Louisiana State University
**LSU Digital Commons**

LSU Master's Theses | Graduate School

2002

# A model of heat and mass transfer in an idealized micro heat pipe

Jin Zhang
*Louisiana State University and Agricultural and Mechanical College*, jzhang1@lsu.edu

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_theses

Part of the Mechanical Engineering Commons

## Recommended Citation

Zhang, Jin, "A model of heat and mass transfer in an idealized micro heat pipe" (2002). *LSU Master's Theses*. 2677.
https://digitalcommons.lsu.edu/gradschool_theses/2677

This Thesis is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Master's Theses by an authorized graduate school editor of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.

# A MODEL OF HEAT AND MASS TRANSFER IN AN INDEALIZED MICRO HEAT PIPE

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College

in partial fulfillment of the
requirements for the degree of
Master of Science in Mechanical Engineering

in

The Department of Mechanical Engineering

by
Jin Zhang
B.S., Shanghai Jiao Tong University, 1997
December, 2002

# ACKNOWLEDGEMENTS

First of all, the author would like to thank her major professor, Dr. Harris Wong for his constant encouragement, insightful suggestions, and devoted guidance and help throughout the research and preparation of this thesis.

The author also likes to thank Dr. E. Nikitopoulos and Dr. T. Charalampopoulos for their time and effort to serve on her examination committee.

The author appreciates her colleagues in lab: T. Xin, D. Min, W. Kan, and Q. Wu for their help and a great time they spent together.

# TABLE OF CONTENTS

**ACKNOWLEDGMENTS** .......... ii

**ABSTRACT** .......... v

**CHAPTER 1. INTRODUCTION** .......... 1

**CHAPTER 2. EVAPORATION AND CONDENSATION** .......... 8
- 2.1. Introduction .......... 8
- 2.2. Formulation .......... 9
- 2.3. Scaling Analysis and Dimensionless Equations .......... 12
- 2.4. Asymptotic Expansions .......... 14
- 2.5. Temperature Field .......... 15
- 2.6. Velocity Field .......... 19
- 2.7. Evaporation Rate .......... 30
- 2.8. Discussion .......... 32
- 2.9. Conclusions .......... 34

**CHAPTER 3. FLUID FLOW AND HEAT TRANSFER** .......... 35
- 3.1. Introduction .......... 35
- 3.2. Formulation .......... 36
- 3.3. Dimensionless Equations .......... 40
- 3.4. Axial Temperature .......... 42
- 3.5. Axial Pressure .......... 46
- 3.6. Liquid-vapor Interfacial Curvature .......... 46
- 3.7. Effective Thermal Conductivity .......... 50
- 3.8. Discussion .......... 50
- 3.9. Conclusions .......... 53

**CHAPTER 4. CONCLUSIONS** .......... 54

**REFERENCES** .......... 56

**APPENDIX A. TOTAL EVAPORATION RATE AT THE INTERFACE** .......... 58

**APPENDIX B. PROGRAMS** .......... 60
- B.1. Basic Operations .......... 60
- B.2. Program for Computing Interfacial Temperature .......... 65
- B.3. Program for Computing the Point Force Strength and the Distance between it and the Wall .......... 67
- B.4. Program for Computing Boundary Condition on $\eta = \eta_\infty$ in Marangoni Flow .......... 70

B.5. Program for Computing Boundary Condition on $\xi = \xi_\infty$ in Marangoni Flow.....72
B.6. Program for Computing Stream Functions of Marangoni Flow.....74
B.7. Program for Computing Boundary Condition at the Liquid-vapor Interface in Sink Flow.....96
B.8. Program for Computing Boundary Condition on $\eta = \eta_\infty$ in Sink Flow.....98
B.9. Program for Computing Boundary Condition on $\xi = \xi_\infty$ in Sink Flow.....101
B.10. Program for Computing Stream Functions of Sink Flow.....104

**VITA**.....126

# ABSTRACT

Micro heat pipes have been used to cool computer chips, but their heat transfer coefficients are low compared with that of conventional heat pipes. To understand this deficiency, an idealized micro heat pipe is proposed that has simplified interfacial geometry but retains the essential physics. It is a long and narrow cavity of rectangular cross section with the bottom made of a wetting material and the top a non-wetting material. A wetting liquid fills the bottom half of the cavity, and its vapor fills the rest. This configuration ensures that the liquid-vapor interface is pinned at the contact line. Since the pipe is long, the evaporation motion at a cross-sectional plane can be taken as two-dimensional. This two-dimensional motion is governed by a Nusselt number Nu and a Marangoni number M, and is solved in the limits of Nu and M tending to infinity. It is found that evaporation occurs mainly near the contact line in a small region of size $Nu^{-1}W$, where W is the half width of the pipe. The nondimensional evaporation rate is of order $Nu^{-1} \ln Nu$. This is used to analyze unidirectional fluid flow and heat transfer along the pipe. Analytic solutions are found for the axial temperature and vapor pressure. The solutions capture for the first time the three distinct regions (evaporation, adiabatic, and condensation) commonly observed in heat pipes. It also explains why the regions do not appear in micro heat pipes. Effective thermal conductivity is studied and improvements are suggested.

# CHAPTER 1. INTRODUCTION

A conventional heat pipe is a closed duct filled with a liquid and its vapor. Figure 1 shows the structure of a conventional heat pipe. The duct is usually circular in cross section with the inner wall covered by a layer of porous material [1]. The liquid fills the porous lining whereas the vapor fills the rest. As one end of a heat pipe is heated, the liquid evaporates and increases the vapor pressure. The higher pressure drives the vapor to the colder end, where the vapor condenses and releases the latent heat. The condensed liquid then flows along the porous lining back to the hot end to complete the cycle. Since heat is transferred by evaporation, convection, and condensation, heat pipes are very efficient in heat transfer. The effective thermal conductivity can be 30 times that of copper [1]. As a result, heat pipes have been employed in a variety of applications, such as in the thermal control of the Alaska pipeline, nuclear reactor cores, and the leading edge of hypersonic aircraft [1, 2, 3, 4, 5].

In 1984, Cotter introduced the concept of micro heat pipes. It has been developed to control the temperature of micro electronic devices and components [6]. A micro heat pipe is a long polygonal (e.g. triangular or rectangular) capillary filled with a liquid and a long vapor bubble. The length of the bubble is comparable to that of the capillary. Figure 2 shows a sketch of a vapor bubble in a square capillary under uniform temperature [7]. A cross section reveals that the corners are covered by liquid menisci. These liquid-filled channels allow liquid to flow from the cold to the hot end, functioning as the porous structure in a conventional heat pipe. Figure 3 shows the structure of a micro heat pipe.

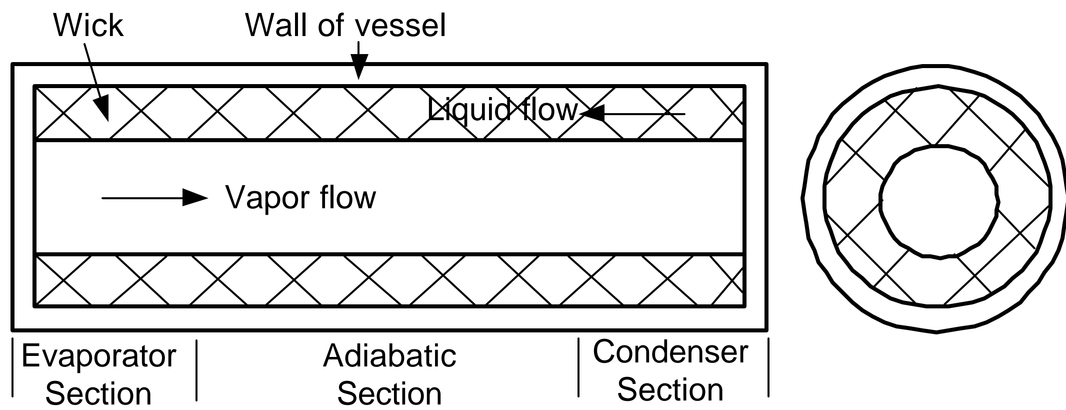


Figure 1. Schematic of a conventional heat pipe showing the principle of operation and circulation of the working fluid.

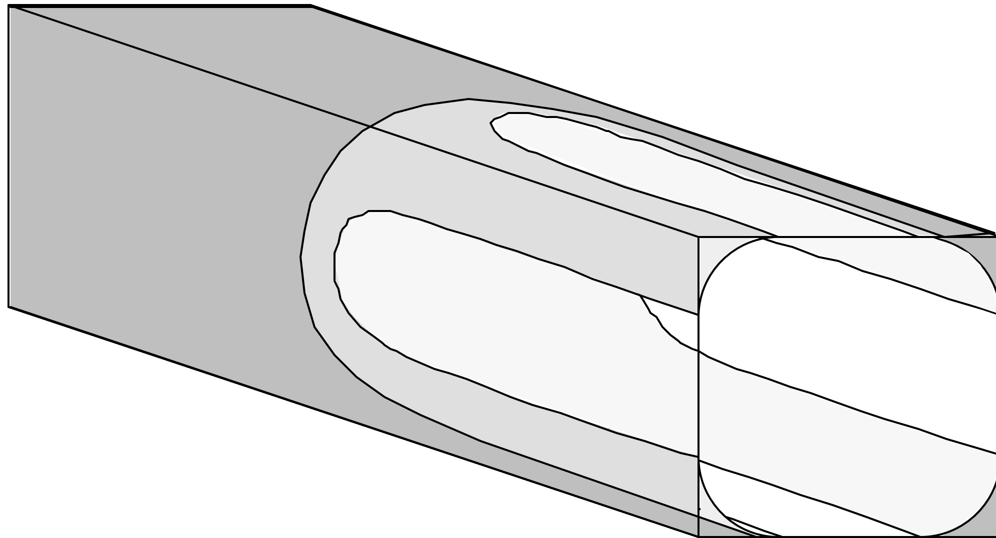

Figure 2. Half of a square micro heat pipe under uniform temperature. The liquid-filled corners take the role of conventional wick structure.

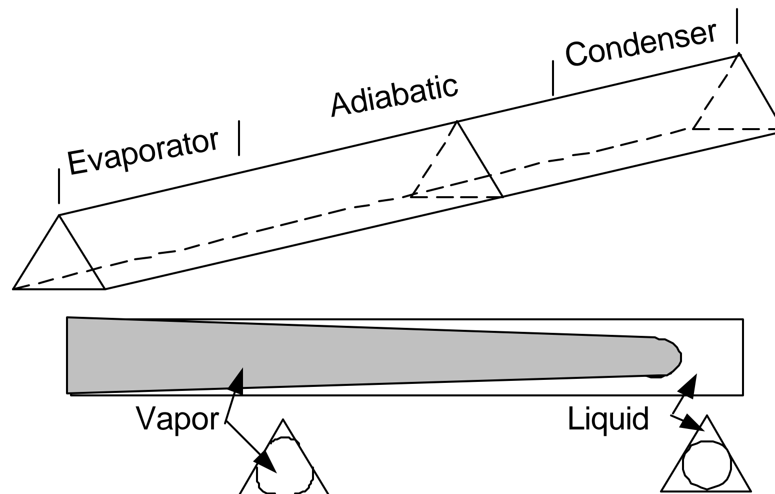


Figure 3. Schematic of a micro heat pipe.

corner meniscus recedes into the corner. The higher vapor pressure at the hot end again drives the vapor to the colder end. The vapor condenses and releases the latent heat. The condensed liquid then flows along the corner channels back to the hot end to complete the cycle. The width of a micro heat pipe can be as small as 100 μm. As a result, micro heat pipes are especially suited for thermal control of micro electronic devices. In fact, they can be fabricated as part of the circuitry [8, 9]. Micro heat pipes have been used in the dissipation of heat from the leading edge of hypersonic aircraft, heat removal from laser diodes, thermal control of photovoltaic cells, and the nonsurgical treatment of cancerous tissue through either hyper- or hypothermia [6]. Despite the wide application, there is a lack of rigorous analysis of the heat and mass transfer in micro heat pipes due to the complexity of the bubble geometry.

The effective thermal conductivity of micro heat pipes is far below that of conventional heat pipes. A typical value for the effective thermal conductivity of a conventional heat pipe is 13200 W/m°C [1], whereas that of a micro heat pipe is 300 W/m°C [8]. The effective thermal conductivity of a micro heat pipe with and without a liquid differs by at most 50% [8, 9]. Thus, it seems that the heat transfer capability of micro heat pipes has not been fully developed.

To understand the basic heat transfer mechanisms of micro heat pipes, it is better to analyze a system without the complicated bubble geometry, but with all the essential physics retained. There are several basic elements of micro heat pipes that are needed for heat transfer. The corner menisci in a micro heat pipe serve as a channel for liquid flow. Thus, the new system must also contain a liquid flow channel. The liquid-vapor interface at the hot and cold ends changes its curvature due to evaporation and condensation. Thus, the new system must also include a liquid-vapor interface that separates the vapor and liquid flow channels,

and the interface must be able to deform under evaporation and condensation. The deformation must be able to generate a capillary pressure difference that drives a liquid flow from the cold end to the hot end. Finally, the contact line region in a micro heat pipe has been shown to play a crucial role in evaporation and condensation and must be retained in the new system. Below, an idealized micro heat pipe is proposed which captures all these essential elements.

Figure 4 shows the idealized micro heat pipe. It is a rectangular heat pipe, the top portion of which is made of a non-wetting material, while the bottom portion is a wetting material. The wetting half is filled to the rim by a wetting liquid, and the upper half is filled by its vapor. This configuration ensures that the contact line of the liquid-vapor interface is pinned at the interception between the wetting and non-wetting materials. Pinning of the interface generates a capillary pressure gradient that drives a liquid flow from the cold to the hot end. When this micro heat pipe is driven at a small temperature difference, the interface should be approximately flat, allowing the analysis to be greatly simplified. The goal is to identify key parameters that govern heat transfer.

Chapter 2 studies the evaporation and condensation mechanisms. Since micro heat pipes are usually slender, flow and temperature fields vary slowly in the axial direction. Thus, locally at each point along the pipe, the liquid flow and heat transfer are taken as two dimensional. The motion within a cross-sectional plane is studied. The governing equations are listed in Section 2.2. A scaling analysis in Section 2.3 identifies two velocity scales, one from Marangoni flow and one from evaporation or condensation. The equations are made dimensionless and two parameters emerge: a Nusselt number Nu ($\gg 1$) and a Marangoni number M ($\gg 1$). The dependent variables, such as velocity, temperature, and pressure, are expanded in $Nu^{-1}$ and $M^{-1}$ in Section 2.4, and solved in the limit $Nu^{-1} \to 0$ and $M^{-1}$

$\rightarrow 0$. An analytic representation of the temperature field is presented in Section 2.5, while a numerical solution of the velocity field appears in Section 2.6. The evaporation or condensation rate Q is then found in Section 2.7. The leading-order term of Q is used in Chapter 3 to calculate the vapor and liquid flow and the temperature distribution along the pipe. The results are discussed in Section 2.8, and Chapter 2 is concluded in Section 2.9.

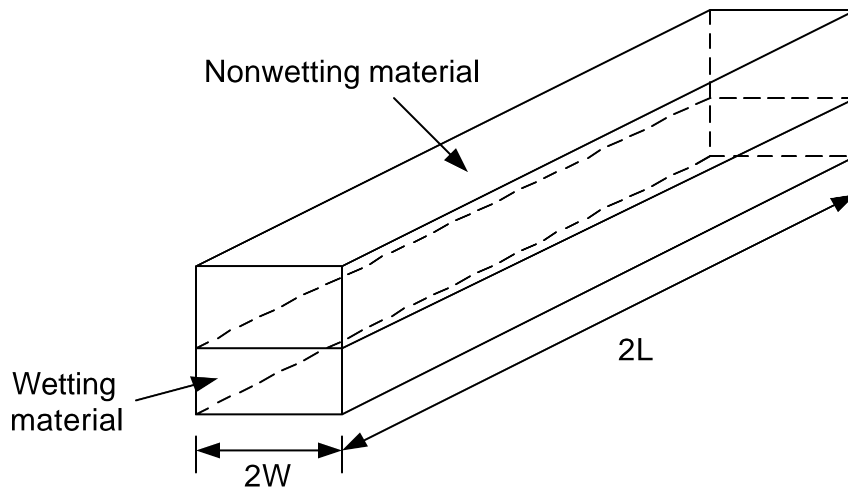


Figure 4. Schematic of an idealized micro heat pipe with 2W width and 2L length. The top half is made of nonwetting material and the lower half wetting material.

Chapter 3 studies heat transfer and fluid flow along the pipe using the evaporation rate derived in Chapter 2. It is assumed that the liquid has the same temperature as the wall. Since the pipe is slender, the flow along the pipe is taken

as unidirectional, and the heat conduction is taken to be one dimensional. Because the Reynolds and Peclect numbers are small, convective heat transfer is negligible, and the energy is balanced in the way that the wall-liquid system loses heat by evaporation. Section 3.2 lists the governing equations for the axial liquid temperature and the vapor pressure, which are coupled through the evaporation rate. Section 3.3 non-dimensionalizes the equations. Through the evaporation rate, the temperature is decoupled from the pressure and is solved first in Section 3.4. Vapor and liquid pressures are solved in Section 3.5. This gives the liquid-vapor interfacial curvature in Section 3.6. Factors that may affect the effective thermal conductivity are surveyed in Section 3.7. Ways to improve the effective thermal conductivity are thereafter suggested. Chapter 3 is concluded in Section 3.8.

This work is concluded in Chapter 4.

# CHAPTER 2. EVAPORATION AND CONDENSATION

## 2.1. Introduction

Evaporation and condensation is one of the major phenomenon studied in micro heat pipe operations. Heat applied to one end of the heat pipe vaporizes the liquid in that region and forces it to move to the cooler end where it condenses and gives up the latent heat of vaporization. This evaporation and condensation process deforms the liquid-vapor interface and thus generates the capillary pressure difference between the evaporator and the condenser regions. This capillary pressure gradient then drives the working fluid back from the condenser end to the evaporator end. Therefore, increasing evaporation will promote the efficiency of micro heat pipes.

Understanding the evaporation mechanism is therefore crucial. Evaporative mass constitutive equation is the key point for the formulation of the problem. This work uses a constitutive equation that relates the evaporation rate to the deviation of vapor pressure from its equilibrium state. This indicates that evaporation and condensation are two symmetric problems. Therefore, in this work, only evaporation is studied.

Based on the idealized micro heat pipe presented in Chapter 1, the liquid motion in the pipe is studied. Because the pipe is long, flow along the pipe varies slowly, which can be well approximated as two-dimensional at each cross-sectional plane. Using asymptotic analysis, an inner structure and an outer structure are captured. Not only does this solve the non-integrable singularity in temperature appearing at the contact line region, which is a usual difficulty in evaporation problems, it also illustrates that the inner region contributes the most

to the evaporation at the interface. Thus, the efficiency of the pipe can be improved by increasing the length of the contact line.

## 2.2. Formulation

The evaporation or condensation region is shown in Figure 5. The region is bounded by two vertical walls separated by a distance 2W. The bottom half of the region is filled with a liquid and the top half by its vapor. The walls are made of a wetting material at the bottom, and a non-wetting material at the top, and both are maintained at temperature $T_\infty$. Initially, the system is in equilibrium, so that the liquid and vapor are at temperature $T_\infty$, and the vapor is at the equilibrium pressure $p_\infty$. There is no motion in either the vapor or liquid, and the interface is taken to be flat. The vapor pressure is subsequently reduced to $p_g < p_\infty$ and maintained at that level. This drop in vapor pressure induces evaporation at the interface. Since we are interested in the steady state performance of micro heat pipes, only time-independent equations are needed for liquid flow and heat conduction:

$$\nabla^* p^* = \mu \nabla^{*2} \mathbf{u}^* \tag{2.2.1a}$$

$$\nabla^* \cdot \mathbf{u}^* = 0 \tag{2.2.1b}$$

$$\nabla^{*2} T^* = 0 , \tag{2.2.1c}$$

where $p^*$ is pressure, $\mu$ is viscosity, $\mathbf{u}^* = u^*\mathbf{i} + v^*\mathbf{j}$ is velocity, and $T^*$ is temperature. A Cartesian coordinate system $(x^*, y^*)$ is defined as shown in Figure 5, $\nabla^* = \partial/\partial x^*\mathbf{i} + \partial/\partial y^*\mathbf{j}$ is the two-dimensional spatial gradient operator. Liquid is supplied at infinity to keep the interface stationary.

