xx and cx */
            if(fu<fc){
                ax=xx;
                fa=fb;
                xx=u;
                fb=fu;
                return;
            }
            /* Got a minimum between ax and u. */
            else if(fu>fb){
                cx=u;
                fc=fu;
                return;
            }
        */ Parabolic fit was no use. Use default magnification */
            u=cx+gold*(cx-xx);
            fu=fldim(u,pcom,xicom);
            /* Parabolic fit is between cx and its allowed limit */
        }
        else if((cx-u)*(u-ulim)>0.){
            fu=fldim(u,pcom,xicom);
            if(fu<fc){
                xx=cx;
                cx=u;
                u=cx+gold*(cx-xx);
fb=fc;
fu=fldim(u,pcom,xicom);
}
else if((u-ulim)*(ulim-cx)>=0.){
    u=ulim;
    fu=fldim(u,pcom,xicom);
}
/* Reject parabolic u, use default magnification */
else{
    u=cx+gold*(cx-xx);
    fu=fldim(u,pcom,xicom);
    /* Eliminate oldest point and continue */
    ax=xx;
    xx=cx;
    cx=u;
    fa=fb;
    fb=fc;
    fc=fu;
}
return;
}

/*****************************************
******************************************
F1DIM
******************************************
******************************************
float fldim(float x,float p[],float xi[])
{
    #define nmax 50
    float ncom,pcom[nmax],xicom[nmax];
    float xt[nmax];
    float f1dim1;
    int j;
    ncom=nun;
    for(j=1;j<=ncom;j++)
    {
        pcom[j]=p[j];
        xicom[j]=xi[j];
    }
    for(j=1;j<=ncom;j++)
    {
        xt[j]=pcom[j]+x*xicom[j];
    }
    f1dim1=func(xt);
    return f1dim1;
}
/***************************************************/
*************
FMAX
*************
/***************************************************/

float FMAX(float c, float d)
{
    if (c >= d)
        return c;
    else
        return d;
}

/***************************************************/
*************
DBRENT
*************
/***************************************************/

DESCRIPTION:

Given a function f and its derivative function df, and given a bracketing triplet of abscissas ax, bx, cx [such that bx is between ax and cx, and f(bx) is less than both f(ax) and f(cx)], this routine isolates the minimum to a fractional precision of about tol using a modification of Brent's method that uses derivatives. The abscissa of the minimum is returned as xmin, and the minimum function value is returned as fx, the returned function value.

float dbrent(float pcom[], float xicom[])
{
    #define ITMAX 30000
    #define zeps 1.0e-5
    int iter, ok1, ok2;
    float a, b, d, d1, d2, du, dv, dw, dx, e = 0.0;
    float fu, fv, fw, fx, olde, toll, to12, u, u1, u2, v, w, x, xm;

    a = (ax < cx ? ax : cx);
    b = (ax > cx ? ax : cx);
    /* ax and bx must be in ascending order, but input abscissas need not be */
    v = xx;
    w = v;
    x = v;
    fx = f1dim(x, pcom, xicom);
    /* Initializations...... */
    fv = fx;
    fw = fx;
    dx = df1dim(x, pcom, xicom);
    dv = dx;
    dw = dx;
    /* Main program loop
All our housekeeping chores are doubled by the necessity of moving derivative values around as well as function values */
for(iter=1; iter<=ITMAX; iter++)
{
    xm=0.5*(a+b);
tol1=ftol*fabs(x)+zepl;
tol2=2.0*tol1;
    /* Test for done here */
    if(fabs(x-xm)<=(tol2-0.5*(b-a))){
        xmin=x;
        return fx;
    }
    /* Construct a trial parabolic fit */
    if(fabs(e)>tol1){
        /* Initialize these d’s to an out-of-bracket value */
        d1=2.0*(b-a);
        d2=d1;
        /* Secant method with one point And the other */
        if(dw!=dx) d1=(w-x)*dx/(dx-dw);
        if(dv!=dx) d2=(v-x)*dx/(dx-dv);
        /* Which of the two estimates of d shall we take?
           We will insist that they be within the bracket, and
           on the side pointed to by the derivative at x */
        ul=x+d1;
        u2=x+d2;
        ok1=((a-ul)*(ul-b)>0.0)&&(dx*d1<=0.0);
            ok2=((a-u2)*(u2-b)>0.0)&&(dx*d2<=0.0);
            olde=e;
        e=d;
        /* Take only an acceptable d, and if both are acceptable,
           then take the smallest one */
        if((ok1||ok2)) {
            if(ok1&ok2)
                d=(fabs(d1) < fabs(d2) ? d1 : d2);
            else if(ok1)
                d=d1;
            else
                d=d2;
            if(fabs(d)<=fabs(0.5*olde)) {
                u=x+d;
                if((u-a)<tol2&&b-u)<tol2) d=sign(tol1, xm-x);
                d=-d;
                xmin=x;
                printf("xminnn=%5.20lf\n",xmin);
                printf("d=%5.20f\n",d);
                printf("tol1=%5.20lf\n",tol1);
                printf("xm-x=%5.20lf\n",xm-x);
            }
            else {
                
            }
        }
    }
}
d=0.5*(e=(dx>=0.0 ? a-x : b-x));
else{
  d=0.5*(e=(dx>=0.0 ? a-x : b-x));
}
else{
  d=0.5*(e=(dx>=0.0 ? a-x : b-x));
}
if(fabs(d>=toll){
  u=x+d;
  fu=fldim(u,pcom,xicom);
} else{
  u=x+sign(toll,d);
  fu=fldim(u,pcom,xicom);
  /* If the minimum step in the downhill direction takes
   us uphill, then we are done */
  if(fu>fx) {
    xmin=x;
    return fx;
  }
  /* Now all the housekeeping, sigh */
  du=dfldim(u,pcom,xicom);
  if(fu<=fx){
    if(u>=x)
      a=x;
    else
      b=x;
    v=w;
    fv=fw;
    dv=dw;
    w=x;
    fw=fx;
    dw=dx;
    x=u;
    fx=fu;
    dx=du;
    xmin =x;
  } else{
    if(a<x)
      a=u;
    else
      b=u;
    if(fu<=fw || w==x) {
      v=w;
      fv=fw;
      dv=dw;
w=u;
fw=fu;
dw=du;
}
else if (fu <= fv || v==x || v==w) {
  v=u;
fv=fu;
dv=du;
}
}
}
printf(" xmin: %5.20lf\n", xmin);
xmin = x;
printf(" xmin1: %5.20lf\n", xmin);
printf(" I am here\n");
return fx;
}