Convective heat transfer and inertia are negligible, as detailed in the Discussion. The vapor exerts insignificant shear stress on the liquid, and has much lower thermal conductivity than that of the liquid. Thus, heat and mass transfer in

the vapor has negligible effects. The pressure $p_g$ of the vapor is taken to be constant and is the only variable considered in the vapor region.

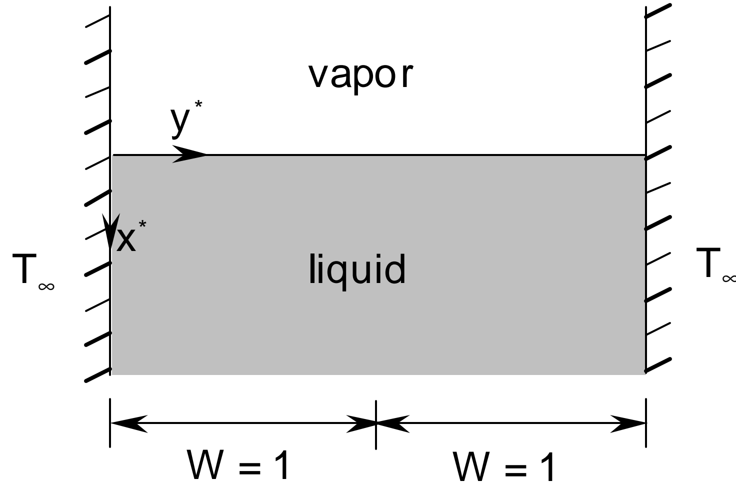


Figure 5. A cross section of the idealized micro heat pipe. Cartesian coordinates ($x^*$, $y^*$) are defined with the origin at the contact line on the left wall.

On the wall at $y^* = 0$, we assume the no-slip boundary condition

$$u^* = 0 \tag{2.2.2a}$$

$$v^* = 0 , \tag{2.2.2b}$$

and impose a prescribed constant temperature,

$$T^* = T_\infty . \tag{2.2.2c}$$

The interface is pinned at the walls between the wetting and non-wetting parts, and liquid is replenished at the bottom to balance the loss due to evaporation. The interface is taken as flat because surface tension dominates in micro heat pipes owing to the small size. Thus, the interface is located at $x^* = 0$ (Figure 5). At the interface, the vertical velocity is nonzero due to evaporation:

$$m = -\ \rho\, u^* \ , \tag{2.2.3}$$

where $\rho$ is the liquid density and m is the local evaporative mass flux (m is positive for evaporation and negative for condensation). The horizontal shear stress balances a surface tension gradient:

$$\mu \frac{\partial v^*}{\partial x^*} = -\ \frac{d\sigma}{dy^*} = -\ \frac{d\sigma}{dT^*}\frac{dT^*}{dy^*}\ , \tag{2.2.4}$$

where $\sigma$ is the surface tension, which varies along the interface because the temperature varies. In this work, $d\sigma/dT^*$ is assumed constant owing to the small variation in temperature. The normal stress balance does not enter at this stage since the interface is taken as flat.

The local evaporative mass flux m is related to the surface temperature $T^*$ by

$$m = \frac{c\ \rho_\infty h_{fg}\,(T^* - T_c)}{T_\infty}\ , \tag{2.2.5a}$$

$$T_c = T_\infty - \frac{T_\infty\,(p_\infty - p_g)}{\rho_\infty\, h_{fg}}\ , \tag{2.2.5b}$$

where $\rho_\infty$ is the equilibrium vapor density, $h_{fg}$ is the latent heat, $p_\infty$ is the equilibrium vapor pressure at temperature $T_\infty$, and c is defined in (2.2.6). This equation is derived by first assuming that the evaporative mass flux is proportional to the vapor-pressure deviation from the equilibrium value:

$$m = c\ \ (p_e - p_g)\ , \tag{2.2.6}$$

where $p_e$ is the equilibrium vapor pressure at temperature $T^*$, and $c = (2\pi R\, T_\infty)^{-1/2}$ with R being the universal gas constant per unit mass of the vapor as determined by the kinetic theory [10, 11, 12, 13]. Since $p_e$ is only a function of temperature and is close to $p_\infty$, by Taylor's expansion,

$$p_e = p_\infty + \frac{\rho_\infty\, h_{fg}}{T_\infty}\left(T^* - T_\infty\right), \tag{2.2.7}$$

where the gradient $dp_e / dT^* = \rho_\infty h_{fg} / T_\infty$ comes from the Clausius-Clapeyron relation, assuming $|T^* - T_\infty| << T_\infty$. Substitution of $p_e$ into (2.2.6) yields (2.2.5).

Energy balance at the interface $x^* = 0$ gives

$$m\ h_{fg} = k\ \frac{\partial T^*}{\partial x^*} . \tag{2.2.8}$$

Here, k is the liquid thermal conductivity.

Due to symmetry, only half of the domain needs to be analyzed. At the symmetry plane $y^* = W$,

$$\frac{\partial u^*}{\partial y^*} = 0 , \tag{2.2.9a}$$

$$v^* = 0 , \tag{2.2.9b}$$

$$\frac{\partial T^*}{\partial y^*} = 0 . \tag{2.2.9c}$$

The evaporation rate in half of the domain is

$$Q^* = \int_0^W u^* \, dy^* . \tag{2.2.10}$$

The main purpose in this chapter is to find $Q^*$.

Far from the pinned interface, the liquid flow is nonzero due to evaporation and the temperature approaches the wall temperature. Thus, as $x^* \to \infty$

$$\int_0^W u^* \, dy^* \to Q^* , \tag{2.2.11a}$$

$$v^* \to 0 , \tag{2.2.11b}$$

$$T^* \to T_\infty . \tag{2.2.11c}$$

## 2.3. Scaling Analysis and Dimensionless Equations

There are two velocity scales in this problem, one induced by evaporation, and the other by Marangoni flow. The evaporation velocity scale comes from (2.2.3) and (2.2.5):

$$V_E = \frac{c\ \rho_\infty h_{fg} (T_\infty - T_c)}{\rho\ T_\infty} . \tag{2.3.1a}$$

Boundary condition (2.2.4) gives the Marangoni velocity scale:

$$V_M = -\ \frac{1}{\mu} \frac{d\sigma}{dT} \left( T_\infty - T_c \right) . \tag{2.3.1b}$$

The ratio of these two velocities is the Marangoni number $M \equiv V_M / V_E$, which for water at 20 °C gives $M \approx 10^3$. Thus, the Marangoni flow dominates, and so $V_M$ is selected as the velocity scale.

A set of dimensionless variables is defined:

$$x = \frac{x^*}{W},\ y = \frac{y^*}{W},\ \mathbf{u} = \frac{\mathbf{u}^*}{V_M},\ p = \frac{p^*\ W}{\mu\ V_M},\ T = \frac{T^* - T_c}{T_\infty - T_c},\ Q = \frac{Q^*}{V_M\ W} . \tag{2.3.2}$$

The governing equations become

$$\tilde{\mathbf{N}} p = \tilde{\mathbf{N}}^2 \mathbf{u}\ , \tag{2.3.3a}$$

$$\tilde{\mathbf{N}} \cdot \mathbf{u} = 0\ , \tag{2.3.3b}$$

$$\tilde{\mathbf{N}}^2 T = 0\ , \tag{2.3.3c}$$

where $\tilde{\mathbf{N}} = \mathbf{i}\,\partial / \partial x + \mathbf{j}\,\partial / \partial y$. At the wall $y = 0$,

$$u = 0\ , \tag{2.3.4a}$$

$$v = 0\ , \tag{2.3.4b}$$

$$T = 1\ . \tag{2.3.4c}$$

At the interface $x = 0$, the velocity components obey

$$u = -\ \frac{1}{M} T\ , \tag{2.3.5a}$$

$$\frac{\partial v}{\partial x} = -\ \frac{\partial T}{\partial y} . \tag{2.3.5b}$$

The constitutive equation (2.2.5) combines with the energy balance (2.2.8) to give

$$Nu\ T = \frac{dT}{dx}\ , \tag{2.3.5c}$$

$$Nu = \frac{h\ W}{k}\ , \tag{2.3.5d}$$

$$h = \frac{c\ \rho_\infty\ h_{fg}^2}{T_\infty}, \tag{2.3.5e}$$

where h is a heat transfer coefficient defined according to Newton's cooling law from (2.2.5a) and (2.2.8). The Nusselt number Nu reflects the ratio of the evaporative heat flux at the interface to the conductive heat flux in the bulk. It defines a length scale ($Nu^{-1}W$) over which the conductive heat flux becomes comparable to the evaporative heat flux under the same temperature difference. For water at 20 °C and W = 1 mm, Nu = 1000.

At the symmetric plane y = 1,

$$\frac{\partial u}{\partial y} = 0, \tag{2.3.6a}$$

$$v = 0, \tag{2.3.6b}$$

$$\frac{\partial T}{\partial y} = 0. \tag{2.3.6c}$$

Far from the liquid interface, as $x \to \infty$,

$$\int_0^1 u\, dy \to Q, \tag{2.3.7a}$$

$$v \to 0, \tag{2.3.7b}$$

$$T \to 1. \tag{2.3.7c}$$

## 2.4. Asymptotic Expansions

Two parameters appear in the above problem: Nu (>> 1) and M (>> 1). The parameter Nu appears only in the heat-transfer problem, and M only in the liquid-flow problem. Thus, the dependant variables are expanded in the limit $Nu^{-1} \to 0$ and $M^{-1} \to 0$ as

$$\mathbf{u} = \mathbf{u}_0 + M^{-1}\ \mathbf{u}_1 + \ldots \tag{2.4.1a}$$

$$p = p_0 + M^{-1}\ p_1 + \ldots \tag{2.4.1b}$$

$$T = T_0 + Nu^{-1}\ T_1 + \ldots \tag{2.4.1c}$$

At the wall y = 0, the leading-order temperature is

$$T_0 = 1, \tag{2.4.2}$$

and at the liquid-vapor interface x = 0,

$$T_0 = 0\ . \tag{2.4.3}$$

This jump in temperature at the contact line appears often in evaporation problems and leads to unbounded temperature gradient. In the present problem, this jump results from the disappearance of the derivative in (2.3.5c) because it is multiplied by $Nu^{-1}$. This hints at the existence of an inner region. An inner problem is formulated next which shows that the temperature T in the inner region decays smoothly along the interface from T = 1 at the wall to T = 0 away from the wall. The existence of the inner problem results from the constitutive equation (2.2.5) for the evaporative mass flux, and from the constant temperature condition at the wall.

The temperature field is independent of the velocity field, but the velocity field depends on the interface temperature. Thus the heat-conduction problem is solved first in Section 2.5, followed by the liquid-flow problem in Section 2.6.

## 2.5. Temperature Field

### 2.5.1. Inner Temperature $T_i$

A set of inner variables is defined as

$$X = Nu\, x \qquad Y = Nu\, y\ . \tag{2.5.1a,b}$$

The inner temperature field obeys

$$\tilde{\mathbf{N}}^2 T_i = 0\ . \tag{2.5.2a}$$

At the wall Y = 0,

$$T_i = 1. \tag{2.5.2b}$$

At the liquid-vapor interface X = 0,

$$T_i = \frac{\partial T_i}{\partial X} . \qquad (2.5.2c)$$

As $Y \to \infty$,

$$T_i \to 0 . \qquad (2.5.2d)$$

Since the parameter Nu is scaled out of the problem, the solution holds for all orders of Nu.

The inner problem describes the local behavior near the contact line. It is intrinsic to the evaporation problem, and is independent of the outer problem. To solve (2.5.2), a new dependent variable is defined: $G = \partial T_i / \partial X - T_i$. The governing equation remains the same $\nabla^2 G = 0$, but the boundary condition at the liquid-vapor interface is reduced to $G = 0$, and that at the wall $Y = 0$, $G = -1$. This leads to a simple solution: $G = 2/\pi \tan^{-1}(Y/X) - 1$. Thus, the inner temperature obeys $\partial T_i / \partial X - T_i = (2/\pi) \tan^{-1}(Y/X) - 1$, the solution of which is

$$T_i = 1 - \frac{2}{\pi} \tan^{-1}\left(\frac{Y}{X}\right) + \frac{2}{\pi} e^X Y \int_X^\infty \frac{e^{-\lambda}}{\lambda^2 + Y^2} d\lambda . \qquad (2.5.3)$$

The Simpson method is used to evaluate the integral, with the upper limit set to be 20. The temperature distribution along the liquid-vapor interface is plotted in Figure 6. It indicates that the temperature varies smoothly from 1 at the wall to 0 away from the wall. This solution agrees with that of Morris [14] who considered a more general problem, in which the isothermal condition is not imposed at the solid-liquid interface, but at a distance away from it. Also, in his problem, the liquid-vapor interface needs not to be perpendicular to the wall. This additional complexity only changes the total evaporative rate by a factor that is less than or equal to one.

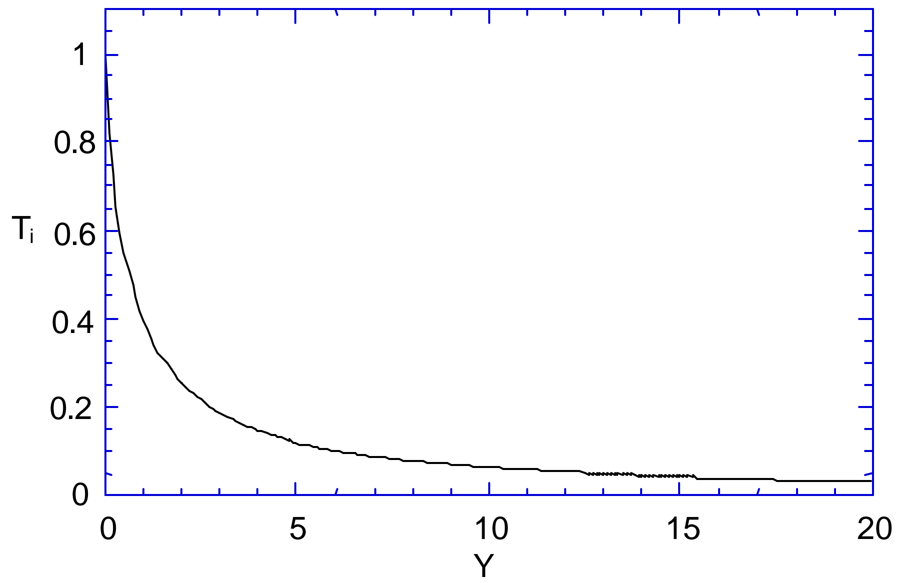


Figure 6. Dimensionless temperature distribution along the liquid-vapor interface in the inner region. The temperature obeys (2.5.3).

## 2.5.2. Zero-order Outer Temperature $T_0$

According to the expansion in Section 2.4, the zero order outer temperature satisfies

$$\tilde{\mathbf{N}}^2 T_0 = 0 . \qquad (2.5.4)$$

At the wall y = 0,

$$T_0 = 1, \qquad (2.5.5)$$

which at the liquid-vapor interface x = 0,

$$T_0 = 0 . \qquad (2.5.6)$$

Finally, on the symmetric plane y = 1,

$$\frac{\partial T_0}{\partial y} = 0. \qquad (2.5.7)$$

Conformal mapping is used to solve $T_0$. First, set $a = \pi\, y / 2$, $b = \pi\, x / 2$, and $G = T - 1$. Then, transform the variables [15]:

$$w_1 = \sin\left(a + i\, b\right) = u + i\, v , \qquad (2.5.8a)$$

$$w_2 = \log \frac{u + i\, v - 1}{u + i\, v + 1} = \bar{u} + i\, \bar{v} , \qquad (2.5.8b)$$

where

$$u = \sin y \cosh x , \qquad (2.5.8c)$$

$$v = \cos y \sinh x , \qquad (2.5.8d)$$

$$\bar{u} = \frac{1}{2} \log \frac{(u - 1)^2 + v^2}{(u + 1)^2 + v^2} , \qquad (2.5.8e)$$

$$\bar{v} = \tan^{-1} \frac{2\, v}{u^2 + v^2 - 1} . \qquad (2.5.8f)$$

Thus, a simple solution of G is found

$$G = - \frac{\bar{v}}{\pi} . \qquad (2.5.8g)$$

This gives the solution of the zero-order outer temperature:

$$T_0 = 1 - \frac{2}{\pi} \tan^{-1} \left[ \frac{\sin\left(\pi y/2\right)}{\sinh\left(\pi x/2\right)} \right] . \qquad (2.5.9)$$

### 2.5.3. First-order Outer Temperature $T_1$

From boundary condition (2.3.5a), the interfacial temperature determines the normal velocity at the interface. Thus, the temperature distribution along the interface is needed for calculation of the evaporation rate. The outer solution (2.5.9) shows that $T_0 = 0$ at the interface $x = 0$, which means that the zero-order outer solution does not contribute to the evaporation. Thus, a higher order outer solution is needed at the interface. Substitution of the temperature expansion in (2.4.1c) into (2.3.5c) yields the first order temperature at the interface $x = 0$,

$$T_1 = \frac{\partial T_0}{\partial x} = \csc \left(\frac{\pi y}{2}\right) . \qquad (2.5.10)$$

## 2.6. Velocity Field

The velocity expansions $\mathbf{u}_0$ and $\mathbf{u}_1$ are zero in the outer region defined by $Nu^{-1}$ in Section 2.5. They are leading and first order terms in the expansion in $M^{-1}$, as listed in (2.4.1a). They are also the leading-order term in the expansion in $Nu^{-1}$. However, to leading order in $Nu^{-1}$, all velocity boundary conditions are zero in the outer region. This is because in the outer region defined by $Nu^{-1}$, $T = 0$ at the interface to the leading order in $Nu^{-1}$. Also, in (2.3.7a), the total evaporation rate $Q \sim Nu^{-1} \ln Nu$ as shown in Section 2.7. Hence, $\mathbf{u}_0$ and $\mathbf{u}_1$ are zero in the outer region, and only their solutions in the inner region are studied below.

### 2.6.1. Zero-order velocity $\mathbf{u}_0$

In component form, $\mathbf{u}_0 = u_0\mathbf{i} + v_0\mathbf{j}$. At the liquid-vapor interface $X = 0$,

$$u_0 = 0 , \qquad (2.6.1a)$$

$$\frac{\partial v_0}{\partial X} = - \frac{\partial T_i}{\partial Y} . \qquad (2.6.1b)$$

Thus, there is zero evaporation at this order, and the flow is driven by the Marangoni stress.

A stream function is defined in terms of the inner variables:

$$u_0 = \frac{\partial \psi_0}{\partial Y}, \tag{2.6.2a}$$

$$v_0 = - \frac{\partial \psi_0}{\partial X}. \tag{2.6.2b}$$

The stream function obeys

$$\tilde{\mathbf{N}}^4 \psi_0 = 0 . \tag{2.6.3}$$

No slip at the wall Y = 0 requires

$$\frac{\partial \psi_0}{\partial Y} = 0 , \tag{2.6.4a}$$

$$\frac{\partial \psi_0}{\partial X} = 0 . \tag{2.6.4b}$$

At the liquid-vapor interface X = 0, (2.6.1) becomes

$$\frac{\partial \psi_0}{\partial Y} = 0 , \tag{2.6.4c}$$

$$\frac{\partial^2 \psi_0}{\partial X^2} = \frac{\partial T_i}{\partial Y} . \tag{2.6.4d}$$

The velocity components decay to zero far from the corner. Thus, as $X \to \infty$ or $Y \to \infty$,

$$\frac{\partial \psi_0}{\partial Y} \to 0 , \tag{2.6.4e}$$

$$\frac{\partial \psi_0}{\partial X} \to 0 . \tag{2.6.4f}$$

The Marangoni stress at the interface is the only nonzero term and it drives the flow.