/*********************
********DF1DIM
**********************/
float df1dim(float x, float pcom[], float xicom[])
{
  float xt[100], df[100];
  int j;
  float df1dim1;
  for(j=1; j<=nun; j++)
  {
    xt[j]=pcom[j]+x*xicom[j];
  }
  dfunc(xt, df);
  df1dim1=0.;
  for(j=1; j<=nun; j++)
  {
    df1dim1=df1dim1+df[j]*xicom[j];
  }
  return df1dim1;
}
Minimization program in FORTRAN

c This program is used to find temperature function using
Minimization approach.

c Assume temperature profile gl[i]= x1[1]+x1[2]t
Depending upon the function the dff[] will be
changed in DFUNC function. dff[] is a derivative
of gl[] with respect to unknown constants.
In my case I found dff[1] and dff[2] when differentiating
the gl=x1[1]+x1[2]t with respect to x1[1] & x1[2], respectively.

c
The Input Data:
Enter number of data points (ndp)
Enter time data and corresponding temperature data
Enter number of unknowns (nun)

Reading data from data file inf.dat
datafile inf.dat consists of two columns i.e,
time and temperature, respectively.
program TO FIND TEMPERATURE FUNCTION

c dimension x1(50)
common/coml/texp(100),time(100),nun,ndp
x1 is array of unknown constants
time is array of time steps
texp is array of temperature(experimental)
values corresponding time
nun is number of unknowns
ndp is number of data points
open(1, file='inf2.dat', status='old')
Data file consists of time and texp values
pi=acos(-1.)
Mathematical constant PI

c ftol=1.0e-10
Assuming tolerance as above

Initializing nun and ndp
nun=2
ndp=10

Reading and writing time and texp from data file
do i=1,ndp
read(1,*), time(i), texp(i)
write(6,*), 'time=', time(i)
write(6,*), 'temp=', texp(i)
end do

c
SUBROUTINE DFPMIN(P,N,FTOL,ITER,FRET)

The dfpmin function is used to find the location of minimum and the minimum value of the function. This is the main function which calls other function linmin to perform the above task. It calls func and dfunc to get the temperature function and to calculate df w.r.t constants. Then it calls the linmin() to find the minimum.

Given a starting point p[1...n] that is a vector of length n, the Broyden Flether-Goldfarb-Shanno varient of Davidon-Fletcher-Powell minimization is performed on a function func(), using its gradient as calculated by a routine dfunc(). The convergence requirement on zeroing the gradient is input as ftol. Returned quantities are p[1...n] (the location of the minimum), iter (the number of iterations that were performed), and fret (the minimum value of the function).

The routine linmin() is called to perform approximate line minimizations.

DFPMIN:

SUBROUTINE DFPMIN(P,N,FTOL,ITER,FRET)

PARAMETER (NMAX=50,ITMAX=2000,EPS=1.0E-10)

INITIAL GUESS FOR UNKNOWN CONSTANTS

X1(1)=0.
X1(2)=-0.

CALLING DFPMIN FUNCTION

CALL DFPMIN(X1,NUN,FTOL,ITER,FRET)

PRINTING NUMBER OF ITERATIONS

121 FORMAT(/)
WRITE(6,*) 'NO ITER = ',ITER

CONTINUE

WRITE(6,121)
WRITE(6,121)

PRINTING FINAL VALUES OF CONSTANTS

WRITE(6,*) 'A1 = ',X1(1)
WRITE(6,*) 'B1 = ',X1(2)

WRITE(6,121)

STOP
END
dimension p(50), hessin(nmax,nmax), xi(nmax), g(nmax), dg(nmax)
dimension hdg(nmax)
c
p is a array of unknown constants the same as c
x1 as mentioned in main
c
calling function func
c
calculate starting function value
fp=func(p)
c
calculate starting gradient value
calling function dfunc
call dfunc(p,g)
c
initialize the inverse hessian to the unit matrix
do 12 i=1,n
do 11 j=1,n
hessin(i,j)=0.0
11 continue
hessin(i,i)=1.
c
initial line direction
xi(i) = -g(i)
12 continue
c
main loop over the iterations
do 24 its=1, itmax
iter=its
write(6,*) iter
c
the new function evaluation occurs in linmin(); save the
c
function value fp for the next line search.
c
it is usually safe to ignore the value of check
call linmin(p, xi, n, fret)
if(2.*abs(fret-fp) .le. ftol*(abs(fret)+abs(fp)+eps)) return
fp=fret

c
save the old gradient
do 13 i=1,n
dg(i)=g(i)
13 continue
c
fret=func(p)
call dfunc(p,g)
c
get the new gradient
c
c
compute difference of gradients and difference times
current matrix
do 14 i=1,n
dg(i)=g(i)-dg(i)
14 continue
c
do 16 i=1,n
hdg(i)=0.
do 15 j=1,n
hdg(i)=hdg(i)+hessin(i,j)*dg(j)
15 continue
Calculate dot products for the denominators
fac=0.
fae=0.
do 17 i=1,n
fac=fac+dg(i)*xi(i)
fae=fae+dg(i)*hdg(i)
17 continue