Because the flow domain is unbounded, an asymptotic solution is found that simplifies the boundary conditions at the finite computational domain boundary. The shear stress at the interface is nonzero near the corner, and decays to zero far from the corner, as shown by (2.6.4d). Thus, far from the corner, the shear stress can be viewed as a point force. The stream function generated by a point force pointing towards a solid wall is [16]

$$\psi_0 = f\,X \left( \frac{2\,H\,Y}{X^2+(Y+H)^{\,2}} - \frac{1}{2} \ln \left( \frac{X^2+(Y+H)^{\,2}}{X^2+(Y-H)^{\,2}} \right) \right), \tag{2.6.5}$$

where f is the strength of the point force and H is the distance between the point force and wall. The strength f is found from the shear stress boundary condition (2.6.4d) by integrating in Y:

$$f = \frac{T(Y_\infty)-1}{2\pi}, \tag{2.6.6}$$

where $Y_\infty$ (>> 1) is the position at which the boundary conditions at infinity are applied. The distance H is determined from the first moment of the shear stress about Y = 0:

$$H = \left( \int_0^{Y_\infty} \frac{\partial v_0}{\partial X}\, dY \right)^{-1} \left( \int_0^{Y_\infty} Y \frac{\partial v_0}{\partial X}\, dY \right). \tag{2.6.7}$$

Point force strength f and the distance between it and wall H are calculated numerically by solving the interface temperature in the contracted coordinates first (program is listed in Appendix B.2.) and then using the Simpson method to solve the integral. The program is listed in Appendix B.3.

Thus, the velocity is determined by $\psi_0$ (2.6.5):

$$u = - \frac{-8\,f\;\;H^2\,X\,Y\,\left(X^2 - Y^2 + H^2\right)}{\left[X^2 + (Y+H)^2\right]^2 \left[X^2 + (Y-H)^2\right]^2}, \tag{2.6.8a}$$

$$v = -f \left[ \frac{2\,Y\,H}{X^2 + (Y+H)^2} - \frac{1}{2} \ln \frac{X^2 + (Y+H)^2}{X^2 + (Y-H)^2} \right]. \tag{2.6.8b}$$

A large domain is needed for the asymptotic conditions to be valid. Therefore, the coordinates are contracted by defining

$$\xi = \ln\,(X+1), \tag{2.6.9a}$$

$$\eta = \ln\,(Y+1). \tag{2.6.9b}$$

Thus, the governing equation becomes

$$2\, e^{-2(\xi+\eta)}\left(\frac{\partial^4\psi_0}{\partial\xi^2\,\partial\eta^2} - \frac{\partial^3\psi_0}{\partial\xi^2\partial\eta} - \frac{\partial^3\psi_0}{\partial\xi\,\partial\eta^2} + \frac{\partial^2\psi_0}{\partial\xi\,\partial\eta}\right)$$
$$+\ e^{-4\xi}\left(\frac{\partial^4\psi_0}{\partial\xi^4} - 6\,\frac{\partial^3\psi_0}{\partial\xi^3} + 11\,\frac{\partial^2\psi_0}{\partial\xi^2} - 6\,\frac{\partial\psi_0}{\partial\xi}\right) \qquad (2.6.10)$$
$$+\ e^{-4\eta}\left(\frac{\partial^4\psi_0}{\partial\eta^4} - 6\,\frac{\partial^3\psi_0}{\partial\eta^3} + 11\,\frac{\partial^2\psi_0}{\partial\eta^2} - 6\,\frac{\partial\psi_0}{\partial\eta}\right) = 0\,.$$

At the wall $\eta = 0$,

$$\frac{\partial\psi_0}{\partial\eta} = 0\,, \qquad (2.6.11a)$$
$$\frac{\partial\psi_0}{\partial\xi} = 0\,. \qquad (2.6.11b)$$

At the liquid-vapor interface $\xi = 0$

$$\frac{\partial\psi_0}{\partial\eta} = 0\,, \qquad (2.6.12a)$$
$$\frac{\partial^2\psi_0}{\partial\xi^2} - \frac{\partial\psi_0}{\partial\xi} = e^{2\xi-\eta}\,\frac{\partial T}{\partial\eta}\,. \qquad (2.6.12b)$$

At the outer boundary $\eta = \eta_\infty$,

$$\frac{\partial\psi_0}{\partial\eta} = e^{\eta}\, u_\eta\,, \qquad (2.6.13a)$$
$$\frac{\partial\psi_0}{\partial\xi} = -\, e^{\xi}\, v_\eta\,, \qquad (2.6.13b)$$

where

$$u_\eta = -\,\frac{8 f\, H^2\left(e^{\xi}-1\right)\left(e^{\eta_\infty}-1\right)\left[\left(e^{\xi}-1\right)^2 - \left(e^{\eta_\infty}-1\right)^2 + H^2\right]}{\left[\left(e^{\xi}-1\right)^2 + \left(e^{\eta_\infty}-1+H\right)^2\right]^2\left[\left(e^{\xi}-1\right)^2 + \left(e^{\eta_\infty}-1-H\right)^2\right]^2}\,, \qquad (2.6.13c)$$
$$v_\eta = -f\left[\frac{2H\left(e^{\eta_\infty}-1\right)}{\left(e^{\xi}-1\right)^2 + \left(e^{\eta_\infty}-1+H\right)^2} - \frac{1}{2}\ln\frac{\left(e^{\xi}-1\right)^2 + \left(e^{\eta_\infty}-1+H\right)^2}{\left(e^{\xi}-1\right)^2 + \left(e^{\eta_\infty}-1-H\right)^2}\right].$$

(2.6.13d)

At the outer boundary $\xi = \xi_\infty$,

$$\frac{\partial\psi_0}{\partial\eta} = e^{\eta}\, u_\xi\,, \qquad (2.6.14a)$$
$$\frac{\partial\psi_0}{\partial\xi} = -\, e^{\xi}\, v_\xi\,, \qquad (2.6.14b)$$

where

$$u_\xi = -\frac{8f\ H^2\left(e^{\xi_\infty}-1\right)\left(e^{\eta}-1\right)\left[\left(e^{\xi_\infty}-1\right)^2-\left(e^{\eta}-1\right)^2+H^2\right]}{\left[\left(e^{\xi_\infty}-1\right)^2+\left(e^{\eta}-1+H\right)^2\right]^2\left[\left(e^{\xi_\infty}-1\right)^2+\left(e^{\eta}-1-H\right)^2\right]^2}, \quad (2.6.14c)$$

$$v_\xi = -f\left[\frac{2H\left(e^{\eta}-1\right)}{\left(e^{\xi_\infty}-1\right)^2+\left(e^{\eta}-1+H\right)^2}\right]-\frac{1}{2}\ln\frac{\left(e^{\xi_\infty}-1\right)^2+\left(e^{\eta}-1+H\right)^2}{\left(e^{\xi_\infty}-1\right)^2+\left(e^{\eta}-1-H\right)^2}. \quad (2.6.14d)$$

The stream function is then solved by a finite-difference method. The discretized governing equation is

$$\begin{aligned}(A\psi_0)_{i,j} = &-(A\psi_0)_{i+1,j}-(A\psi_0)_{i-1,j}-(A\psi_0)_{i+2,j}-(A\psi_0)_{i-2,j}\\ &-(A\psi_0)_{i,j+1}-(A\psi_0)_{i,j-1}-(A\psi_0)_{i,j+2}-(A\psi_0)_{i,j-2}\\ &-(A\psi_0)_{i+1,j+1}-(A\psi_0)_{i+1,j-1}-(A\psi_0)_{i-1,j+1}-(A\psi_0)_{i-1,j-1}.\end{aligned} \quad (2.6.15)$$

Here, A is defined as

$$A_{i,j} = 1/\ h^4\left(\left(e^{-4\xi_i}+e^{-4\eta_j}\right)\left(6-22\ h^2\right)+8\ e^{-2\left(\xi_i+\eta_j\right)}\right) \quad (2.6.16a)$$

$$A_{i+1,j} = 1/\ h^4\left(e^{-4\xi_i}\left(-3h^3+11\ h^2+6h-4\right)+2\ \left(h-2\right)e^{-2\left(\xi_i+\eta_j\right)}\right) \quad (2.6.16b)$$

$$A_{i-1,j} = 1/\ h^4\left(e^{-4\xi_i}\left(3h^3+11\ h^2-6h-4\right)-2\ \left(h+2\right)e^{-2\left(\xi_i+\eta_j\right)}\right) \quad (2.6.16c)$$

$$A_{i,j+1} = 1/\ h^4\left(e^{-4\eta_j}\left(-3h^3+11\ h^2+6h-4\right)+2\ \left(h-2\right)e^{-2\left(\xi_i+\eta_j\right)}\right) \quad (2.6.16d)$$

$$A_{i,j-1} = 1/\ h^4\left(e^{-4\eta_j}\left(3h^3+11\ h^2-6h-4\right)-2\ \left(h+2\right)e^{-2\left(\xi_i+\eta_j\right)}\right) \quad (2.6.16e)$$

$$A_{i+1,j+1} = 2\ e^{-2\left(\xi_i+\eta_j\right)}\left(1-h+\ h^2/4\right)/\ h^4 \quad (2.6.16f)$$

$$A_{i-1,j+1} = 2\ e^{-2\left(\xi_i+\eta_j\right)}\left(1-\ h^2/4\right)/\ h^4 \quad (2.6.16g)$$

$$A_{i+1,j-1} = 2\ e^{-2\left(\xi_i+\eta_j\right)}\left(1-\ h^2/4\right)/\ h^4 \quad (2.6.16h)$$

$$A_{i-1,j-1} = 2\ e^{-2\left(\xi_i+\eta_j\right)}\left(1+h+\ h^2/4\right)/\ h^4 \quad (2.6.16i)$$

$$A_{i+2,j} = e^{-4\xi_i}\left(1-3h\right)/\ h^4 \quad (2.6.16j)$$

$$A_{i-2,j} = e^{-4\xi_i}\left(1+3h\right)/\ h^4 \quad (2.6.16k)$$

$$A_{i,j+2} = e^{-4\eta_j}\left(1-3h\right)/\ h^4 \quad (2.6.16l)$$

$$A_{i,j-2} = e^{-4\eta_j}\left(1+3h\right)/\ h^4. \quad (2.6.16m)$$

Here, h is the step size, $(\xi_i, \eta_j)$ represents a point on (i, j).

The boundary conditions are imposed as follows. Take $\psi_0 = 0$ at the wall. Thus, at the wall j = 0 and i = [0, n],

$$(\psi_0)_{i,0} = 0 , \tag{2.6.17a}$$

$$(\psi_0)_{i,-1} = (\psi_0)_{i,1} . \tag{2.6.17b}$$

At the liquid-vapor interface i = 0, j = [1, n]

$$(\psi_0)_{0,i} = 0 , \tag{2.6.18a}$$

$$(\psi_0)_{-1,j} = \frac{h-2}{h+2}(\psi_0)_{1,j} + \frac{h}{h+2}\left(T_{j+1} - T_{j-1}\right) e^{-\eta_j} . \tag{2.6.18b}$$

At the outer boundary j = n and i = [1, n],

$$(\psi_0)_{i,n} = -\int_0^{\xi_i} e^{\xi} v_\eta \, d\xi , \tag{2.6.19a}$$

$$(\psi_0)_{i,n+1} = \psi_{0\,i,n-1} + 2h \; e^{\eta_\infty} u_\eta . \tag{2.6.19b}$$

$(\psi_0)_{i,n}$ and $u_\eta$ are solved numerically using the Simpson method. The program is listed in Appendix B.4.

At the outer boundary i = n and j = [1, n],

$$(\psi_0)_{n,j} = \int_0^{\eta_j} e^{\eta} u_\xi \, d\eta , \tag{2.6.20a}$$

$$(\psi_0)_{n+1,j} = (\psi_0)_{n-1,j} - 2h \; e^{\xi_\infty} v_\xi . \tag{2.6.20b}$$

$(\psi_0)_{n,j}$ and $v_\xi$ are solve numerically using the Simpson method. The program is listed in Appendix B.5.

The Gauss-Seidel iteration with under-relaxation is used to solve the algebraic equations. By changing the domain size $(\xi_\infty, \eta_\infty)$ = (5, 5), (6, 6), (7, 7), and (8, 8), and reducing the step size from 0.1, 0.05, 0.025 to 0.0125, the streamlines in Figure 7 are shown to be accurate to four decimal places. The streamlines show that at the liquid-vapor interface, the shear stress drives the liquid flow. The program for solving the stream functions is listed in Appendix B.6.

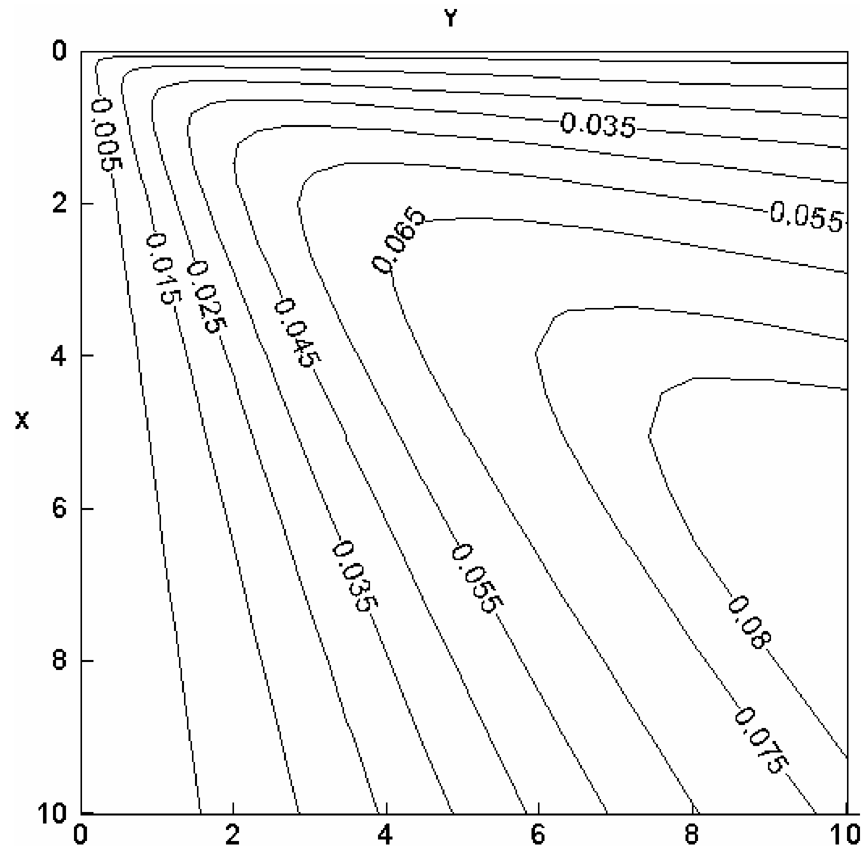


Figure 7. Streamlines of the Marangoni flow in the inner region. The X-axis is the wall surface and the Y-axis is the liquid-vapor interface. Liquid is dragged away from the hot wall by the Marangoni stress at the interface.

## 2.6.2. First-order velocity $\mathbf{u}_1$

In component form, $\mathbf{u}_1 = u_1\mathbf{i} + v_1\mathbf{j}$. A stream function $\psi_1$ is defined by

$$u_1 = \frac{\partial \psi_1}{\partial Y}, \quad (2.6.21a)$$

$$v_1 = -\frac{\partial \psi_1}{\partial X}. \quad (2.6.21b)$$

This stream function solves

$$\tilde{\nabla}^4 \psi_1 = 0, \quad (2.6.22)$$

and meets the no-slip boundary condition at the wall Y = 0:

$$\frac{\partial \psi_1}{\partial Y} = 0, \quad (2.6.23a)$$

$$\frac{\partial \psi_1}{\partial X} = 0. \quad (2.6.23b)$$

At the liquid-vapor interface X = 0, there is evaporation but no shear stress:

$$\frac{\partial \psi_1}{\partial Y} = -T_i = -\frac{2}{\pi} Y \int_0^\infty \frac{e^{-\lambda}}{\lambda^2 + Y^2}\, d\lambda, \quad (2.6.23c)$$

$$\frac{\partial^2 \psi_1}{\partial X^2} = 0. \quad (2.6.23d)$$

Far from the corner, as X or Y $\to \infty$,

$$\frac{\partial \psi_1}{\partial Y} \to 0, \quad (2.6.23e)$$

$$\frac{\partial \psi_1}{\partial X} \to 0. \quad (2.6.23f)$$

Thus, liquid flow at the first order is driven by evaporation at the interface.

Similar to that in the zero-order velocity field, an asymptotic solution is used to supply the boundary conditions at the outer domain boundaries. Since
the evaporation occurs mainly near the corner at the interface and decays to zero far from the wall, far from the corner the flow behaves like a sink flow. The radial velocity generated by a line sink on a planar wall is [17]

$$u_r = -\frac{4F}{\pi r} \sin^2\theta, \quad (2.6.24)$$

where F is the sink strength or the volume flow rate per unit length, r is the radial distance, and $\theta$ is the angle starting counter-clockwise from the wall. F is found by equating the leakage rate:

$$F = \int_0^{Y_\infty} u\,(0, Y)\, dY\,, \tag{2.6.25}$$

where $Y_\infty$ is the position of the outer domain boundary. At the interface X = 0, $u_1$ = - T. Thus,

$$F = -\frac{2}{\pi}\int_0^{Y_\infty} Y \int_0^{\infty} \frac{e^{-\lambda}}{\lambda^2 + Y^2}\, d\lambda\, dY\,. \tag{2.6.26}$$

Let $\lambda = s\,Y$, then

$$F = -\ \frac{2}{\pi}\int_0^{Y_\infty} \int_0^{\infty} \frac{e^{-s\,Y}}{s^2 + 1}\, ds\, dY\,. \tag{2.6.27}$$

Integration with respect to Y yields

$$F = \frac{2}{\pi}\left( Y_\infty \int_0^{\infty} \ln s\ e^{-s\,Y_\infty} ds - \int_0^{\infty} \frac{s}{s^2 + 1}\, e^{-s\,Y_\infty}\, ds \right). \tag{2.6.28}$$

This gives

$$F = -\frac{2}{\pi}\,\gamma + \frac{1}{\pi}\int_0^{\infty} \ln\left(s^2 + Y_\infty^2\right) e^{-s}\, ds\,, \tag{2.6.29}$$

where $\gamma \approx 0.577$ is the Euler number. Given $Y_\infty$, F is evaluated numerically by Simpson's integration method. Thus, velocity components at the outer domain boundaries $X = X_\infty$, or $Y = Y_\infty$ obey

$$\frac{\partial \psi_1}{\partial Y} = u_r\, \cos\ \theta\,, \tag{2.6.30a}$$

$$\frac{\partial \psi_1}{\partial X} = -\ u_r\, \sin\ \theta\,. \tag{2.6.30b}$$

Coordinates contraction is used again to make the asymptotic conditions valid, and they are defined the same as (2.6.9a) and (2.6.9b). And the governing

equation is the same as (2.6.10) except the subscript 0 needs to be displaced by the subscript 1.

At the wall $\eta = 0$,

$$\frac{\partial \psi_1}{\partial \eta} = 0 , \tag{2.6.31a}$$

$$\frac{\partial \psi_1}{\partial \xi} = 0 . \tag{2.6.31b}$$

At the liquid-vapor interface $\xi = 0$,

$$\frac{\partial \psi_1}{\partial \eta} = - \ e^{-\eta} T , \tag{2.6.32a}$$

$$\frac{\partial^2 \psi_1}{\partial \xi^2} = \frac{\partial \psi_1}{\partial \xi} \ . \tag{2.6.32b}$$

At the outer boundary $\eta = \eta_\infty$,

$$\frac{\partial \psi_1}{\partial \eta} = e^{-\eta} u_\eta \cos \theta_\eta , \tag{2.6.33a}$$

$$\frac{\partial \psi_1}{\partial \xi} = - \ e^{-\xi} u_\eta \sin \theta_\eta , \tag{2.6.33b}$$

where

$$\theta_\eta = \tan^{-1} \frac{\eta_\infty}{\xi} , \tag{2.6.33c}$$

$$u_\eta = - \frac{4 F \sin^2 \theta_\eta}{\pi \left(\xi^2 + \eta_\infty^2\right)^{1/2}} . \tag{2.6.33d}$$

At the outer boundary $\xi = \xi_\infty$,

$$\frac{\partial \psi_1}{\partial \eta} = e^{-\eta} u_\xi \cos \theta_\xi , \tag{2.6.34a}$$

$$\frac{\partial \psi_1}{\partial \xi} = - \ e^{-\xi} u_\xi \sin \theta_\xi , \tag{2.6.34b}$$

where

$$\theta_\xi = \tan^{-1} \frac{\eta}{\xi_\infty} , \tag{2.6.34c}$$

$$u_\xi = - \frac{4 F \sin^2 \theta_\xi}{\pi \left(\xi_\infty^2 + \eta^2\right)^{1/2}} . \tag{2.6.34d}$$

The stream function is solved by a finite-difference method. The discretized governing equation is the same as (2.6.15) and (2.6.16). The boundary conditions

are imposed as the follows. Take $\psi_1 = 0$ at the wall. Thus, at the wall j = 0 and i = [0, n],

$$(\psi_1)_{i,0} = 0 \,, \tag{2.6.35a}$$

$$(\psi_1)_{i,-1} = (\psi_1)_{i,-1} \,. \tag{2.6.35b}$$

At the liquid-vapor interface i = 0 and j = [1, n],

$$(\psi_1)_{0,j} = -\frac{2}{\pi}\gamma + \frac{1}{\pi}\int_0^{\infty} \ln\left[s^2 + \left(e^{\eta_j} - 1\right)^2\right] e^{-s}\, ds \,, \tag{2.6.36a}$$

$$(\psi_1)_{-1,j} = \frac{h-2}{h+2}(\psi_1)_{1,j} + \frac{4}{h+2}(\psi_1)_{0,j} \,. \tag{2.6.36b}$$

Sink strength F and $(\psi_1)_{0,\,j}$ are calculated numerically using the Simpson method. The program is listed in Appendix B.7.

At the outer boundary i = n and j = [1, n],

$$(\psi_1)_{n,j} = -\frac{4F}{\pi}\left(e^{\xi_\infty} - 1\right)\int_0^{\eta_j} \frac{e^{\eta}\left(e^{\eta} - 1\right)^2}{\left[\left(e^{\xi_\infty} - 1\right)^2 + \left(e^{\eta} - 1\right)^2\right]^2}\, d\eta \,, \tag{2.6.37a}$$

$$(\psi_1)_{n+1,j} = (\psi_1)_{n-1,j} - 2h\, e^{\xi_\infty} v_{n,j} \,, \tag{2.6.37b}$$

where

$$v_{n,j} = -\frac{4F}{\pi}\frac{\left(e^{\eta_j} - 1\right)^3}{\left[\left(e^{\xi_\infty} - 1\right)^2 + \left(e^{\eta_j} - 1\right)^2\right]^2} \,. \tag{2.6.37c}$$

$(\psi_1)_{n,\,j}$ and $v_{n,\,j}$ are calculated numerically. The program is listed in Appendix B. 8.

At the outer boundary j = n and i = [1, n]

$$(\psi_1)_{i,n} = (\psi_1)_{0,n} + \frac{4F}{\pi}\left(e^{\eta_\infty} - 1\right)^3 \int_0^{\xi_i} \frac{e^{\xi}}{\left[\left(e^{\eta_\infty} - 1\right)^2 + \left(e^{\xi} - 1\right)^2\right]^2}\, d\xi \,, \tag{2.6.38a}$$

$$(\psi_1)_{i,n+1} = (\psi_1)_{i,n-1} + 2h\, e^{\eta_\infty} u_{i,n} \,, \tag{2.6.38b}$$

where

$$u_{i,\,n} = -\frac{4F}{\pi}\left(e^{\eta_\infty}-1\right)^2 \frac{e^{\xi_i}-1}{\left[\left(e^{\eta_\infty}-1\right)^2+\left(e^{\xi_i}-1\right)^2\right]^2}. \qquad (2.6.38c)$$

$(\psi_1)_{i,\,n}$ and $u_{i,\,n}$ are calculated numerically. The program is listed in Appendix B.9.

The Gauss-Seidel iteration with under-relaxation is used to solve the algebraic equations. By changing the domain size $(\xi_\infty, \eta_\infty)$ = (5, 5), (6, 6), (7, 7), and (8, 8), and reducing the step size from 0.1, 0.05, 0.025 to 0.0125, the streamlines in Figure 8 are found to be accurate to four decimal places. It shows that at the liquid-vapor interface, liquid leaves the domain because of evaporation, and far from the corner, liquid enters into the domain to make the interface pinned. The program for calculating the stream functions is listed in Appendix B.10.

## 2.7. Evaporation Rate

The dimensionless evaporation rate at the interface x = 0 is

$$Q = -\int_0^1 u\,dy = \frac{1}{M}\int_0^1 T\,dy\ , \qquad (2.7.1)$$

where the vertical velocity u is related to the temperature by (2.3.5a).

The temperature field has an inner and outer structure, and the composite solution at x = 0 is

$$T = \frac{2\,Nu\,y}{\pi}\int_0^\infty \frac{e^{-\lambda}}{\lambda^2+Nu^2\,y^2}\,d\lambda + \frac{1}{Nu}\csc\left(\frac{\pi y}{2}\right) - \frac{2}{\pi\,Nu\,y}. \qquad (2.7.2)$$

The first term on the right side comes from the inner region, the second term from the outer region, and the last term from the matching region.

The evaporation rate along the interface is calculated in the Appendix A as

$$Q = \frac{2}{\pi M}\left[Nu^{-1}\ \ln Nu + \gamma\,Nu^{-1} + \ln\left(4/\pi\right) Nu^{-1}\right], \qquad (2.7.3)$$

where $\gamma$ (= 0.57722...) is the Euler number. The first two terms come from the inner

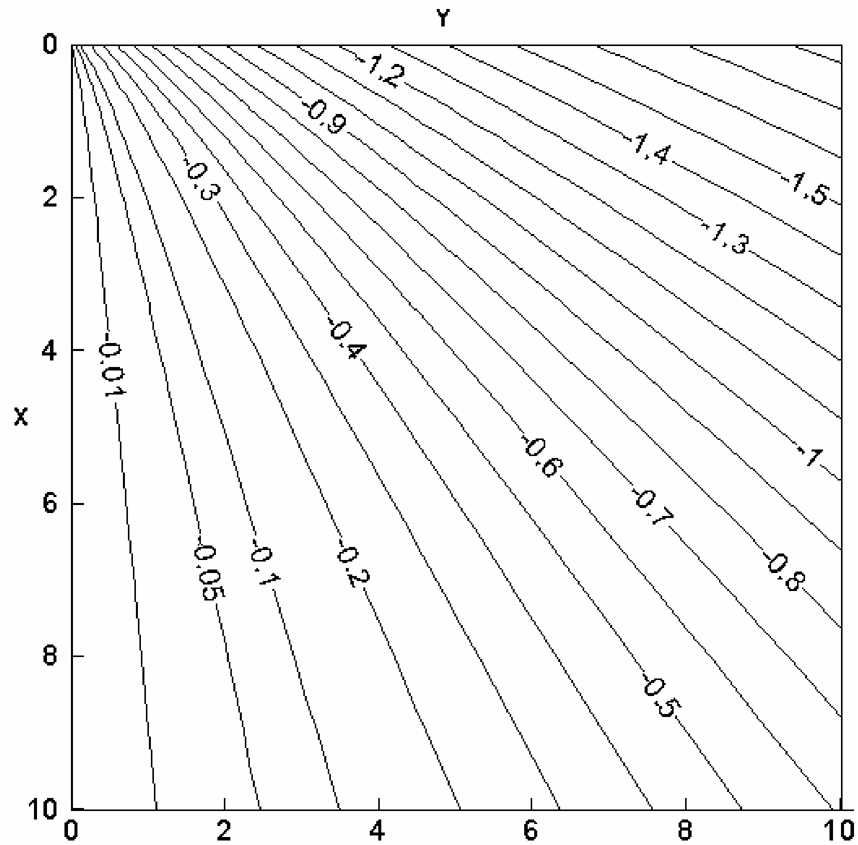


Figure 8. Streamlines near the corner. The Y-axis coincides with the interface, and the X-axis the wetting wall. Liquid leaves the interface because of evaporation. Far from the interface, liquid enters the domain to make the interface pinned.

region, and the third term is the net contribution from the outer region with the matching region subtracted. The largest evaporation rate comes from the inner region and is $O(Nu^{-1} \ln Nu)$. Our analysis of the outer region allows a precise comparison of evaporation from the outer region to that from the inner region. For example, for water, Nu = 1000 and M = 1000, the total evaporation rate Q is about $4.9 \times 10^{-6}$, while that comes from the inner region is about $4.4 \times 10^{-6}$, which stands for about 90% of the total evaporation rate.

## 2.8. Discussion

This chapter studies two-dimensional flow in a cross-sectional plane of an idealized micro heat pipe. Flow along the axial direction is assumed negligible compared with the in-plane flow. This is valid if the length of the heat pipe is much longer than its width. Thus, results obtained are the leading-order solution in the limit $L/W \to \infty$.

A constitutive equation of evaporation/condensation is used which specifies the evaporative mass flux to be proportional to the deviation from the equilibrium vapor pressure. This constitutive equation is physically sound because for evaporation to occur the system must deviate from the equilibrium state. This constitutive equation is also applied to predict condensation, and is taken to be the only mode of condensation. Deposition on the nonwetting wall at the cold end is assumed negligible. Since the wall is nonwetting, it is difficult for a liquid film to form. If a droplet forms on the wall, it will have high surface curvature given the large contact angle, and therefore continuous deposition on the droplet is unfavorable. Thus, application of the constitutive equation to model condensation is valid.

The two-dimensional domain is taken to be unbounded vertically (Figure 5). Liquid is assumed to enter at the bottom and leave at the top as vapor. This

assumption of unbounded domain does not affect the result significantly. As shown in this chapter, the interfacial temperature has an inner and outer structure. The inner and the intermediate regions are not affected by the domain shape. The outer solution does depend on the domain shape, and the unbounded domain allows an analytical solution of the temperature field. However, since the dominant contribution to evaporation comes from the inner region, and this dominant evaporation term is the only term used in Chapter 3, the unbounded domain assumption does not affect the final solution. It also has no effect on the liquid motion induced by evaporation and the Marangoni stress, because these flows to leading order in $Nu^{-1}$ occur only in the inner region. Hence, the simplification gained from the assumption of unbounded domain justifies its usage.

Convective heat transfer is negligible in the two-dimensional problem. The Peclet number ($P_e = V_M W / Nu\ \alpha$, where $\alpha$ is the heat diffusivity) measures the ratio of convective to conductive heat transfer. The Marangoni velocity scale $V_M$ is given in (2.3.1b). For water at 20 $^{\circ}$C, $\mu = 1.0 \times 10^{-3}$ kg/ms, $d\sigma / dT = 1.6 \times 10^{-4}$ N/mk, and $\alpha = 2.1 \times 10^{-4}$ m$^2$/s. Chapter 3 gives $T_\infty - T_c = 10$ $^{\circ}$C for W = 1 mm, L = 50 mm, and a temperature difference of 10 $^{\circ}$C across the whole pipe. Thus, $V_M$ = 1.60 m/s, and $P_e = 0.01$, indicating that convective heat transfer is negligible. In the definition of $P_e$, $Nu^{-1}W$ is used as the length scale. This is because the temperature varies mainly in the inner region, which is measured by a length scale of $Nu^{-1}W$.

The inertia effect is also negligible because the Reynolds number ($Re = \rho\ V_M W / Nu\ \mu$) is small. For water at 20 $^{\circ}$C, Re = 1.60. This Reynolds number is defined using the Marangoni velocity $V_M$. Since the Marangoni flow is nonzero only in the inner region, the inner length scale $Nu^{-1}W$ is used in defining Re. The Marangoni flow is decoupled from the evaporation-induced flow, and thus any

inertia effect on the Marangoni flow will have no effect on the evaporation rate. A Reynolds number based on the evaporation velocity scale $V_E$ will be even smaller since $V_E / V_M = M^{-1} << 1$.

The total evaporation rate is of order $Nu^{-1}$ ln Nu, which comes from the inner region near the contact line. The temperature varies smoothly in the inner region from 1 at the wall to 0 away from the wall. This is a result of the constitutive equation of evaporation. Without the constitutive equation, a non-integrable singularity in temperature appears at the contact line; the temperature jumps at the contact line from 1 at the wall to 0 at the interface. The wall is assumed isothermal, but this is not the reason for the inner region to exist. Even if the isothermal boundary condition of the wall is imposed at a distance away from the solid-liquid interface, an inner region still persists [14]. Thus, an inner region is intrinsic to evaporation near a solid wall.

## 2.9 Conclusions

An idealized micro heat pipe is proposed with simplified geometry while retaining the essential physics. This chapter studies flow and heat conduction in a cross-section plane. The wall temperature is assumed known and the vapor pressure is decreased below the equilibrium vapor pressure to induce evaporation. Two dimensionless parameters emerge: Nu and M, and the problem is solved in the limits $Nu^{-1} \to 0$ and $M^{-1} \to 0$. The temperature field has an inner region near the contact line defined by a length scale $Nu^{-1}W$. The interfacial temperature varies smoothly in the inner region from 1 at the wall to 0 away from the wall. This creates a Marangoni flow and an evaporation-induced flow, which are solved at different levels of expansion in $M^{-1}$. It is found that the total evaporation rate is $Nu^{-1}$ ln Nu to leading order. This is used in Chapter 3 to calculate heat transfer along the pipe.

# CHAPTER 3. FLUID FLOW AND HEAT TRANSFER

## 3.1. Introduction

Cotter in 1984 first introduced the concept of very small "micro" heat pipes incorporated into semiconductor devices to promote more uniform temperature distributions and improve thermal control [18]. Since then, various analytical models have been developed for predicating micro heat pipes' steady-state and transient performance characteristics. Axial temperature distribution and effective thermal conductivity are two basic characteristics measured and modeled.

In previous models, a micro heat pipe is assumed to contain three regions, and they are treated separately: an evaporator and a condenser, separated by an adiabatic section. Temperature profile in each region is solved and the results are then assembled to give the overall distribution of the axial temperature [19]. However, this prediction of axial temperature has kinks at the interceptions between two sections. Experimental results do not show the adiabatic region at all [8, 20, 21, 22]. Temperature decreases smoothly from the evaporator to the condenser. The model presented in this work explains why the adiabatic region, which is commonly seen in experimental results of conventional heat pipe, is not seen in micro heat pipes.

This work derives an equation that contains all three regions. An effective thermal conductivity is calculated. It depends on many factors, including the length and width of the pipe, the latent heat of the liquid.

Other characteristics are also studied by using the axial heat transfer and liquid-vapor flow model presented in this chapter. Vapor and liquid pressures are calculated. This allows the interfacial curvature to be calculated.

## 3.2. Formulation

Figure 9 shows half of a micro heat pipe, which has a rectangular cavity of dimensions 2W×4W×L surrounded by a uniform wall of thickness H. The lower half of the pipe is made of a wetting material, and the top half a nonwetting material. A wetting liquid fills the lower half of the cavity, whereas its vapor fills the rest. This micro heat pipe pins the liquid-vapor interface at the contact line. Initially, the pipe is at a uniform temperature $T_m$. One end of the heat pipe is then heated to temperature $T_m + \Delta T$, and the other end cooled to $T_m - \Delta T$. The temperature difference is maintained and the heat pipe reaches a steady state. In general, $\Delta T << T_m$, and the flow and temperature distributions along the pipe are symmetric. Thus, only half of the pipe needs to be studied.

Chapter 2 studies the evaporation and condensation in the idealized micro heat pipe. It is found that the maximum evaporation or condensation occurs near the contact line, and the liquid volume evaporation rate per unit contact line length is

$$Q^* = \frac{2 k_f \left( \ln Nu \right)}{\pi \rho_f h_{fg}} \left( \left( T^* - T_m \right) - \frac{T_m}{\rho_m h_{fg}} \left( p_g^* - p_m \right) \right), \tag{3.2.1a}$$

$$Nu = \frac{h W}{k}, \tag{3.2.1b}$$

$$h = \frac{\rho_m h_{fg}^2}{\left( 2 \pi R T_m \right)^{1/2} T_m}, \tag{3.2.1c}$$

where $T^*$ is the wall or liquid temperature, $p_g^*$ is the vapor pressure, $T_m$, $P_m$ and $\rho_m$ are, respectively, the temperature, equilibrium vapor pressure, and equilibrium vapor density at the mid-point of the pipe, $k_f$ is the liquid thermal conductivity, $\rho_f$ is the liquid density, and $h_{fg}$ is the liquid latent heat. This evaporation rate is derived from Chapter 2 by expanding the equilibrium vapor pressure ($p_\infty$) at the wall temperature ($T_\infty$) about the mid-point temperature $T_m$, with the gradient $dp_\infty / dT_\infty = \rho_m h_{fg} / T_m$ given by the Clausius-Clapeyron relation. Since $T^*$ is close to $T_m$ and $p_g^*$

to $P_m$, only linear terms in $(T^* - T_m)$ and $(p_g^* - P_m)$ are kept in this work. Near the hot end, liquid evaporates into vapor ($Q^* > 0$), which carries heat energy to the cold end where it condenses ($Q^* < 0$) and releases the latent heat.

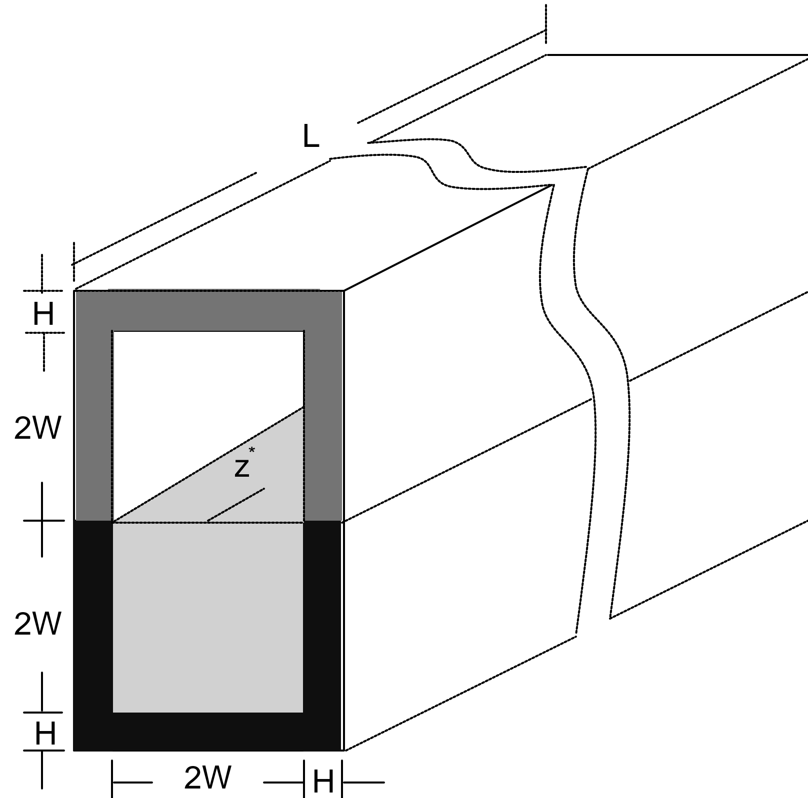


Figure 9. Half of an idealized micro heat pipe. The cavity of the heat pipe is of dimensions $2W \times 4W \times L$ and is surrounded by a uniform wall of thickness H. The axial coordinate $z^*$ starts at the mid-point and points towards the evaporator end.