Skip update if fac not sufficiently positive
fac=1./fac
fad=1./fae
do 18 i=1,n
dg(i)=fac*xi(i)-fad*hdg(i)
18 continue

The vector that makes BFGS different from DFP:
The BFGS updating formula:
do 21 i=1,n
do 19 j=1,n
hessin(i,j)=hessin(i,j)+fac*xi(i)*xi(j)
+ -fad*hdg(i)*hdg(j)+fae*dg(i)*dg(j)
19 continue
21 continue

Now calculate the next direction to go and go back
for another iteration
do 23 i=1,n
xi(i)=0.
do 22 j=1,n
xi(i)=xi(i)-hessin(i,j)*g(j)
22 continue
23 continue
24 continue

return
end

FUNCTION()

The function func() finds the function value
FUNCTION:

function func(x)
dimension x(50),f(100),g1(100)
c
common/coml/texp(100),time(100),nun,ndp
c
x is a array of unknown constants
c
same as x1 as mentioned main
Assigning x values to a, b...
gl is an array of analytical function values

a = x(1)
b = x(2)

Finding analytical function gl values using
Initial guess

do i = 1, ndp
gl(i) = a + b * time(i)
end do

initialize function f to zero

do j = 1, ndp
  f(j) = 0.0
end do

calculating f

do i = 1, ndp
  f(i) = (texp(i) - gl(i)) * (texp(i) - gl(i))
end do

Calculating func
func = 0.0
do j = 1, ndp
  func = func + f(j)
end do

return
end

************************************************************************
DFUNC()
************************************************************************
Calculates the gradient value of the function

DFUNC:
subroutine dfunc(x, dff)

dimension gl(100)
dimension x(50), dff(50)
common/com1/texp(100), time(100), nun, ndp

gl is an array of analytical function
x is an array of unknown constants same as x1
in main
nun is number of unknowns
ndp is number of data points

Assigning array x to a, b...
a = x(1)
b = x(2)

Calculating analytical function gl
do i = 1, ndp
c

gl(i)=a+b*time(i)
end do

c

Initialize derivative of analytical function
w.r.t all constants equal to zero

do i=1,nun
dff(i)=0.
end do

c
Calculation of derivatives w.r.t unknown constants

do i=1,ndp
dff(1)=dff(1)-2.*(texp(i)-gl(i))
dff(2)=dff(2)-2.*(texp(i)-gl(i))\*time(i)
end do

c
return
end

c*************************************************

LINMIN()
*************************************************

given an n-dimensional point p[1..n] and an n-dimensional
direction xi[1...n], moves and resets p to where
the function func(p) takes on a minimum along the
direction xi and p and replaces xi by the actual vector
displacement that p was moved. Also returns as fret the
value of func at the returned location p. This is actually
accomplished by calling the routines mnbrak and brent.

LINMIN:

subroutine linmin(p,xi,n,fret)

parameter (nmmax=50,tol=1.e-10)
external fdlmin,dfdlmin
dimension p(50),xi(50)

common/f1com/ncom,pcom(nmax),xicom(nmax)
ncom=n
do 11 j=1,n
  pcom(j)=p(j)
  xicom(j)=xi(j)
11 continue

Initial guess for brackets
ax=0.
xx=1.

Calling mnbrak function
call mnbrak(ax,xx,bx,fa,fx,fb,fdlmin)
write(6,*)'xmin=',xmin

Calling dbrent function
fret=dbrent(ax,xx,bx,ftol,xmin)
write(6,*)'fret=',fret

Construct the vector results to return
do 12 j=1,n
  xi(j)=xmin*xi(j)
p(j)=p(j)+xi(j)
12 continue

return
end

***********************************************
F1DIM()
***********************************************

function f1dim(x)
parameter(nmax=50)
common/f1com/ncom,pcom(nmax),xicom(nmax)
dimension xt(nmax)
do 11 j=1,ncom
  xt(j)=pcom(j)+x*xicom(j)
11 continue

f1dim=tunc(xt)
return
end

***********************************************
DF1DIM()
***********************************************

function df1dim(x)
parameter(nmax=50)
common/f1com/ncom,pcom(nmax),xicom(nmax)
dimension xt(nmax),df(nmax)
do 11 j=1,ncom
  xt(j)=pcom(j)+x*xicom(j)
11 continue

call dfunc(xt,df)
df1dim=0.
do 12 j=1,ncom
  df1dim=df1dim+df(j)*xicom(j)
12 continue

return
end

***********************************************
MNBRAK()
***********************************************

MNBRAK:
Given a function func, and given distinct initial
points ax, and bx.
This subroutine searches in the downhill direction
and returns new points, ax, bx, cx that bracket a
minimum of the function. Also returned are the function values at the three points fa, fb and fc

```fortran
subroutine mnbrak(ax,bx,cx,fa,fb,fc,fldim)

parameter(gold=1.618034,glimit=100.,tiny=1.e-20)
external fldim

fa=fldim(ax)
fb=fldim(bx)

Switch roles of a and b so that we can go downhill in the direction from ax to bx
if(fb.gt.fa) then
  dum=ax
  ax=bx
  bx=dum
  dum=fb
  fb=fa
  fa=dum
endif

First guess for cx
cx=bx+gold*(bx-ax)
fc=fldim(cx)

Keep returning here until we bracket Compute u by parabolic extrapolation from ax,bx,cx.
tiny is used to prevent any possible division by zero.
if(fb.ge.fc) then
  r=(bx-ax)*(fb-fc)
  q=(bx-cx)*(fb-fa)
  u=bx-((bx-cx)*q-(bx-ax)*r)/(2.*sign(max(abs(q-r),tiny),q-r))
  ulim=bx+glimit*(cx-bx)
endif

We won't go farther than this. Test various possibilities
Parabolic u is between bx and cx : try it.
if((bx-u)*(u-cx).gt.0.) then
  fu=fldim(u)
Got a minimum between bx and cx
  if(fu.lt.fc) then
    ax=bx
    fa=fu
    bx=u
    fb=fu
    return
Got a minimum between ax and u.
  else if(fu.gt.fb) then
    cx=u
    fc=fu
    return
  endif
Parabolic fit was no use. Use default magnification
  u=cx+gold*(cx-bx)
  fu=fldim(u)
Parabolic fit is between cx and its allowed limit
```


else if((cx-u)*(u-ulim).gt.0.) then
  fu=fldim(u)
  if(fu.lt.fc) then
    bx=cx
    cx=u
    u=cx+gold*(cx-bx)
    fb=fc
    fc=fu
    fu=fldim(u)
  endif
else if((u-ulim)*(ulim-cx).ge.0.) then
  u=ulim
  fu=fldim(u)
else
  Reject parabolic u, use default magnification
  u=cx+gold*(cx-bx)
  fu=fldim(u)
endif
Eliminate oldest point and continue
ax=bx
bx=cx
cx=u
fa=fb
fb=fc
fc=fu
go to 1
endif
return
end

******************************************************************************
DBREN\t
******************************************************************************
Given a function f and its derivative function df,
and given a bracketing triplet of abscissas ax, bx, cx [such that bx is between ax and cx, and f(bx)
is less than both f(ax) and f(cx) ], this routine
isolates the minimum to a fractional precision of about
tol using a modification of Brent’s method that uses
derivatives. The abscissa of the minimum is returned
as xmin, and the minimum function value is returned
as fx, the returned function value.
function dbrent(ax,bx,cx,tol,xmin)

parameter(itmax=30000,zeps=1.0e-15)
external fldim,dfldim
logical ok1,ok2
a=min(ax,cx)
b=max(ax,cx)
v=bx
w=v
x=v
c This will be the distance moved on the step before last
e=0.
c ax and bx must be in ascending order, but input
c abscissas need not be
fx=f1dim(x)
c Initializations......
fv=fx
fw=fx
dx=df1dim(x)
dv=dx
dw=dx
c Main program loop
c All our housekeeping chores are doubled by the necessity
c of moving derivative values around as well as
function values
do 11 iter=1,imax
xm=0.5*(a+b)
tol1=tol*abs(x)+ze ps
tol2=2.*tol1
c Test for done here
if(abs(x-xm).le.(tol2-.5*(b-a))) go to 3
c Construct a trial parabolic fit
if(abs(e).gt.tol1) then
   c Initialize these d's to
   an
   out-of-bracket value
   dl=2.*(b-a)
d2=d1
c Secant method with one point And the other
   if(dw.ne.dx) d1=(w-x)*dx/(dx-dw)
   if(dv.ne.dx) d2=(v-x)*dx/(dx-dv)
c Which of the two estimates of d shall we take?
c We will insist that they be within the bracket, and
c on the side pointed to by the derivative at x
u1=x+d1
u2=x+d2
write(6,*)'xmindbrent=',xmin
ok1=([a-u1]*(u1-b).gt.0.).and.(dx*d1.le.0.)
ok2=([a-u2]*(u2-b).gt.0.).and.(dx*d2.le.0.)
olde=e
e=d
c Take only an acceptable d, and if both are acceptable,
c then take the smallest one
if(.not.(ok1.or.ok2)) then
go to 1
else if(ok1.and.ok2) then
   if(abs(d1).lt.abs(d2)) then
d=d1
else
d=d2
endif
else if(ok1) then
d=d1
else
d=d2
endif
if(abs(d).gt.abs(0.5*olde)) go to 1
u=x+d
if(u-a.lt.tol2.or.b-u.lt.tol2) d=sign(toll,xm-x)
write(6,*)'d=',d
write(6,*)'toll=',toll
write(6,*)'xm-x=',xm-x
go to 2
endif
1 if(dx.ge.0.) then
c Decide which segment by the sign of the derivative
e=a-x
else
e=b-x
endif
d=0.5*e
2 if(abs(d).ge.toll) then
u=x+d
fu=fldim(u)
else
u=x+sign(toll,d)
fu=fldim(u)
c If the minimum step in the downhill direction takes
c us uphill, then we are done
if(fu.gt.fx) go to 3
endif
c Now all the housekeeping, sigh
du=dfldim(u)
if(fu.le.fx) then
if(u.ge.x) then
a=x
else
b=x
endif
v=w
fv=fw
dv=dw
w=x
fw=fx
dw=dx
x=u
fx=fu
dx=du
else
if(u.lt.x) then
a=u
else
b=u
endif
if(fu.le.fw.or.w.eq.x) then
v=w
fv=fw
dv=dw
w=u
fw=fu
dw=du
else if(fu.le.fv .or. v.eq.x .or. v.eq.w) then
v=u
fv=fu
dv=du
endif
endif
continue
3 xmin=x
dbrent=fx
return
end
APPENDIX B

******************************************************************************
******************************************************************************
Program: This program is used to calculate average of every 100 data
points.
Written by Kumar Anagurthi
******************************************************************************
******************************************************************************
#include<stdio.h>
******************************************************************************
Global Data
******************************************************************************

double time[12000], temp[12000];
int n;

main()
{
    int i;
    double sumTemp=0.0; /* Sum of Temperatures */
    double sumTime=0.0; /* Sum of Time */
    int j;
    n=11000;  /* No of Data Points */
    for(i=0;i<n;i++)
    {
        scanf("%lf\t%lf\n", &time[i], &temp[i]);
    }
    for(i=0;i<n;i++)
    {
        sumTemp = sumTemp + temp[i];
        sumTime = sumTime + time[i];
        j=(i%100);
        if(j==0)
        {
            if(i!=0)
                printf("%lf\t%lf\n", sumTime/100, sumTemp/100);
            if(i==0)
                printf("%lf\t%lf\n", sumTime, sumTemp);
            sumTemp=0.0;
            sumTime=0.0;
        }
    }
}
Program: This program is used to solve the Inverse Heat Conduction Problem.
Written by Kumar Anagurthi