Fluid flow along the pipe is taken as unidirectional because the pipe is slender. Thus, the pressure gradient varies linearly with the local flow rate:

$$\frac{dp_i^*}{dz^*} = - \frac{\mu_i \, V_i^*}{C \, W^4} \, , \; i = g, f \qquad (3.2.2)$$

where $p^*$ is pressure, $z^*$ is a coordinate starting at the mid-point of the pipe pointing towards the hot end (Figure 9), $\mu$ is viscosity, $V^*$ is the volume flow rate in the $z^*$ direction, and C is a constant that depends on the cross-sectional shape. For a square cross-section of width 2W, C = 0.5623 [23]. The subscript corresponds to vapor (g) or liquid (f).

The volume flow rate varies along the pipe due to evaporation. A local mass balance gives

$$\frac{dV_g^*}{dz^*} = \frac{\rho_f \, Q^*}{\rho_m} \qquad (3.2.3a)$$

$$\frac{dV_f^*}{dz^*} = - \, Q^* \, . \qquad (3.2.3b)$$

Thus, the liquid and vapor pressures obey

$$\frac{d^2 p_g^*}{dz^{*\,2}} + \frac{\mu_g \, \rho_f}{C \, W^4 \, \rho_m} Q^* = 0 \qquad (3.2.4a)$$

$$\frac{d^2 p_f^*}{dz^{*\,2}} - \frac{\mu_f}{C \, W^4} Q^* = 0 \, . \qquad (3.2.4b)$$

Mass conservation requires that at any point along the pipe,

$$\rho_f V_f^* + \rho_m V_g^* = 0 \, . \qquad (3.2.5)$$

Combining (3.2.2) and (3.2.5) gives

$$\frac{dp_f^*}{dz^*} = - \frac{\mu_f \, \rho_m}{\mu_g \, \rho_f} \frac{dp_g^*}{dz^*} \, . \qquad (3.2.6)$$

Thus, the liquid pressure gradient is proportional to the vapor pressure gradient, and they have opposite signs.

At the mid-point $z^* = 0$, the pressure is assumed to be the equilibrium vapor pressure corresponding to temperature $T_m$:

$$p_g^* = p_f^* = P_m . \tag{3.2.7}$$

At the hot end $z^* = L$,

$$p_g^* = P_g \tag{3.2.8a}$$

$$p_f = P_f . \tag{3.2.8b}$$

These two pressures are unknown and need to be determined.

The curvature of the liquid-vapor interface is determined by the Young-Laplace equation:

$$p_g^* - p_f^* = \frac{\sigma}{R^*} , \tag{3.2.9}$$

where $\sigma$ is the surface tension and $R^*$ is the radius of curvature of the interface.

Heat conduction along the heat pipe is taken to be one dimensional. The pipe is insulated outside, and at each point along the pipe, the liquid temperature is assumed to be the same as the wall temperature $T^*$. The wall-liquid system loses heat by evaporation. An energy balance gives

$$\left(A_w k_w + A_f k_f\right) \frac{d^2 T^*}{dz^{*2}} - \rho_f h_{fg} Q^* = 0 , \tag{3.2.10}$$

where A is cross-sectional area and k is thermal conductivity with the subscript indicating either wall (w) or liquid (f). At the mid-point $z^* = 0$,

$$T^* = T_m . \tag{3.2.11}$$

At the hot end $z^* = L$,

$$T^* = T_h . \tag{3.2.12}$$

These two temperatures are specified.

Since the heat pipe is insulated outside, heat flow $q^*$ along the pipe is constant:

$$q^* = (A_w k_w + A_f k_f) \frac{dT^*}{dz^*} - \rho_m V_g^* h_{fg} , \quad (3.2.13)$$

where the first term on the right side is the conductive heat flux and the second the heat flux carried by the vapor flow. This equation can be derived by substituting $Q^*$ in (3.2.3a) into (3.2.10), and integrating once. At the hot end $z^* = L$, the heat flow rate is

$$q_h^* = (A_w k_w + A_f k_f) \frac{dT^*}{dz^*} + 2 W \rho_f Q^* h_{fg} . \quad (3.2.14)$$

The first term is the conductive heat input into the liquid and wall, whereas the second term is the evaporative heat input through the contact line length 2W at the end of the pipe. This contact line length introduces the existence of an end of the pipe into the problem. The heat flow rate $q_h^*$ is equated to $q^*$ to complete the problem.

## 3.3. Dimensionless Equations

Define a set of dimensionless variables:

$$z = \frac{z^*}{L}, \ p_g = \frac{p_g^* - P_m}{P_g - P_m}, \ p_f = \frac{p_f^* - P_m}{P_g - P_m}, T = \frac{T^* - T_m}{T_h - T_m}, R = \frac{R^*}{W},$$

$$q = \frac{q^* L}{(A_w k_w + A_f k_f)(T_h - T_m)}, \ V_g = \frac{\mu_g L V_g^*}{(P_g - P_m) C W^4} .$$

Since the heat flow rate q along the pipe is constant, (3.2.13) gives

$$\frac{dq}{dz} = \frac{d^2T}{dz^2} - E \frac{dV_g}{dz} = \frac{d^2}{dz^2}(T + E p_g) = 0 \quad (3.3.1a)$$

$$E = \frac{C W^4 \rho_m h_{fg} (P_g - P_m)}{\mu_g (A_w k_w + A_f k_f)(T_h - T_m)} , \quad (3.3.1b)$$

where $V_g$ is replaced by $dp_g / dz$ by (3.2.2), and $E = q - 1$, found by (3.4.9), represents the portion of evaporation heat transfer carried by vapor flow. Thus, $T + E p_g$ is a linear function of z. Hence, once T is determined, (3.3.1) gives $p_g$. Since

the temperature is usually measured in experiments, it is solved first. Equation (3.2.10) together with (3.2.1) is made dimensionless as

$$\frac{d^2T}{dz^2} - B\,T + G\quad p_g = 0\,. \tag{3.3.2a}$$

$$B = \frac{2\ k_f\ L^2 \ln Nu}{\pi\left(A_w k_w + A_f k_f\right)}\,, \tag{3.3.2b}$$

$$G = B\ \frac{T_m\left(p_g - P_m\right)}{\rho_m\ h_{fg}\left(T_h - T_m\right)} = B\ \frac{p_g - P_m}{P_{he} - P_m}\,. \tag{3.3.2c}$$

Here, $P_{he}$ is the equilibrium vapor pressure at $T_h$, and $B = 1 / \delta^2$ as discussed later in (3.4.8). If $A_f = 4W^2$, which is true for the micro heat pipe presented in Chapter 2, then B is simplified:

$$B = \frac{\ln Nu}{2\ \pi\left(1 + r\right)}\left(\frac{L}{W}\right)^2\,. \tag{3.3.2.d}$$

Here, $r = A_w k_w / A_f k_f$. At the mid-point z = 0,

$$T = 0. \tag{3.3.3}$$

At the hot end z = 1,

$$T = 1, \tag{3.3.4}$$

and the nondimensionlized heat flow rate is

$$q_h = \frac{dT}{dz} + \frac{2\,W}{L}\left(B\,T - G\quad p_g\right)\,. \tag{3.3.5a}$$

The nondimensionlized heat flow rate is constant along the pipe:

$$q = \frac{dT}{dz} + E\ \frac{dp_g}{dz}\,. \tag{3.3.5b}$$

This is evaluated at the mid-point z = 0, and equated to $q_h$ to complete the problem.

The liquid pressure is related to the vapor pressure by

$$\frac{dp_f}{dz} = -\ \frac{\mu_f\ \rho_m}{\mu_g\ \rho_f}\ \frac{dp_g}{dz}\,. \tag{3.3.6}$$

At the mid-point z = 0,

$$p_g = 0\,, \tag{3.3.7a}$$

$$p_f\ = 0\,. \tag{3.3.7b}$$

At the hot end $z = 1$,

$$p_g = 1\ , \tag{3.3.8a}$$

$$p_f = \frac{P_f - P_m}{P_g - P_m}\ . \tag{3.8b}$$

The curvature of the liquid-vapor interface is

$$p_g - p_f = \frac{\sigma}{W\left(P_g - P_m\right)} \frac{1}{R}\ . \tag{3.3.9}$$

## 3.4. Axial Temperature

Integrating equation (3.3.1a) together with applying boundary conditions (3.3.3)-(3.3.4) and (3.3.7a)-(3.3.8a) gives

$$T + E\ p_g = \left(1 + E\right) z\ . \tag{3.4.1}$$

Therefore, $p_g$ is expressed in terms of T as

$$p_g = -\frac{T}{E} + \frac{1 + E}{E} z\ . \tag{3.4.2}$$

This together with (3.3.2a) gives the governing equation of temperature, which is decoupled from vapor pressure $p_g$, as

$$\frac{d^2T}{dz^2} - \left(B + S\right) T = -\left(1 + E\right) S\, z\ , \tag{3.4.3}$$

Where

$$S = \frac{2\ k_f\ \mu_g\ T_m\ L^2\ \ln Nu}{\pi\ C\ W^4\ \rho_m^2\ h_{fg}^2} = G / E\ . \tag{3.4.4}$$

S is related to physical and geometric characteristics of the problem, and thus is known.

A General solution to the temperature field (3.4.3) is

$$T = A \sinh\left(\sqrt{B + S}\ z\right) + \frac{\left(1 + E\right) S}{S + B} z\ , \tag{3.4.5}$$

where A is an unknown constant to be determined. The boundary condition of T = 1 at z = 1 gives

$$A = \frac{B - G}{B + S} \frac{1}{\sinh\left(\sqrt{B + S}\right)}\ . \tag{3.4.6}$$

To solve the unknown parameter G and E, $q_h$ in (3.3.5a) is equated to q in (3.3.5b). This together with the relationship between $p_g$ and T in (3.4.1) gives

$$G = \frac{\dfrac{B \cosh \sqrt{B+S}}{\sqrt{B+S}\, \sinh \sqrt{B+S}} - \dfrac{B}{B+S} + \dfrac{2W}{L} B}{\dfrac{\cosh \sqrt{B+S}}{\sqrt{B+S}\, \sinh \sqrt{B+S}} + \dfrac{1}{S} - \dfrac{1}{B+S} + \dfrac{2W}{L}}, \qquad (3.4.7a)$$

$$E = G / S . \qquad (3.4.7b)$$

Thus, the temperature field is solved and its distribution is plotted in Figure 10. The temperature profile is relatively flat along the micro heat pipe except near the hot end, which is the evaporation region, where a significant temperature jump occurs. And the length of that region decreases when the ratio L/W increases. It is also affected by the ratio r. For larger r, the length of the evaporation region increases; consequently, the adiabatic region disappears. Using the same dimension of the micro heat pipe fabricated [21], this effect is illustrated in Figure 11.

The temperature profile indicates that an inner solution exists near the end. From the temperature solution (3.4.5)-(3.4.6), 1/(S+B) is of order $10^{-4}$ for water and water vapor at T = 20°C. This is because of the disappearance of the second-order differential term in the temperature governing equation (3.4.3). This introduces an inner solution and an outer solution. Because B >> S, to the leading order, we can define a small parameter $\delta$

$$\delta = \sqrt{\frac{\pi \left(A_w k_w + A_f k_f\right)}{2\, k_f\, L^2 \ln Nu}} = \sqrt{\frac{1}{B}} . \qquad (3.4.8)$$

Because $\delta << 1$, the asymptotic behavior of $\sinh(z/\delta)/\sinh(1/\delta)$ is $e^{(z-1)/\delta}$. This defines an inner solution, which is scaled by $\delta$. To the leading order, the outer solution gives a linear temperature profile.

The constant heat transfer q, thus, is calculated. Combining (3.3.5b) and (3.4.2) gives

$$q = 1 + E . \qquad (3.4.9)$$

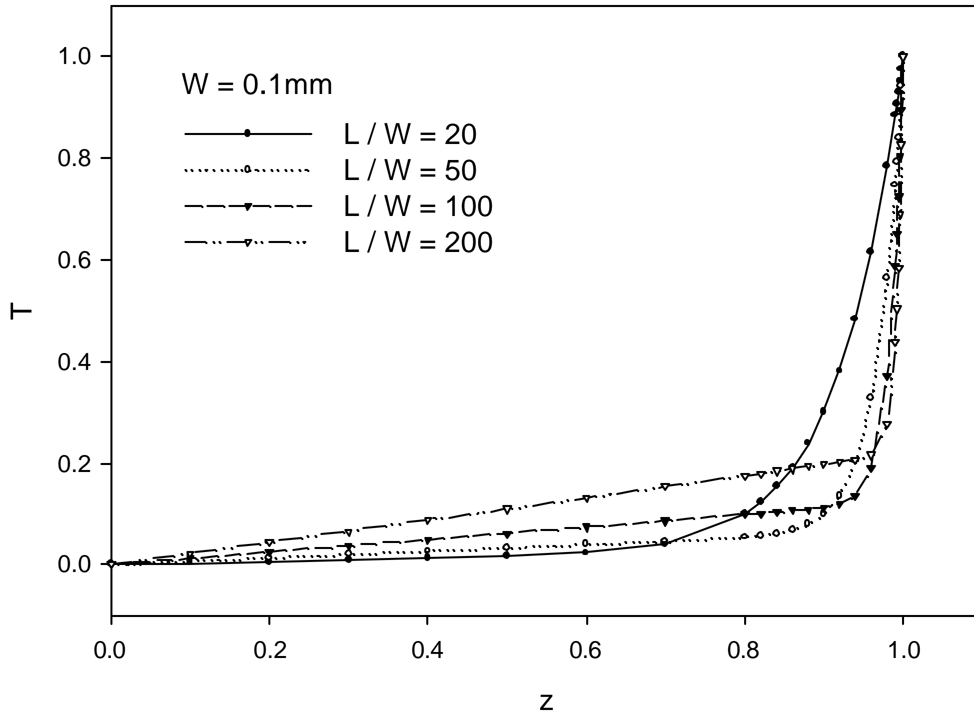


Figure 10. Axial temperature distribution along an idealized micro heat pipe. L is the half length of the pipe, and W is the half width of the pipe.

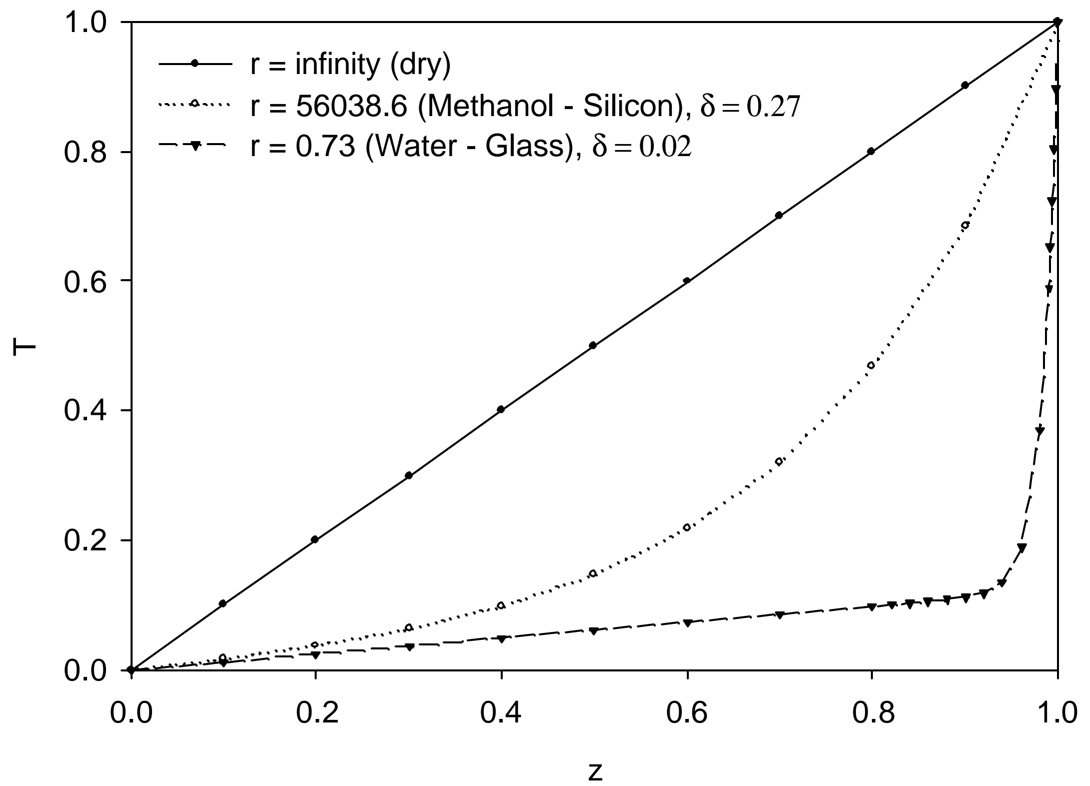


Figure 11. Axial temperature distribution along an idealized micro heat pipe for different r. $r = A_w k_w / A_f k_f$.

## 3.5. Axial Pressure

Once the temperature is known the vapor pressure follows from (3.4.2),

$$p_g = -\frac{A}{E}\sinh\left(\sqrt{B+S}\; z\right) + \frac{B}{B+S}\left(1+\frac{1}{E}\right) z\,, \qquad (3.5.1)$$

where A is the same as (3.4.6).

The vapor pressure along the pipe is plotted in Figure 12. It demonstrates that the vapor pressure is approximately linear and is not sensitive to the ratio L / W.

Liquid pressure is solved by integrating (3.3.6) with respect to z and applying boundary condition (3.3.7):

$$p_f = -\frac{\mu_f\;\rho_m}{\mu_g\;\rho_f}\,p_g\,. \qquad (3.5.2)$$

## 3.6. Liquid-vapor Interfacial curvature

The interfacial curvature 1/R can be found by combining (3.3.9) and (3.5.2):

$$\frac{1}{R} = \frac{\Delta p\, W}{\sigma}\left(1+\frac{\rho_m\;\mu_f}{\rho_f\;\mu_g}\right) p_g\,, \qquad (3.6.1)$$

where $\Delta p = P_g - P_m$.

The interfacial curvature along the pipe is plotted in Figure 13 for W = 1 mm and 0.1 mm, and L / W = 100. It varies along the pipe linearly with the maximum at the hot end. If L / W is constant, the curvature will decrease when L increases.

For water at 20 °C, $\mu_f\rho_m$ / $\mu_g\rho_f$ is of order $10^{-3}$. Therefore, the dominant factor that controls the interfacial curvature is the dimensionless pressure drop $\Delta pW/\sigma$, which itself is affected by L / W. This is demonstrated in Figure 14; $\Delta pW$ / $\sigma$ increases with L if W is kept constant, and it decreases with W if L / W is kept constant.

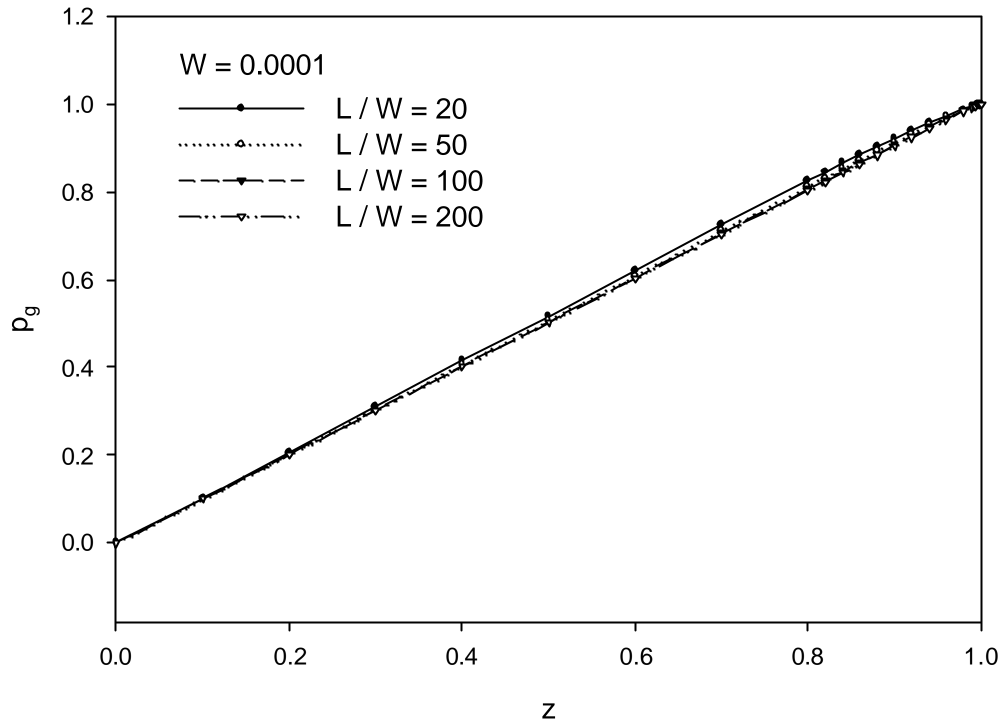


Figure 12. Axial vapor pressure distribution along an idealized micro heat pipe.