#include<stdio.h>
#include<math.h>
#include<stdlib.h>

Global Data

double t[12500], Texp[12500], a[125][125], b[125];
double c[125][125];
int Kk;
int nc;

void guass();

Main Program

main()
{
    Local Data

    int i,j;
    int n,m;
    double l, alpha, k, p, temperature;
    double sum1, sum2, sum3, sum4, sum5, sum6, sum7, sum8;
    double sum9, sum10, sum11, sum12, sum13, sum14, sum15, sum16;
    double sum17, sum18, sum19, sum20, sum21, sum22;
    double sum23, sum24, sum25, sum26, sum27, sum28;
    double sum29, sum30, sum31, sum32, sum33, sum34;
    double sum35, sum36, sum37, sum38, sum39, sum40;

    The Following Data should be entered

    alpha=0.0000039;
k=17.0; /* Thermal Conductivity */
l=0.00635; /* Length of the beam */
n=50; /* No. of Iterations */
m=110; /* No. of Data Points */
nc=6; /* No. of Coefficients */
/******************************************************

Input File
*******************************************************/

for(i=0;i<=m;i++){  
    scanf("%lf\t%lf", &t[i], &Texp[i]);
}

sum1=0.0;
for(i=1;i<=n;i=i+2){
    sum1=sum1+(1.0/pow(i, 4));
}

for(j=0;j<=m;j++){  
    sum2=0.0;
    for(i=1;i<=n;i=i+2){
        sum2=sum2+((-i*i*M_PI*M_PI)+12.0)/(pow(i, 4))
        *exp(-alpha*pow((i*M_PI/l), 2)*t[j]);
    }
}

sum3=0.0;
for(i=2;i<=n;i=i+2){
    sum3=sum3+((exp(-alpha*pow((i*M_PI/l), 2)*t[j]))/(i*i));
}

sum4=0.0;
for(i=1;i<=n;i=i+2){
    sum4=sum4+(exp(-alpha*pow((i*M_PI/l), 2)*t[j])/(pow(i, 4)));
}

for(i=0;i<=m;i++){  
    scanf("%lf\t%lf", &t[i], &Texp[i]);
}

sum5=0.0;
for(i=1;i<=n;i=i+2){
    p=alpha*pow((i*M_PI/l), 2);
    sum5=sum5+((1.0/(i*i))*(t[j]/p)-(1.0/(p*p)));
}

sum6=0.0;
for(i=1;i<=n;i=i+2){
    p=alpha*pow((i*M_PI/l), 2);
    sum6=sum6+((-i*i*M_PI*M_PI+12.0)/(pow(i, 4)*p));
}

sum7=0.0;
for(i=2;i<=n;i=i+2){
    p=alpha*pow((i*M_PI/l), 2);
}
sum7 = sum7 + (1.0/(1.0*i*p));
}

sum8 = 0.0;
for (i = 1; i <= n; i = i + 2) {
    p = alpha*pow(i*M_PI/1, 2);
    sum8 = sum8 + (exp(-alpha*pow((i*M_PI/1, 2)*t[j]) / (-1.0*i*p));
}

sum9 = 0.0;
for (i = 1; i <= n; i = i + 2) {
    p = alpha*pow(i*M_PI/1, 2);
    sum9 = sum9 + (exp(-alpha*pow((i*M_PI/1, 2)*t[j])
                    *(12.0 - (1.0*i*M_PI*M_PI)) / (p*pow(i, 4)));
}

sum10 = 0.0;
for (i = 2; i <= n; i = i + 2) {
    p = alpha*pow(i*M_PI/1, 2);
    sum10 = sum10 + (exp(-alpha*pow((i*M_PI/1, 2)*t[j]) / (p*pow(i, 2));
}

c[1][j] = ((alpha * t[j] * t[j]) / (2.0*k1)) + ((t[j]) / (12.0*k))
-(((24.0*alpha)/(k*1*M_PI*M_PI)) * sum5)
-(((2.0*1)/(k*pow(M_PI, 4))) * (sum6 + (M_PI*M_PI*sum7)))
+(((24.0*alpha)/(k*1*M_PI*M_PI)) * sum8)
+(((2.0*1)/(k*pow(M_PI, 4))) * (sum9 + (M_PI*M_PI*sum10)));

sum11 = 0.0;
for (i = 1; i <= n; i = i + 2) {
    p = alpha*pow(i*M_PI/1, 2);
    sum11 = sum11 + (((t[j] * t[j]) / p) - ((2.0*t[j]) / (p*p))
                      + (2.0 / (p*p*p)) / (i*i));
}

sum12 = 0.0;
for (i = 1; i <= n; i = i + 2) {
    p = alpha*pow(i*M_PI/1, 2);
    sum12 = sum12 + (((-1.0*i*M_PI*M_PI + 12.0) / pow(i, 4))
                      * ((2.0*t[j]) / p) - (2.0 / (p*p)));
}

sum13 = 0.0;
for (i = 2; i <= n; i = i + 2) {
    p = alpha*pow(i*M_PI/1, 2);
    sum13 = sum13 + (((2.0*t[j]) / p) - (2.0 / (p*p)) * (1.0 / (i*i)));
}
sum14=0.0;
for(i=1;i<=n;i=i+2){
    p=alpha*pow(i*M_PI/l,2);
    sum14=sum14+((exp(-alpha*pow((i*M_PI/l),2)*t[j]))
    *(2.0/(i*i*pow(p,3))));
}

sum15=0.0;
for(i=1;i<=n;i=i+2){
    p=alpha*pow(i*M_PI/l,2);
    sum15=sum15+((-i*i*M_PI*M_PI)+12.0)*(-2.0/(pow(i,4)*p*p))
    *(exp(-alpha*pow((i*M_PI/l),2)*t[j]));
}