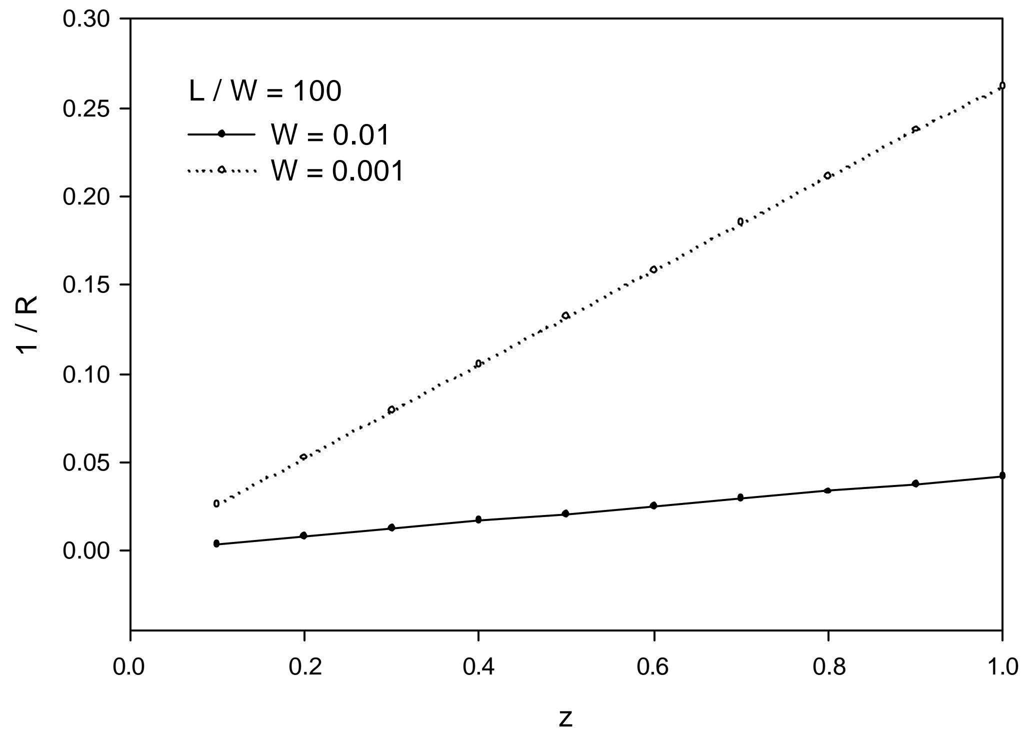


Figure 13. Dimensionless curvature along an idealized micro heat pipe.

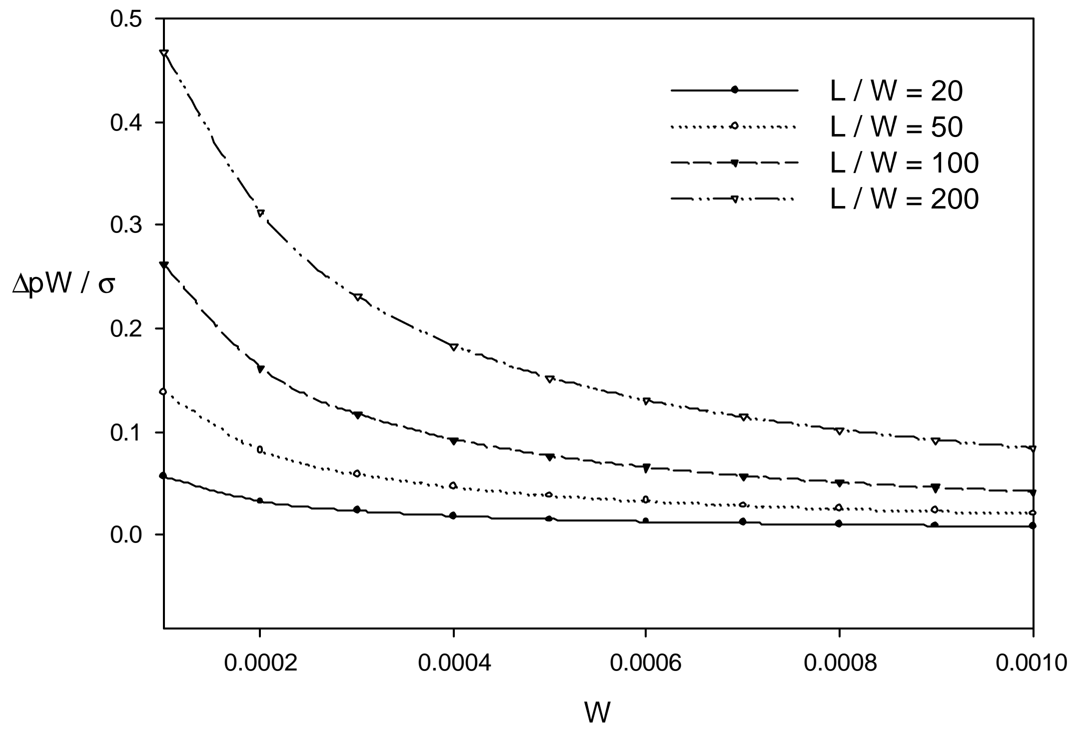


Figure 14. Relationship between the dimensionless pressure drop and the width of an idealized micro heat pipe.

## 3.7. Effective Thermal Conductivity

One major task of this Chapter is to examine the factors that may affect the effective thermal conductivity in an idealized micro heat pipe. The effective thermal conductivity $k_e$ is

$$k_e = \frac{q\,L}{A_e\,(T_h - T_m)}, \tag{3.7.1}$$

where $q$ is the heat transfer along the pipe and is calculated by (3.4.9), $A_e$ is the effective area: $A_e = A_w + A_f + A_g$, where $A_g$ is the portion of the cross sectional area filled with vapor.

Thus, $k_e$ can be solved by (3.7.1) and (3.4.9). Effects of length and width of a pipe on $k_e$ are examined in Figure 15. It is shown that $k_e$ increases with L if W is kept constant, and it increases with W if L / W is kept constant. The latent heat of the working fluid inside the micro heat pipe will also affect the effective thermal conductivity. In Figure 16, it is shown that the effective thermal conductivity increases when the latent heat of the working fluid increases. This indicates that to improve the heat transfer of the micro heat pipe, fluids with higher latent heat are recommended.

## 3.8. Discussion

The small parameter $\delta$ occurring in the temperature governing equation introduces an inner region near the evaporator end. Within that region, temperature jumps significantly. Thus this region is identified as the evaporation region, where heat transfer by evaporation is dominant compared with that by conduction. Away from that region conduction becomes dominant. Therefore, the temperature profile tends to be linear. The length of that region depends on the width and length of the pipe as well as the ratio $r = A_w k_w / A_f k_f$. For water at 20°C, $\delta W$ is 12.8 mm when W = 1 mm and L = 100 mm, and $\delta W$ is 1.59 mm when W = 0.1 mm and L = 10 mm. This region is always captured in conventional heat pipes

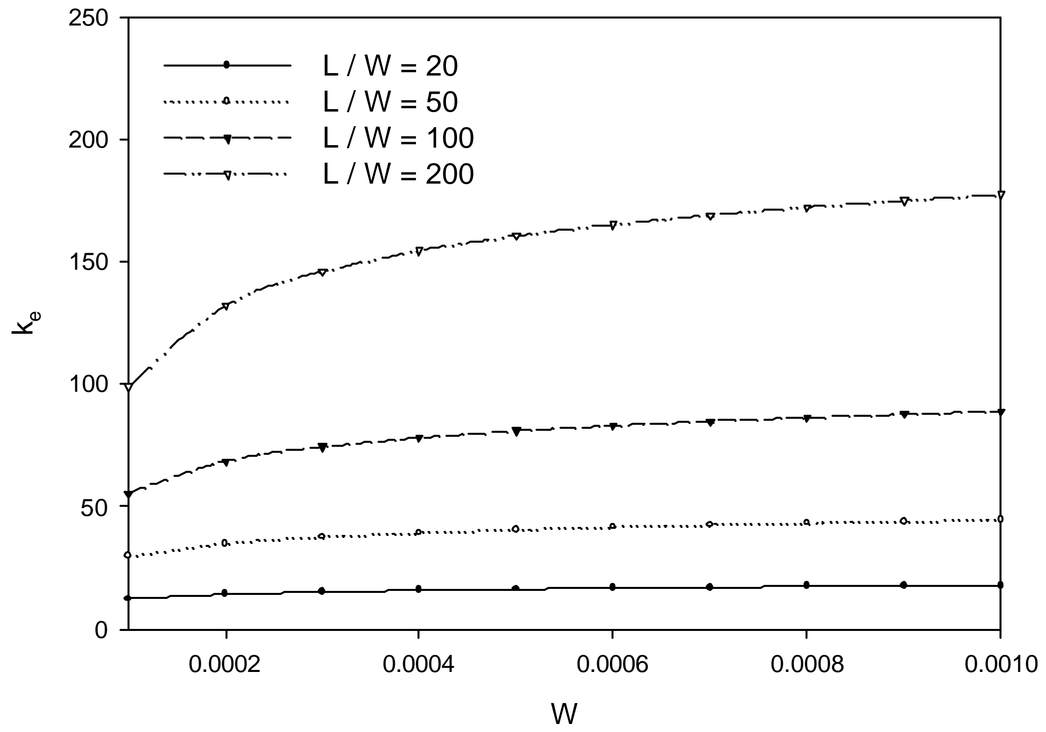


Figure 15. Relationship between the effective thermal conductivity and the width of an idealized micro heat pipe.

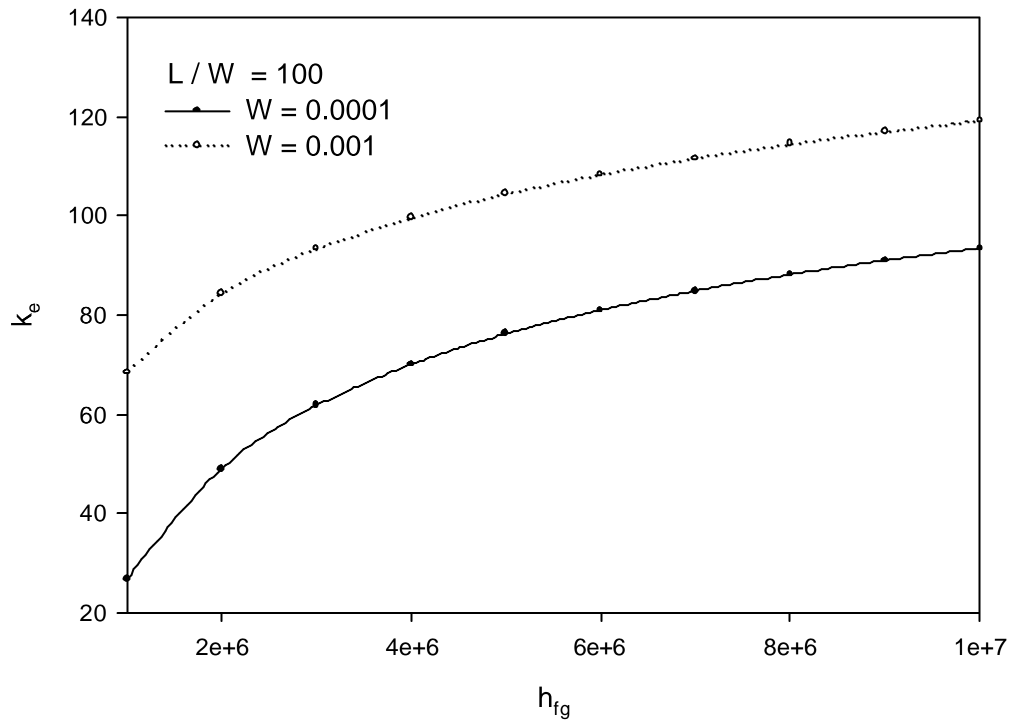


Figure 16. Relationship between the effective thermal conductivity and the latent heat of the working fluid in an idealized micro heat pipe.

experiments but not in micro heat pipes. It is caused by the value of r. The parameter $\delta$ can be considered as the ratio of the conduction to evaporation. In the limit $r \rightarrow \infty$, the micro heat pipe is dried out and only heat conduction occurs, $\delta \rightarrow \infty$ making the axial temperature profile linear. When r decreases, heat is then transferred by both conduction and evaporation. This decreases the length of the evaporation region $\delta$.

## 3.9. Conclusions

In the axial direction, a major temperature jump occurs near the evaporator end. The length of that region is affected by the ratio L / W and $r = A_w k_w / A_f k_f$. For a larger r, the temperature profile tends to be linear, which results in the disappearance of the adiabatic region.

Liquid-vapor interfacial curvature varies along the pipe with the largest curvature at the hot end. It is controlled by the dimensionless pressure drop $\Delta pW / \sigma$. This pressure drop increases with L if W is constant, and it decreases with W if L / W is kept constant.

Effective thermal conductivity increases with L if W is kept constant, and it increases with W if L / W is kept constant. It will also increase if the latent heat of the working fluid is larger.

# CHAPTER 4. CONCLUSIONS

The idealized micro heat pipe presented in this work removes the complicated geometry, simplifies the analysis of evaporation and condensation in micro heat pipes, and retains the essential physics. The evaporative mass constitutive equation holds for both evaporation and condensation, and allows us to only examine the evaporation occurring in the pipe.

Flow and heat conduction in a cross-section plane, with constant wall temperature, is studied. Two dimensionless parameters, Nusselt number Nu and Marangoni number M, determine the two-dimensional motion and heat transfer and introduce an inner and an outer region. In the limits $Nu^{-1} \rightarrow 0$ and $M^{-1} \rightarrow 0$, the interfacial temperature is solved. The interfacial temperature decreases smoothly from 1 at the wall to 0 far away in an inner region scaled by $Nu^{-1}W$. The Marangoni flow and the evaporative flow caused by this interfacial temperature structure are solved at different levels by expanding in $M^{-1}$. The evaporation rate is thus calculated, which leads to one of our important conclusions that the evaporation rate is of order $Nu^{-1} \ln Nu$ and it occurs mainly near the contact line region.

Based on this evaporation rate, axial heat transfer and liquid and vapor flow are studied with the assumption that the liquid has the same temperature as the wall and the heat flux is constant along the pipe. Three basic regions (evaporation, adiabatic, and condensation), commonly observed in experiments

of conventional heat pipes are predicted by our model. The length scale for the evaporation/condensation region is found. It explains why the adiabatic region is not usually observed in experiments of micro heat pipes. Vapor pressure and liquid pressure are solved and they yield the interfacial curvature. The parameters that control the curvature are also illustrated. Effective thermal conductivity is calculated and ways to improve it are suggested.

# REFERENCES

1. Peterson, G. P., *An Introduction to Heat Pipes*, John Wiley & Sons, Inc., New York, (1994).

2. Chi, S. W., *Heat Pipe Theory and Practice*, McGraw-Hill, New York, (1976).

3. Dunn, P. D., Reay, D. A., *Heat Pipes*, Pergamon Press, Oxford (1978).

4. Ivanovskii, M. N., Sorokin, V. P., and Yagodkin, I. V., *The Principles of Heat Pipes*, Clarendon Press, Oxford, (1982).

5. Andrews, J., Akbarzadeh, A., and Sauciuc, I., Heat Pipe Technology, Pergamon Press, Oxford (1997).

6. Peterson, G. P. *Appl. Mech. Rev.* **45**, 175 (1992).

7. Wong, H., Morris, S. and Radke, C. J., *J. Colloid Interface Sci.* **148**, 317 (1992).

8. Peterson, G. P., *J. Heat Transfer* **115**, 751 (1993).

9. Badran, B., Gerner, F. M., Ramadas, P., Henderson, T., and Baker, W. W., *Experimental Heat Transfer* **10**, 253 (1997).

10. Plesset, M. S. and Prosperetti, A., *J. Fluid Mech.* **78**, 433 (1976).

11. Moosman, Steven and Homsy, G. M., *J. of Colloid and Interface Science* **73**, 212 (1980).

12. Cammenga, H. K. *Current Topics in Material Science* **5**, Chap. 4., North-Holland (1980).

13. Morris, S. J. S., *J. Fluid Mech.* **432**, 1 (2001).

14. Morris, S. J. S., *J. Fluid Mech.* **411**, 59(2000).

15. Churchill, R. V. and Brown J. W., *Complex Variables and Applications*, McGraw-Hill Book Company, New York, (1984).

16. Liron, N and Blake J. R., *J. Fluid Mech.* **107**, 109 (1981).

17. White, F. M., *Viscous Fluid Flow*, McGraw-Hill, Inc., New York, (1991).

18. Cotter, T. P., *Research and development of heat pipe technology, Proceeings of 5IHPC Part IV*, 328 (1984).

19. Wu, D. and Peterson, G. P., *J. Thermophysics* **5**, 129 (1991).

20. Duncan, A. B., *An Experimental and Analytical Evaluation of Etched Micro Heat Pipes*, Ph.D. dissertation, Texas A & M University, (1993).

21. Peterson, G. P., Duncan A. B. and Weichold, M. H., *Journal of Heat Transfer* **115**, 751 (1993).

22. Park, J. S., Choi, J. H., Cho, Ya ng, S. S., and Yoo, J. S., *Proceedings of SPIE* **4408**, 424 (2001).

23. Rosenhead, L., *Laminar Boundary Layers*, Dover Pubns (1998).

# APPENDIX A. TOTAL EVAPORATION RATE AT THE INTERFACE

The total evaporation rate at the interface x = 0 in half of the domain shown in Figure 5 is

$$Q = \frac{1}{M}\int_0^1 T\,dy\,, \tag{A1}$$

where T is the interfacial temperature:

$$T = \frac{2\,Nu\,y}{\pi}\int_0^\infty \frac{e^{-\lambda}}{\lambda^2 + Nu^2\,y^2}\,d\lambda + \frac{1}{Nu}\csc\left(\frac{\pi\,y}{2}\right) - \frac{2}{\pi\,Nu\,y}\,. \tag{A2}$$

The evaporation rate Q is found in the limit $Nu^{-1} \rightarrow 0$. The first term on the right side is the inner interfacial temperature. Its contribution to the evaporation rate is

$$Q_1 = \frac{2\,Nu}{\pi\,M}\left(\int_0^\infty e^{-\lambda}\int_0^1 \frac{y}{\lambda^2 + Nu^2 y^2}\,dy\,d\lambda\right), \tag{A3}$$

which is simplified to

$$Q_1 = \frac{1}{M}\left(-\frac{2}{\pi}Nu^{-1}\ \ln\ Nu^{-1} + \frac{2}{\pi}\gamma + \frac{1}{\pi\,Nu}\int_0^\infty e^{-\lambda}\ln\left(1 + Nu^{-2}\lambda^2\right)d\lambda\right), \tag{A4}$$

where $\gamma \approx 0.57722...$) is the Euler number. The last term is of order $Nu^{-3}$. Thus, to the leading order,

$$Q_1 = \frac{2}{\pi\,M}\left(Nu^{-1}\ \ln Nu + \gamma\,Nu^{-1}\right). \tag{A5}$$

The second and the third terms on the right side of (A2) come from the outer and the matching regions. Their contributions to the evaporation rate can be calculated exactly. Thus, the evaporation rate Q is

$$Q = \frac{2}{\pi M}\left(- Nu^{-1} \ln Nu^{-1} + Nu^{-1}\gamma + Nu^{-1} \ln (4/\pi)\right) . \qquad (A6)$$

# APPENDIX B. PROGRAMS

## B.1. BASIC OPERATIONS (BasicOperation.h)

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE1 161
#define SIZE2 161

/* Usage: open a file with a specified name and mode
   Input: a string containing the file's name and the other the open mode
   Output: a pointer pointing to an opened file
*/
FILE* openFile(char* fileName, char* openMode){
        FILE *pFile;
        pFile = fopen(fileName, openMode);
        return pFile;
}

/* Usage: judge if a file is opened correctly or not
  Input: pFile: a pointer pointing to a file
  Output: if the file is not opened correctly, print an error message and exit the
program
*/
void judgeFileOpenCorrectly(FILE* pFile, char* fileName){
        if(pFile == NULL){
                printf("%s can not be opened.\n", fileName);
                exit(-1);
        }
}

/* Usage: close a specified file
   Input: pFile: a pointer pointing to a file
   Output: an integer indicating if the file is closed correctly or not(-1)
*/
int closeFile(FILE *pFile){
        return fclose(pFile);
}

/* Usage: judge if a file is closed correctly or not
   Input: an integer returned from function fclose, and the file's name
   Output: if the file is not closed correctly, print an error message and exit
*/
```

```
void judgeFileCloseCorrectly(int fileIsClose, char * fileName){
        if(fileIsClose == -1)
        {
                printf("%s is not closed correctly!\n",  fileName);
                exit(-1);
        }
}

/* Usage: Read one double value from a specified file
   Input: pFile: a pointer pointing to a file
          d1: pointer pointing to a double variable
   Output: the updated double variable
*/
void readOneDoubleValue(FILE* pFile, double *d1){
        fscanf(pFile, "%lf", d1);
}

/* Usage: Read two double variables from a specified file
   Input: pFile: a pointer pointing to a file
          d1 & d2: pointers pointing to two double variables
   Output: the updated two double variables with value read from the file
*/
void readTwoDoubleValues(FILE * pFile, double *d1, double *d2){
        fscanf(pFile, "%lf%lf", d1, d2);
}

/* Usage: write the output to a specified file
  Input: pFile: a pointer pointing to the output file
          d1 : one double value need to be output
  Output: an updated file into one value written
*/
void writeOneDoubleValue(FILE* pFile,const double d1){
        fprintf(pFile, "%15.12lf", d1);
}

/* Usage: write the output to a specified file
  Input: pFile: a pointer pointing to the output file
          d1 & d2: two double values need to be output
  Output: an updated file into which two values are written
*/
void writeTwoDoubleValue(FILE* pFile,const double d1, const double d2){
        fprintf(pFile, "%15.12lf %15.12lf", d1, d2);
}
```

```
/* Usage: write a 1D double array to a specified file
   Input: pFile: a pointer pointing to the output file
          pDoubleArray: a pointer pointing to the 1D double array
          size: size of the 1D double array
  Output: an updated file with one 1D double array written into
*/
void outputOne1DDoubleArray( FILE *pFile, double *pDoubleArray, int size)
{
        int i;
        for(i = 0; i < size; i++)
                fprintf(pFile, "%15.12f\n", pDoubleArray[i]);
}

/* Usage: write two 1D double arrays with same size to a specified file
   Input: pFile: a pointer pointing to the output file
          pDoubleArr1 & pDoubleArr2: pointers pointing to two 1D double
        arrays
          size: the size of the arrays
   Output: an updated file with two 1D double arrays written into
*/
void outputTwo1DDoubleArrays( FILE *pFile, double *pDoubleArr1, double
*pDoubleArr2, int size){
        int i ;
        for( i = 0 ; i < size; i ++ )
                fprintf(pFile, "%15.12f %15.12f\n", pDoubleArr1[i], pDoubleArr2[i]);
}

/* Usage: write two 1D double arrays and one 2D double array to a specified file
   Input: pFile: a pointer pointing to the output file
pD1DArr1(sizeX), pD1DArr2(sizeY),  pD2DArr(sizeX,sizeY): pointers   pointing
to those arrays
          sizeX & sizeY: sizes of the arrays
   Output: an updated file
*/
void outputThreeArrays(FILE* pFile, double arr1[], double arr2[], double
*arr3[SIZE2], int sizeX, int sizeY){
        int i = 0, j = 0;
        for(i = 0; i <= sizeX; i++){
                for(j = 0; j <= sizeY; j++)
                        fprintf(pFile, "%lf %lf %15.12lf\n", arr1[i], arr2[j], arr3[i][j]);
        }
}

/* Usage: create a 1D double array
   Input: the address of a pointer pointing to a 1D double array and the size
```

```
   Output: if the array is not created correctly, print an error message and exit
*/
void create1DDoubleArray(double **pDoubleArray, int size){
        *pDoubleArray = (double*)malloc( size * sizeof(double));
        if( *pDoubleArray == NULL)
        {
                printf("Error in creating the 1D double array!\n");
                exit(-2);
        }
}

/* Usage: create a 2D double array
   Input: the address of a pointer pointing to a 2D double array
          Number of rows and number of columns
   Output: the 2D array with memory allocated
*/
void create2DDoubleArray(double ***pDoubleArr, int numRows, int  numCols){
        int j = 0;
        // Allocate rows
        (*pDoubleArr) = (double**)malloc(numRows * sizeof(double*));
        if(*pDoubleArr == NULL)
        {
                printf("create 2D double array error!\n");
                exit(-2);
        }
        else
        {
            // Allocate columns
            for( j = 0 ; j < numCols; j ++ )
            {
                (*pDoubleArr)[j] = (double*)malloc( numCols * sizeof( double ) ) ;
                if( (*pDoubleArr)[j] == NULL)
                {
                printf("Create 2D  double array error!\n");
                exit(-2);
                }//if
            }//for
        }//else
}

/* Usage: set initial value of a 1D double array
  Input: the array; the size of the array; and the factor used to set the initial
value
  Output: the updated array with initial value set as a factor * i
*/
```

```
void setA1DDoubleArrayIniValue(double doubleArray[], int size, double factor){
        int i = 0;
        for( i = 0 ; i < size ; i ++ )
                doubleArray[i]  = factor * i ;
}

/* Usage: set initial value of a 2D double array
   Input: the array, number of rows; number of columns; factor used to set the
initial value
   Output: The initialized 2D array with initial value set as a factor * I
*/
void setA2DDoubleArrayIniValue(double *double2Darr[SIZE2], int numRows, int
numCols, double factor ){
        int i = 0, j = 0;
        for( i = 0 ; i < numRows ; i ++ )
                for( j = 0; j < numCols; j ++ )
                        double2Darr[i][j] = i * factor;
}
/* Usage: free1D double array
   Input: the pointer pointing to the array to be freed
   Output: released resource
*/
void free1DDoubleArray(double *pDoubleArray){
        free(pDoubleArray);
}

/* Usage: free 2D double array
   Input: the pointer pointing to the 2D array
           Number of rows
  Output: the released resource
*/
void free2DDoubleArray( double **p2DDoubleArr, int numRows )
{
        int i = 0;
        for( i = 0 ; i < numRows; i ++ )
                free( p2DDoubleArr[i] );
}
```

## B.2. PROGRAM TO COMPUTE INTERFACIAL TEMPERATURE (InterfaceTemp.c)

```
#include <math.h>
#include "BasicOperation.h"

#define Q0 0.
#define QL 8.0
#define SIZE 161

/* Usage: calculate the inner integrand which is a function
   Input: two double parameters which appear in the function
   Output: the inner integrand based on the two inputs
*/
double finner(double s, double q)
{
        double f = 0.;
        f = exp(-s) / (pow(s, 2) + pow((exp(q) - 1), 2));
        return f;
}

/* Usage: calculate the integral by Simpson Integration
   Input: sLower: the low limit; sUpper: the upper limit;
          N: number of intervals; q: a variable occurring in the integrand
   Output: the value of the integral
*/
double Simpson(double sLower, double sUpper, int N, double q)
{
        double stepSize = 0., integral = 0.;
        int i = 0, odd = 1;
        stepSize = (sUpper - sLower) / N;
        integral = finner(sLower, q);
        integral += finner(sUpper, q);
        for(i = 1; i < N; i++)
        {
                integral += finner(sLower + i * stepSize, q) * (odd ? 4 : 2);
                odd = !odd;
        }
        return integral * stepSize / 3.;
}
/* Usage: a driver to calculate the integration
   Input: a pointer pointing to the value of the integral
   Output: the result of the integration stored where the pointer pointed to
*/
void calculateIntegration(double *integral)
```

```
{
	int i;
	double qi = 0., stepSize =0.;
	stepSize = (QL - Q0) / (SIZE - 1);
	integral[0] = 1.;
	for(i = 1; i < SIZE; i ++)
	{
		qi = i * stepSize;
		integral[i] = Simpson(0., 50., 5000, qi);
		integral[i] = 2. / 3.1415 * (exp(qi) - 1) * integral[i];
	}
}


/* Main function: start point of execution*/
void main()
{
	FILE *pFile =NULL;
	int fileIsClose = -1;
	double integral[SIZE] = {0.};
	pFile = openFile("data\\h0_05\\TemperatureP8Q8.dat","w");
	judgeFileOpenCorrectly(pFile,"TemperaturP8Q8.dat");

	calculateIntegration(integral);

	/* Write interface temperature to a file*/
	outputOne1DDoubleArray( pFile, integral, SIZE);
	fileIsClose = closeFile(pFile);
	judgeFileCloseCorrectly(fileIsClose, "TemperatureP8Q8.dat");

	/* Write T(0) and T(ql) to file for using */
	pFile = openFile("data\\h0_05\\T0AndTqlP8Q8.dat","w");
	judgeFileOpenCorrectly(pFile,"T0AndTqlP8Q8.dat");
	writeTwoDoubleValue(pFile,integral[0],integral[SIZE-1]);
	fileIsClose = closeFile(pFile);
	judgeFileCloseCorrectly(fileIsClose, "T0AndTqlP8Q8.dat");
}
```

## B.3. PROGRAM FOR COMPUTING THE POINT FORCE STRENGTH F AND THE DISTANCE BETWEEN IT AND THE WALL (calFAndH.c)

```
#include <math.h>
#include "BasicOperation.h"
#define SIZE 81
#define QL 8.0
#define PI 3.14159

double f2(double);

/* Usage: calculate the inner integrand
   Input: two double variables on which the inner integrand depends
   Output: the value of the inner integrand based on the two inputs
*/
double f11(double x,double q) {

        double valF11 = 0.;

        if(x == 0. && q == 0.)
                valF11 = 1.;
        else
                valF11 = exp(-x) / (pow((exp(q) - 1) ,2) + pow( x, 2 ) );
        return valF11;
}

/* Usage: calculate the double integral by using Simpson Method
   Input: integrateLength: the implicit upper limit of the integration (the implicit
lower limit of the integration is 0)
          choice: an integer indicating if it's for the inner integration(1) or the
outer
          q: a double variable on which the integral depends
*/
double simpsonIntegration(double integrateLength, int choice, double q) {
        double stepSize = 0.1, x[10001], integral =  0., temp1 = 0., temp2 = 0.,
        temp3=0.;
        int numOfIntervals = 0, i = 0;
        numOfIntervals = (int)(integrateLength / stepSize);
        for(i = 0; i <= numOfIntervals; i++)
                x[i] = stepSize * i;
        for(i =0; i < numOfIntervals; i++){
                if(choice == 1){
                    temp1 = f11(x[i], q);
```

```
                temp2 = f11((x[i] + x[i+1]) / 2., q);
                temp3 = f11(x[i+1], q);
                integral += 1. / 6. * (temp1 + 4. * temp2 + temp3) * stepSize;
            }
            else if(choice == 2){
                temp1 = f2(x[i]);
                temp2 = f2((x[i] + x[i+1]) / 2.);
                temp3 = f2(x[i+1]);
                integral += 1. / 6. * (temp1 + 4. * temp2 + temp3) * stepSize;
            }
        }
        return integral;
}

/* Usage: calculate the point force strength
   Input: tQl: temperature at q = ql; t0: temperature at q = 0
   Output: the point force strength
*/
double calculateF(double tQl, double t0){
        double f;
        f = (tQl - t0) / 2 / PI;
        return f;
}

/* Usage: calculate the distance between the point force and wall
   Input: tQl: temperature at q = ql; t0: temperature at q = 0
   Output: the distance between the point force and wall
*/
double calculateH(double tQl, double t0){
        double temp1,temp2,integrate,H;
        temp1 = (exp(QL) - 1.) * tQl;
        integrate = simpsonIntegration(QL, 2, 0.);
        temp2 = 2. / PI * integrate;
        H = (temp1 - temp2) / (tQl - t0);
        printf("temp1 = %lf temp2 = %lf\n", temp1, temp2);
        return H;
}


/* Usage: calculate the outer integrand
   Input: a double variable on which the outer integrand depends
   Output: the value of the outer integrand based on the input
*/
double f2(double q){
        double valF2, integral = 0.;
```

```
	if( q == 0 )
		integral = 1.;
	else
		integral = simpsonIntegration(1000, 1, q);
	valF2 = ( exp(q) - 1 ) * exp(q) * integral;
	return valF2;
}


/* Main Function: starting point of execution*/
void main(){
	FILE * pFile = NULL;
	double tAtQl = 0., tAt0 = 0., H = 0., F = 0.;
	int fileIsClose = -1;

	pFile = openFile("data\\h0_05\\T0AndTqlP8Q8.dat", "r");
	judgeFileOpenCorrectly(pFile, "T0AndTqlP8Q8.dat");
	readTwoDoubleValues(pFile, &tAt0, &tAtQl);
	fileIsClose =  closeFile(pFile);
	judgeFileCloseCorrectly(fileIsClose, "T0AndTqlP8Q8.dat");

	F = calculateF(tAtQl, tAt0);
	H = calculateH(tAtQl, tAt0);

	pFile =openFile("data\\h0_05\\FandHP8Q8.dat", "w");
	judgeFileOpenCorrectly(pFile, "FandHP8Q8.dat");
	writeTwoDoubleValue(pFile, F, H);
	fileIsClose = closeFile(pFile);

}
```

## B.4. PROGRAM FOR COMPUTING BOUNDARY CONDITION ON on $h = h_{¥}$ IN MARANGONI FLOW (rightBound.c)

```
#include <math.h>
#include "BasicOperation.h"
#define QL 8.0
#define SIZE 161

/* Usage: calculate x component of velocity at the right boundary
    Input: p: the array containing the x coordinate
             u: the array used to store the x component velocity
             pasi: the array used to store the stream function along the right
          boundary
             F: the point force strength
             H: the distance between the point force and wall
*/
void calculateUVandPasi(double p[],double u[], double pasi[], double F, double
H){
        int i = 0;
        double temp1 = 0., temp2 = 0., temp3 = 0., x = 0., z = 0.;
        z = exp(QL) - 1.;
        for(i = 1; i < SIZE; i++){
                x = exp(p[i]) - 1.;
                temp1 = pow( x , 2 );
                temp2 = pow( ( z + H ) , 2 );
                temp3 = pow( ( z - H ) , 2 );
                pasi[i] = F * x * ( 2. * H * z / (temp1 + temp2 )
                            - 1. / 2. * log((temp1 +  temp2) / (temp1 + temp3 )) );
                u[i] = -8. * F * pow(H,2) * x * z * ( temp1 -pow(z,2) + pow(H ,2 ) )/
                                  (pow((temp1 + temp2 ), 2 )) / (temp1 + temp3 );
        }
}

/* Main Function: start point for execution*/
void main(){
        FILE *pFile = NULL;
        double *p = NULL, *u = NULL, *pasi = NULL, F = 0., H = 0., stepSize =
        .05;
        int fileIsClose = 0;
        create1DDoubleArray(&p, SIZE);
        create1DDoubleArray(&u, SIZE);
        create1DDoubleArray(&pasi, SIZE);
        setA1DDoubleArrayIniValue(p, SIZE, 0.);
        setA1DDoubleArrayIniValue(u, SIZE, 0.);
```

```
        setA1DDoubleArrayIniValue(pasi, SIZE, 0.);

        pFile = openFile("data\\h0_05\\FandHP8Q8.dat", "r");
        judgeFileOpenCorrectly(pFile, "FandHP8Q8.dat");
        readTwoDoubleValues(pFile, &F, &H);
        fileIsClose = closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose, "FandHP8Q8.dat");
        setA1DDoubleArrayIniValue(p, SIZE, stepSize);

        calculateUVandPasi(p, u, pasi, F, H);

        /* Output u to a file*/
        pFile = openFile("data\\h0_05\\rightUP8Q8.dat", "w");
        judgeFileOpenCorrectly(pFile, "rightUP8Q8.dat");
        outputTwo1DDoubleArrays(pFile, p, u, SIZE);
        fileIsClose =  closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose, "rightUP8Q8.dat");

        /* Output stream function to a file*/
        pFile = openFile("data\\h0_05\\rightBoundaryP8Q8.dat", "w");
        judgeFileOpenCorrectly(pFile,"rightBoundaryP8Q8.dat");
        outputTwo1DDoubleArrays(pFile, p, pasi, SIZE);
        fileIsClose =  closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose, "rightBoundaryP8Q8.dat");

        /* Free arrays*/
        free1DDoubleArray(p);
        free1DDoubleArray(u);
        free1DDoubleArray(pasi);
}
```

## B.5. PROGRAM FOR COMPUTING BOUNDARY CONDITION ON $x = x_{\infty}$ IN MARANGONI FLOW (bottomBound.c)

```
#include <math.h>
#include "BasicOperation.h"
#define PL 8.0
#define SIZE 161

/* Usage: calculate y component of velocity at the bottom boundary
   Input: q: the double array containing the y coordinate
          v: a double array used to store the y component of velocity
          pasi: a double array used to store the stream function along the bottom
       boundary
          F: point force strength
          H: the distance between the point force and the wall
*/
void calculateUVandPasi(double q[],double v[],double pasi[], double F, double
H){
        int i = 0;
        double temp1 = 0., temp2 = 0., temp3 = 0., x = 0., z = 0.;
        x = exp(PL) - 1.;
        for( i = 0 ; i < SIZE ; i++ ){
                z = exp(q[i]) - 1;
                temp1 = pow( x , 2 );
                temp2 = pow( ( z + H ) , 2 );
                temp3 = pow( ( z - H ) , 2 );
                pasi[i] = F * x * ( 2. * H * z / (temp1 + temp2 )
                   - 1. / 2. * log( (temp1 + temp2) / (temp1 + temp3) ) );
                v[i] = -1. * F * (2. * H * z / (temp1 + temp2) - 1. / 2. * log((temp1 +
                temp2) / (temp1 + temp3)));
        }
}



/* Main Function: start execution point*/
void main(){
        FILE *pFile = NULL;
        double *q = NULL, *v = NULL, *pasi = NULL, F = 0., H = 0., stepSize =
        0.05;
        int fileIsClose = 0;

        create1DDoubleArray(&q, SIZE);
        create1DDoubleArray(&v, SIZE);
```

```
        create1DDoubleArray(&pasi, SIZE);

        pFile = openFile("data\\h0_05\\FandHP8Q8.dat", "r");
        judgeFileOpenCorrectly(pFile, "FandHP8Q8.dat");
        readTwoDoubleValues(pFile, &F, &H);
        fileIsClose = closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose, "FandHP8Q8.dat");

        setA1DDoubleArrayIniValue(q, SIZE, stepSize);

        calculateUVandPasi(q, v, pasi, F, H);

        pFile = openFile("data\\h0_05\\bottomVP8Q8.dat", "w");
        judgeFileOpenCorrectly(pFile, "bottomVP8Q8.dat");
        outputTwo1DDoubleArrays(pFile, q, v, SIZE);
        fileIsClose = closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose, "bottomVP8Q8.dat");
        pFile = openFile("data\\h0_05\\bottomBoundaryP8Q8.dat", "w");
        judgeFileOpenCorrectly(pFile, "bottomBoundaryP8Q8.dat");
        outputTwo1DDoubleArrays(pFile, q, pasi, SIZE);
        fileIsClose =  closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose,"bottomBoundaryP8Q8.dat");

        free1DDoubleArray(q);
        free1DDoubleArray(v);
        free1DDoubleArray(pasi);
}
```

## B.6. PROGRAM FOR COMPUTING STREAM FUNCTIONS OF MARANGONI FLOW (calStream.c)