sum16=0.0;
for(i=2;i<=n;i=i+2){
    p=alpha*pow(i*M_PI/l,2);
    sum16=sum16+exp(-alpha*pow((i*M_PI/l),2)*t[j])
    *(-2.0/(i*i*p*p));
}

c[2][j]=((alpha*pow(t[j],3))/(3.0*k*l))
    +((1*t[j]*t[j])/(12.0*k))
    -((24.0*alpha)/(k*M_PI*M_PI)*sum11)
    -((2.0*1)/(k*pow(M_PI,4))*(sum12+(M_PI*M_PI*sum13)))
    +((24.0*alpha)/(k*M_PI*M_PI)*sum14)
    +((2.0*1)/(k*pow(M_PI,4))*(sum15+(M_PI*M_PI*sum16)));

sum17=0.0;
for(i=1;i<=n;i=i+2){
    p=alpha*pow(i*M_PI/l,2);
    sum17=sum17+((t[j]*t[j]*t[j])/p)
    -((3.0*t[j]*t[j])/(p*p))
    +((6.0*t[j])/(p*p*p))
    -(6.0/(p*p*p*p))/(i*i));
}

sum18=0.0;
for(i=1;i<=n;i=i+2){
    p=alpha*pow(i*M_PI/l,2);
    sum18=sum18+((-i*i*M_PI*M_PI)+12.0)/pow(i,4))
    *(((3.0*t[j]*t[j])/p)
    -((6.0*t[j])/(p*p))+(6.0/(p*p*p))));
}

sum19=0.0;
for(i=2;i<=n;i=i+2){
    p=alpha*pow(i*M_PI/l,2);
    sum19=sum19+((3.0*t[j]*t[j])/p)
-((6.0*t[j]/(p*p))
+((6.0)/(p*p*p))*((1.0/(i*i)));

sum20=0.0;
for(i=1;i<=n;i=i+2){
p=alpha*pow(i*M_PI/l,2);
sum20=sum20+(exp(-alpha*pow((i*M_PI/l),2)*t[j])
*(-6.0/(i*i*pow(p,4))));
}

sum21=0.0;
for(i=1;i<=n;i=i+2){
p=alpha*pow(i*M_PI/l,2);
sum21=sum21+((-(i*M_PI*M_PI)+12.0)*(6.0/(pow(i,4)*p*p*p))
*(exp(-alpha*pow((i*M_PI/l),2)*t[j]));
}

sum22=0.0;
for(i=2;i<=n;i=i+2){
p=alpha*pow(i*M_PI/l,2);
sum22=sum22+((exp(-alpha*pow((i*M_PI/l),2)*t[j])
*(6.0/(i*i*p*p*p)));
}

c[3][j]=((alpha*pow(t[j],4))/(4.0*k*1))
+((1*pow(t[j],3))/(12.0*k))
-((24.0*alpha)/(k*M_PI*M_PI)*sum17)
-(((2.0)*1)/(k*pow(M_PI,4))*(sum18+(M_PI*M_PI*sum19))
+((24.0*alpha)/(k*M_PI*M_PI)*sum20)
+((2.0)*1)/(k*pow(M_PI,4))*(sum21+(M_PI*M_PI*sum22)));

sum23=0.0;
for(i=1;i<=n;i=i+2){
p=alpha*pow(i*M_PI/l,2);
sum23=sum23+((((t[j]*t[j]*t[j]*t[j])/p)
-((4.0*t[j]*t[j]*t[j])/p*p))
+((12.0*t[j]*t[j])/(p*p*p))
-((24.0*t[j])/(p*p*p))
+(24.0/pow(p,5)))/(i*i));
}

sum24=0.0;
for(i=1;i<=n;i=i+2){
p=alpha*pow(i*M_PI/l,2);
sum24=sum24+((-(i*M_PI*M_PI)+12.0)/pow(i,4))*(((4.0*t[j]*t[j]*t[j])/p)
-((12.0*t[j]*t[j])/(p*p*p))+(24.0*t[j])/(p*p*p))-
(24.0/pow(p,4)));
}

sum25=0.0;
for(i=2;i<=n;i=i+2){
\[ p = \alpha \cdot \text{pow}(i \cdot M_{\text{PI}}/l, 2); \]
\[ \text{sum25} = \text{sum25} + (((4.0 \cdot t[j] \cdot t[j] \cdot t[j]) / p) \]
\[ - ((12.0 \cdot t[j] \cdot t[j]) / (p \cdot p)) + ((24.0 \cdot t[j]) / (p \cdot p \cdot p)) \]
\[ - (24.0 / \text{pow}(p, 4)) \cdot (1.0 / (i \cdot i)); \]

\[ \text{sum26} = 0.0; \]
\[ \text{for}(i=1; i<=n; i=i+2) \{
\quad p = \alpha \cdot \text{pow}(i \cdot M_{\text{PI}}/l, 2);
\quad \text{sum26} = \text{sum26} + ((\exp(-
\alpha \cdot \text{pow}(i \cdot M_{\text{PI}}/l, 2) \cdot t[j]) \cdot (24.0 / (i \cdot i \cdot \text{pow}(p, 5))));
\}\]

\[ \text{sum27} = 0.0; \]
\[ \text{for}(i=1; i<=n; i=i+2) \{
\quad p = \alpha \cdot \text{pow}(i \cdot M_{\text{PI}}/l, 2);
\quad \text{sum27} = \text{sum27} + (((-i \cdot i \cdot M_{\text{PI}} \cdot M_{\text{PI}} + 12.0) \cdot (-24.0 / (\text{pow}(i, 4) \cdot p \cdot p \cdot p \cdot p))) \cdot (\exp(-\alpha \cdot \text{pow}(i \cdot M_{\text{PI}}/l, 2) \cdot t[j]));
\}\]

\[ \text{sum28} = 0.0; \]
\[ \text{for}(i=2; i<=n; i=i+2) \{
\quad p = \alpha \cdot \text{pow}(i \cdot M_{\text{PI}}/l, 2);
\quad \text{sum28} = \text{sum28} + (((-((i \cdot i \cdot M_{\text{PI}} \cdot M_{\text{PI}}) + 12.0) \cdot (-24.0 / (i \cdot i \cdot \text{pow}(p, 5)))) \cdot (\exp(-\alpha \cdot \text{pow}(i \cdot M_{\text{PI}}/l, 2) \cdot t[j]));
\}\]

\[ c[4][j] = ((\alpha \cdot \text{pow}(t[j], 5)) / (5.0 \cdot k \cdot l)) + ((l \cdot \text{pow}(t[j], 4)) / (12.0 \cdot k)) \]
\[ - ((24.0 \cdot \alpha \cdot \text{pow}(t[j], 4)) \cdot \text{sum23}) \]
\[ - ((2.0 \cdot 1) / (k \cdot \text{pow}(M_{\text{PI}}, 4))) \cdot \text{sum24} + (M_{\text{PI}} \cdot M_{\text{PI}} \cdot \text{sum25}) \]
\[ + ((24.0 \cdot \alpha \cdot \text{pow}(t[j], 4)) \cdot \text{sum25}) \]
\[ + ((2.0 \cdot 1) / (k \cdot \text{pow}(M_{\text{PI}}, 4))) \cdot (\text{sum27} + (M_{\text{PI}} \cdot M_{\text{PI}} \cdot \text{sum28})); \]

\[ \text{sum29} = 0.0; \]
\[ \text{for}(i=1; i<=n; i=i+2) \{
\quad p = \alpha \cdot \text{pow}(i \cdot M_{\text{PI}}/l, 2);
\quad \text{sum29} = \text{sum29} + (((\text{pow}(t[j], 5) / p) - ((5.0 \cdot \text{pow}(t[j], 4)) / (p \cdot p)) \]
\[ + ((20.0 \cdot t[j] \cdot t[j] \cdot t[j] \cdot t[j]) / (p \cdot p \cdot p \cdot p)) \]
\[ - (60.0 \cdot t[j] \cdot t[j]) / (p \cdot p \cdot p \cdot p)) \]
\[ + ((120.0 \cdot t[j]) / (\text{pow}(p, 5)) - (120.0 / \text{pow}(p, 6))) / (i \cdot i));
\}\]

\[ \text{sum30} = 0.0; \]
\[ \text{for}(i=1; i<=n; i=i+2) \{
\quad p = \alpha \cdot \text{pow}(i \cdot M_{\text{PI}}/l, 2);
\quad \text{sum30} = \text{sum30} + (((-i \cdot i \cdot M_{\text{PI}} \cdot M_{\text{PI}}) + 12.0) / (\text{pow}(i, 4))) \]
\[ - ((5.0 \cdot t[j] \cdot t[j] \cdot t[j] \cdot t[j]) / p) \]
\[ - ((20.0 \cdot t[j] \cdot t[j] \cdot t[j]) / (p \cdot p \cdot p)) \]
\[ + (60.0 \cdot t[j] \cdot t[j]) / (p \cdot p \cdot p \cdot p)) \]
\[ - (120.0 \cdot t[j]) / (\text{pow}(p, 4)) + (120.0 / \text{pow}(p, 5));
\}\]
\begin{verbatim}
sumJ1=0.0;
for(i=2;i<=n;i=i+2){
p=alpha*pow(i*M_PI/l,2);
sumJ1=sumJ1+(((5.0*t[j]*t[j]*t[j]*t[j])/p)
-((20.0*t[j]*t[j]*t[j])/(p*p))
+((60.0*t[j]*t[j])/(p*p*p))
-((120.0*t[j])/pow(p,4))+(120.0/pow(p,5)))*(1.0/(i*i));
}

sumJ2=0.0;
for(i=1;i<=n;i=i+2){
p=alpha*pow(i*M_PI/l,2);
sumJ2=sumJ2+((exp(-alpha*pow((i*M_PI/l),2))*t[j])
*(-120.0/(i*i*pow(p,6))));
}

sumJ3=0.0;
for(i=1;i<=n;i=i+2){
p=alpha*pow(i*M_PI/l,2);
sumJ3=sumJ3+((-1*(i*i*M_PI*M_PI)+12.0)
*(120.0/(pow(i,4)*p*p*p*p*p))
*(exp(-alpha*pow((i*M_PI/l),2))*t[j]));
}

sumJ4=0.0;
for(i=2;i<=n;i=i+2){
p=alpha*pow(i*M_PI/l,2);
sumJ4=sumJ4+((exp(-alpha*pow((i*M_PI/l),2))*t[j])
*(120.0/(i*i*p*p*p*p*p)));
}

c[5][j]=((alpha*pow(t[j],6))/(6.0*k*l))
+((1*pow(t[j],5))/(12.0*k))
-((24.0*alpha/(k*l*M_PI*M_PI))*sum29)
-(((2.0)*k/pow(M_PI,4))*(sum30+(M_PI*M_PI*sum31))
+((24.0*alpha)/(k*l*M_PI*M_PI))*sum32)
+(((2.0)*k/pow(M_PI,4))*(sum33+(M_PI*M_PI*sum34)));

sumJ5=0.0;
for(i=1;i<=n;i=i+2){
p=alpha*pow(i*M_PI/l,2);
sumJ5=sumJ5+(((pow(t[j],6))/p)-((6.0*pow(t[j],5))/(p*p))
+((30.0*pow(t[j],4))/(p*p*p))
-((120.0*pow(t[j],3))/(p*p*p*p))
+((360.0*pow(t[j],2))/pow(p,5))
-((720.0*720.0)/pow(p,6))+(720.0/pow(p,7)))/((i*i));
}

sumJ6=0.0;
\end{verbatim}
for(i=1;i<=n;i=i+2){
    p=alpha*pow(i*M_PI/l,2);
    sum36=sum36+((-1+M_PI*M_PI)*p/(i,i+2)*(i,i+2)*p/(i,i+2))
        *((6.0*pow(t[j],5))/p)
        +((120.0*pow(t[j],3))/(p,p))
        -((360.0*pow(t[j],2))/(p,p))
        +((720.0*t[j]/pow(p,5))-(720.0/pow(p,6)));
}

sum37=0.0;
for(i=2;i<=n;i=i+2){
    p=alpha*pow(i*M_PI/l,2);
    sum37=sum37+((-1+M_PI*M_PI)*p/(i,i+2)*(i,i+2)*p/(i,i+2))
        +((120.0*pow(t[j],3))/(p,p))
        -((360.0*pow(t[j],2))/(p,p))
        +((720.0*t[j]/pow(p,5))-(720.0/pow(p,6)))*(1.0/(i,i));
}

sum38=0.0;
for(i=1;i<=n;i=i+2){
    p=alpha*pow(i*M_PI/l,2);
    sum38=sum38+((-1+M_PI*M_PI)*p/(i,i+2)*(i,i+2)*p/(i,i+2))
        +((120.0*pow(t[j],3))/(p,p))
        -((360.0*pow(t[j],2))/(p,p))
        +((720.0*t[j]/pow(p,5))-(720.0/p^6)) *(1.0/(i,i));
}

sum39=0.0;
for(i=1;i<=n;i=i+2){
    p=alpha*pow(i*M_PI/l,2);
    sum39=sum39+((-1+M_PI*M_PI)*p/(i,i+2)*(i,i+2)*p/(i,i+2))
        +((120.0*pow(t[j],3))/(p,p))
        -((360.0*pow(t[j],2))/(p,p))
        +((720.0*t[j]/pow(p,5))-(720.0/p^6)) *(1.0/(i,i));
}

sum40=0.0;
for(i=2;i<=n;i=i+2){
    p=alpha*pow(i*M_PI/l,2);
    sum40=sum40+((-1+M_PI*M_PI)*p/(i,i+2)*(i,i+2)*p/(i,i+2))
        +((120.0*pow(t[j],3))/(p,p))
        -((360.0*pow(t[j],2))/(p,p))
        +((720.0*t[j]/pow(p,5))-(720.0/p^6)) *(1.0/(i,i));
}

c[6][j]=((alpha*pow(t[j],7))/(7.0*k+1))
    +((1*pow(t[j],6))/(12.0*k))
    -((24.0*alpha/(k*l*M_PI*M_PI))*sum35)
    -((2.0*1)/(k*pow(M_PI,4)))*sum36+((M_PI*M_PI*sum37))
    +((24.0*alpha)/(k*l*M_PI*M_PI))*sum38
    +((2.0*1)/(k*pow(M_PI,4)))*(sum39+((M_PI*M_PI*sum40));
}
for(i=0;i<=nc;i++){
    for(j=0;j<=nc;j++){
        if(i==j){
            a[i][j]=0.0;
            for(kk=0;kk<=m;kk++){
                a[i][j]+=c[i][kk]*c[i][kk];
            }
        }
        else{
            a[i][j]=0.0;
            for(kk=0;kk<=m;kk++){
                a[i][j]+=c[i][kk]*c[j][kk];
            }
        }
    }
}

for(i=0;i<=nc;i++){
    for(kk=0;kk<=m;kk++){
        b[i]+=(Texp[kk]-Texp[0])*c[i][kk];
    }
}

printf("\n\n");
for(j=0;j<=nc;j++){
    for(i=0;i<=nc;i++){
        printf("%lf\t",a[i][j]);
    }
    printf("\n");
}

for(j=0;j<=nc;j++){
    printf("%lf\n",b[j]);
}

guass();

/* Printing analytical temperature */
printf("\n\n");
for(i=0;i<=m;i++){
    temperature=Texp[0]+(b[0]*c[0][i])+(b[1]*c[1][i])
    +(b[2]*c[2][i])+(b[3]*c[3][i])
    +(b[4]*c[4][i])+(b[5]*c[5][i])
    +(b[6]*c[6][i]);
    printf("%lf\t%lf\n",t[i],temperature);
}

/* Gauss Elimination Method */
void
guass()
{
int i, j, k, p, max;
double add, large, temp, ramp;

for (i = 0; i < nc; i++) {
    max = i;
    large = a[i][i];
    for (p = 0; p <= nc; p++) {
        if (a[p + 1][i] > large) {
            max = p + 1;
            large = a[p + 1][i];
        } else {
            max = max;
            large = large;
        }
    }
    ramp = b[i];
    b[i] = b[max];
    b[max] = ramp;
    for (j = 0; j <= nc; j++) {
        temp = a[i][j];
        a[i][j] = a[max][j];
        a[max][j] = temp;
    }
}

/* RIGHT SIDE MATRIX AFTER PIVOTING */

for (k = 0; k <= nc; k++) {
    for (i = k + 1; i <= nc; i++) {
        a[i][k] = a[i][k] / a[k][k];
        for (j = k + 1; j <= nc; j++)
            a[i][j] = a[i][j] - a[k][j] * a[i][k];
        b[i] = b[i] - a[i][k] * b[k];
    }
}

b[nc] = b[nc] / a[nc][nc];
for (i = nc - 1; i >= 0; i--) {
    add = b[i];
    for (k = i + 1; k <= nc; k++)
        add = add - b[k] * a[i][k];
    b[i] = add / a[i][i];
}

printf("**** Coefficients *****\n");
for(j=0;j<=nc;j++){
    printf("Coefficient[%d] = %lf\n",j,b[j]);
}
}