```
#include <math.h>
#include "BasicOperation.h"
#define SIZE1 161
#define SIZE2 161
#define PL 8.0
#define QL 8.0
#define NP 160
#define NQ 160

/* Usage: calculate the step size
   Input: nothing
   Output: the step size
*/
double calStepSize(){
        return PL / NP;
}


/* Usage: read stream function from a specified file
    Input: pFile: the pointer pointing to the file
            p & q: 1D double arrays used to store x & ycoordinates
            pasi: 2D double array used to store the stream functions
                        nx & ny: number of rows and cols for pasi
    Output: updated arrays
*/
void readStreamFunctionFromFile(FILE* pFile, double p[], double q[], double
*pasi[SIZE2], int nx, int ny){
        int i = 0, j = 0;
        for(i=0;  i <= nx; i++)
                for(j = 0; j <= ny; j++)
                    fscanf(pFile, "%lf %lf %lf", &p[i], &q[j], &pasi[i][j]);
}

/* Usage: interpolate stream functions got when step size h = 0.1
    Input: pasi: used to store stream functions interpolated
        TempPasi: used to store stream function to interpolate
    Output: updated stream functions stored in pasi
*/
void interpolateStreamFunction( double* pasi[SIZE2], double
*tempPasi[SIZE2/2]){
        int i = 0, j = 0;
        for(i = 0; i < SIZE1; i += 2)
```

```
			for(j = 0; j < SIZE2; j += 2)
				pasi[i][j] = tempPasi[i / 2][j / 2];
		for(i = 1; i < SIZE1 - 1; i += 2)
			for(j = 0; j < SIZE2; j += 2)
				pasi[i][j] = (tempPasi[(i - 1) / 2][j / 2] +
					tempPasi[(i + 1) / 2][j / 2])/ 2.;
		for(i = 0; i < SIZE1; i++)
			for(j = 1; j < SIZE2 - 1; j += 2)
				pasi[i][j] = (pasi[i][j - 1] + pasi[i][j + 1]) / 2.;
}

/* Usage: read top boundary conditions
   Input: pFile: a pointer pointing to the file to be read
          q: a 1D double array to store y coordinate
          T: a 1D double array to store the temperature at the top boundary
   Output: the updated T array
*/
void readTopBoundary(FILE *pFile, double q[], double T[]){
		int j = 0;
		for( j = 0 ; j < SIZE2;  j++ )
			readTwoDoubleValues(pFile, &q[j], &T[j]);
}

/* Usage: read right boundary conditions
   Input: pFile: a pointer pointing to the file to be read
          p: 1D double array to store x coordinate
          pasi: 2D double array to store stream functions
   Output: updated pasi array
*/
void readRightBoundary(FILE* pFile, double p[], double *pasi[SIZE2]){
		int i = 0, j = 0;
		for( i = 0 ; i <= NP; i ++ )
			readTwoDoubleValues(pFile, &p[i], &pasi[i][NQ] );
}

/* Usage: read x component of velocity on the right boundary
   Input: pFile: a pointer pointing to the file to be read
          p: 1D double array to store x coordinate
          u: 1D double array to store x component velocity on the right-bound
   Output: updated u array
*/
void readRightU(FILE* pFile, double p[], double u[]){
		int i = 0;
		for( i = 0 ; i <= NP; i ++ )
			readOneDoubleValue(pFile, &u[i]);
```

```
}

/* Usage: read bottom boundary conditions
   Input: pFile: a pointer pointing to the file to be read
          q: 1D double array to store y coordinate
          pasi: 2D double array to store stream functions
   Output: updated pasi array
*/
void readBottomBoundary(FILE* pFile, double q[], double *pasi[SIZE2]){
        int i = 0, j = 0;
        for( i = 0 ; i <= NP; i ++ )
                readTwoDoubleValues(pFile, &q[j], &pasi[NP][j] );
}

/* Usage: read y component of velocity on the bottom boundary
   Input: pFile: a pointer pointing to the file to be read
          q: 1D double array to store x coordinate
          v: 1D double array to store x component velocity on the right-bound
   Output: updated u array
*/
void readBottomV(FILE* pFile, double q[], double v[]){
        int j = 0;
        for( j = 0 ; j <= NQ; j ++ )
                readOneDoubleValue(pFile, &v[j]);
}

/* Usage: set the initial value of the stream function on inner points
   Input: pasi: 2D double array for store stream functions
          nx & ny: x & y dimensions of array pasi
   Output: the updated array pasi
*/
void setIniStreamOnInnerPoints(double *pPasi[SIZE2], int nx, int ny){
        int i = 0, j = 0;
        for( i = 1 ; i < nx ; i++)
                for( j = 1 ; j < ny ; j++ )
                        pPasi[i][j] = pPasi[nx][j] / (nx) * i;
}

/* Usage: set pasi0 as a reference of pasi
   Input: two 2D double arrays: pasi and pasi0
          nx & ny are x & y dimensions of the 2D arrays
   Output: updated pasi0
*/
void setPasi0(double *pPasi[SIZE2], double *pPasi0[SIZE2], int nx, int ny){
        int i = 0, j = 0;
```

```
        for(i = 0; i <= nx; i++)
                for(j = 0; j <= ny; j++)
                        pPasi0[i][j] = pPasi[i][j];
}

/* Usage: calculate the coefficient on (i, j)
   Input: step size and x & y coordinate
   Output: coefficient on (i, j)
*/
double calculateCoefIAndJ(double stepSize, double p, double q){
        double temp1, temp2, temp3, temp4, coefOnIJ;
        temp1 = exp(-4. * p);
        temp2 = exp(-4. * q);
        temp3 = (temp1 + temp2) / pow(stepSize,4) * (6.-22.*pow(stepSize,2));
        temp1 = exp(-2. * p - 2. * q);
        temp4 = temp1 * 8. / pow(stepSize,4);
        coefOnIJ = temp3 + temp4;
        return coefOnIJ;
}

/* Usage: calculate the coefficient on (i+1, j)
   Input: step size and x & y coordinates
   Output: coefficient on (i+1, j)
*/
double calculateCoefIPlus1J(double stepSize, double p, double q){
        double temp1, temp2, temp3, temp4, coefOnIPlus1J;
        temp1 = exp( -4. * p);
        temp2 = (-3. * pow(stepSize,3) + 11. * pow(stepSize,2) + 6. * stepSize -
        4.) / pow(stepSize,4);
        temp3 = temp1 * temp2;
        temp1 = exp(-2. * p - 2. * q);
        temp2 = 2. * (stepSize - 2.) / pow(stepSize,4);
        temp4 = temp1 * temp2;
        coefOnIPlus1J = temp3 + temp4;
        return coefOnIPlus1J;
}

/* Usage: calculate the coefficient on (i-1, j)
   Input: step size and x & y coordinates
   Output: coefficient on point (i-1, j)
*/
double calculateCoefIMinus1J(double stepSize, double p,  double q){
        double temp1, temp2, temp3, temp4, coefOnIMinus1J;
        temp1 = exp( -4. * p);
```

```
        temp2 = (3. * pow(stepSize,3) + 11. * pow(stepSize,2) - 6. * stepSize -
        4.) / pow(stepSize,4);
        temp3 = temp1 * temp2;
        temp1 = exp(-2. * p - 2. * q);
        temp2 = (2. + stepSize) * 2. / pow(stepSize,4);
        temp4 = temp1 * temp2;
        coefOnIMinus1J = temp3 - temp4;
        return coefOnIMinus1J;
}

/* Usage: calculate the coefficient on (i, j+1)
   Input: step size and x & y coordinates
   Output: coefficient on (i, j +1)
*/
double calculateCoefIJPlus1(double stepSize, double p, double q){
        double temp1, temp2, temp3, temp4, coefOnIJPlus1;
        temp1 = exp( -4. * q);
        temp2 = (-3. * pow(stepSize,3) + 11. * pow(stepSize,2) + 6. * stepSize -
        4.) / pow(stepSize,4);
        temp3 = temp1 * temp2;
        temp1 = exp(-2. * p - 2. * q);
        temp2 = 2. * (stepSize - 2.) / pow(stepSize,4);
        temp4 = temp1 * temp2;
        coefOnIJPlus1 = temp3 + temp4;
        return coefOnIJPlus1;
}

/* Usage: calculate the coefficient on (i, j-1)
   Input: step size and x & y coordinates
   Output: coefficient on (i, j -1)
*/
double calculateCoefIJMinus1(double stepSize, double p, double q){
        double temp1, temp2, temp3, temp4, coefOnIJMinus1;
        temp1 = exp( -4. * q);
        temp2 = (3. * pow(stepSize,3) + 11. * pow(stepSize,2) - 6. * stepSize -
        4.) / pow(stepSize,4);
        temp3 = temp1 *temp2;
        temp1 = exp(-2. * p - 2. * q);
        temp2 = 2. * (2. + stepSize) / pow(stepSize,4);
        temp4 = temp1 * temp2;
        coefOnIJMinus1 = temp3 - temp4;
        return coefOnIJMinus1;
}

/* Usage: calculate the coefficient on (i+2, j)
```

```
   Input: step size and x coordinate
   Output: coefficient on (i + 2, j)
*/
double calculateCoefIPlus2J(double stepSize, double p){
        double temp1, temp2, coefOnIPlus2J;
        temp1 = exp(-4. * p);
        temp2 = (1. - 3. * stepSize) / pow(stepSize,4);
        coefOnIPlus2J = temp1 * temp2;
        return coefOnIPlus2J;
}

/* Usage: calculate the coefficient on (i-2, j)
   Input: step size and x coordinate
   Output: coefficient on (i-2, j)
*/
double calculateCoefIMinus2J(double stepSize, double p){
        double temp1, temp2, coefOnIMinus2J;
        temp1 = exp(-4. * p);
        temp2 = (1. + 3. * stepSize) / pow(stepSize,4);
        coefOnIMinus2J = temp1 * temp2;
        return coefOnIMinus2J;
}

/* Usage: calculate the coefficient on (i, j+2)
   Input: step size and the y coordinate
   Output: the coefficient on (i, j + 2)
*/
double calculateCoefIJPlus2(double stepSize, double q){
        double temp1, temp2, coefOnIJPlus2;
        temp1 = exp(-4. * q);
        temp2 = (1. - 3. * stepSize) / pow(stepSize,4);
        coefOnIJPlus2 = temp1 * temp2;
        return coefOnIJPlus2;
}

/* Usage: calculate the coefficient on (i, j-2)
   Input: step size and the y coordinate
   Output: the coefficient on (i, j - 2)
*/
double calculateCoefIJMinus2(double stepSize, double q){
        double temp1, temp2, coefOnIJMinus2;
        temp1 = exp(-4. * q);
        temp2 = (1. + 3. * stepSize) / pow(stepSize,4);
        coefOnIJMinus2 = temp1 * temp2;
        return coefOnIJMinus2;
```

```
}

/* Usage: calculate the coefficient on (i+1, j+1)
   Input: step size, x & y coordinates
   Output: the coefficient on (i+1, j+1)
*/
double calculateCoefIPlus1JPlus1(double stepSize, double p, double q){
        double temp1, temp2, coefOnIPlus1JPlus1;
        temp1 = exp(-2. * p - 2. * q);
        temp2 = 2. * (1. - stepSize + pow(stepSize,2) / 4.) / pow(stepSize,4);
        coefOnIPlus1JPlus1 = temp1 * temp2;
        return coefOnIPlus1JPlus1;
}

/* Usage: calculate the coefficient on (i+1, j  -1)
   Input: step size and x & y coordinates
   Output: the coefficient on (i + 1, j - 1 )
*/
double calculateCoefIPlus1JMinus1(double stepSize, double p, double q){
        double temp1, temp2, coefOnIPlus1JMinus1;
        temp1 = exp( -2. * p - 2. * q);
        temp2 = 2. * (1. - pow(stepSize,2) / 4.) / pow(stepSize,4);
        coefOnIPlus1JMinus1 = temp1 * temp2;
        return coefOnIPlus1JMinus1;
}

/* Usage: calculate the coefficient on (i-1, j+1)
   Input: step size and x & y coordinates
   Output: the coefficient on (i - 1, j + 1)
*/
double calculateCoefIMinus1JPlus1(double stepSize, double p, double q){
        double temp1, temp2, coefOnIMinus1JPlus1;
        temp1 = exp( -2. * p - 2. * q);
        temp2 = 2. * (1. - pow(stepSize,2) / 4.) / pow(stepSize,4);
        coefOnIMinus1JPlus1 = temp1 * temp2;
        return coefOnIMinus1JPlus1;
}

/* Usage: calculate the coefficient on the (i-1,j-1)
   Input: step size, x & y coordinates
   Output: coefficient on (i - 1, j - 1)
*/
double calculateCoefIMinus1JMinus1(double stepSize, double p, double q){
        double temp1, temp2, coefOnIMinus1JMinus1;
        temp1 = exp(-2. * p - 2. * q);
```

```
        temp2 = 2. * (1. + stepSize + pow(stepSize,2) / 4.) / pow(stepSize,4);coefOnIMinus1JMin
        return coefOnIMinus1JMinus1;
}

/* Usage: calculate coefficients in the descretized equation
   Input: addresses of all the coefficients need to be calculated
          Step size and x and y coordinates on the point
*/
void calculateCoef(double *coefOnIJ, double *coefOnIPlus1J, double
*coefOnIMinus1J, double *coefOnIJPlus1, double* coefOnIJMinus1, double*
coefOnIPlus2J, double* coefOnIMinus2J, double* coefOnIJPlus2, double*
coefOnIJMinus2, double* coefOnIPlus1JPlus1, double *coefOnIPlus1JMinus1,
double* coefOnIMinus1JPlus1, double *coefOnIMinus1JMinus1, double
stepSize, double p, double q){
        *coefOnIJ = calculateCoefIAndJ(stepSize, p, q);
        *coefOnIPlus1J = calculateCoefIPlus1J(stepSize, p, q);
        *coefOnIMinus1J = calculateCoefIMinus1J(stepSize, p, q);
        *coefOnIJPlus1 = calculateCoefIJPlus1(stepSize, p, q);
        *coefOnIJMinus1 = calculateCoefIJMinus1(stepSize, p, q);
        *coefOnIPlus2J = calculateCoefIPlus2J(stepSize, p);
        *coefOnIMinus2J = calculateCoefIMinus2J(stepSize, p);
        *coefOnIJPlus2 = calculateCoefIJPlus2(stepSize, q);
        *coefOnIJMinus2 = calculateCoefIJMinus2(stepSize, q);
        *coefOnIPlus1JPlus1 = calculateCoefIPlus1JPlus1(stepSize, p, q);
        *coefOnIPlus1JMinus1 = calculateCoefIPlus1JMinus1(stepSize, p, q);
        *coefOnIMinus1JPlus1 = calculateCoefIMinus1JPlus1(stepSize, p, q);
        *coefOnIMinus1JMinus1 =calculateCoefIMinus1JMinus1(stepSize, p,q);
}

/* Usage: calculate coef * pasi (usual expressions)
   Input: coefficient on that point and the pasi array, i & j index
   Output: part1 ~ part 12
*/
void calculateParts(double*part1, double* part2, double* part3, double* part4,
double *part5, double *part6, double *part7, double *part8, double *part9,
double *part10, double *part11, double *part12, double coefOnIPlus1J, double
coefOnIMinus1J, double coefOnIJPlus1, double coefOnIJMinus1, double
coefOnIPlus2J, double coefOnIMinus2J, double coefOnIJPlus2, double
coefOnIJMinus2, double coefOnIPlus1JPlus1, double coefOnIPlus1JMinus1,
double coefOnIMinus1JPlus1, double coefOnIMinus1JMinus1, double
*pPasi[SIZE2], int i, int j){
        *part1 = coefOnIPlus1J * pPasi[i+1][j];
        *part2 = coefOnIMinus1J * pPasi[i-1][j];
        *part3 = coefOnIJPlus1 * pPasi[i][j+1];
        *part4 = coefOnIJMinus1 * pPasi[i][j-1];
```

```
        if( i != NP - 1 )
           *part5 = coefOnIPlus2J * pPasi[i+2][j];
        if( i != 1)
           *part6 = coefOnIMinus2J * pPasi[i-2][j];
        if( j != NQ - 1)
           *part7 = coefOnIJPlus2 * pPasi[i][j+2];
        if( j != 1)*
           *part8 = coefOnIJMinus2 * pPasi[i][j-2];
        *part9 = coefOnIPlus1JPlus1 * pPasi[i+1][j+1];
        *part10 = coefOnIPlus1JMinus1* pPasi[i+1][j-1];
        *part11 = coefOnIMinus1JPlus1* pPasi[i-1][j+1];
        *part12 = coefOnIMinus1JMinus1 * pPasi[i-1][j-1];
}

/* Usage: calculate the sum of those 12 parts
    Input: twelve double variables
   Output: the sum
*/
double sum12Parts(double part1, double part2, double part3, double part4,
double part5, double part6, double part7, double part8, double part9, double
part10, double part11, double part12){
        double sum=0.;
        sum=part1+part2+part3+part4+part5+part6+part7+part8+part9+part10+p
        art11+part12;
        return sum;
}

/* Usage: calculate stream function on (1,1)
   Input: pasi: 2D double array used to store stream function
           stepSize: the step size
           p & q: x & y coordinate on that point
           i & j: x & y index on that point
   Output: pasi[1][1]
*/
void calculateStreamFunctionOn11(double *pPasi[SIZE2], double stepSize,
double p, double q, int i, int j, double T[]){
        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp1, temp2, temp3, part1,
part2, part3, part4, part5, part6, part7, part8, part9, part10, part11, part12;

        //calculate coefficients
        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
```

```
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1,  &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);

        temp1 = coefOnIJ + coefOnIMinus2J * (stepSize - 2.) / (stepSize + 2. ) +
        coefOnIJMinus2;
        temp2 = -1. / temp1;
        temp3 = temp2;

        //calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j)                              ;

        //calculate special parts
        part6 = coefOnIMinus2J * (stepSize / (2. + stepSize) * exp(-q) * (T[j+1] -
        T[j-1]));
        part8 = 0.;

        pPasi[i][j] = temp3 * sum12Parts(part1, part2, part3, part4, part5, part6,
        part7, part8, part9, part10, part11, part12);
}

/* Usage: calculate stream function on (1, NQ - 1)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[1][NQ - 1]
*/
void calculateStreamFunctionOn1NqMinus1(double *pPasi[SIZE2], double
stepSize, double ql, double u, double p, double q, int i, int j,double T[]){
        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp1, temp2, temp3,
        part1, part2, part3, part4, part5, part6, part7, part8, part9, part10, part11,
        part12;

        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
```

```
        &coefOnIPlus1JMinus1,  &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);

        temp1=(-2.+stepSize)/(2.+stepSize)*coefOnIMinus2J+coefOnIJPlus2;
        temp2=coefOnIJ+temp1;
        temp3=-1./temp2;

        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);

        // Special parts
        part6 = coefOnIMinus2J * (stepSize / (2. + stepSize) * exp(-q) * (T[j+1] -
        T[j-1]));
        part7 = coefOnIJPlus2 * (2. * stepSize * exp(ql) * u);

        pPasi[i][j] = temp3 * sum12Parts(part1, part2, part3, part4, part5, part6,
        part7, part8, part9, part10, part11, part12);

}

/* Usage: calculate stream function on (NP - 1, 1)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[NP-1][1]
*/
void calculateStreamFunctionOnNpMinus11(double *pPasi[SIZE2], double
stepSize, double pl, double v, double p, double q, int i, int j){

        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp1, temp2, temp3,
        part1, part2, part3, part4, part5, part6, part7, part8, part9, part10, part11,
        part12;


        calculateCoef(&coefOnIJ,&coefOnIPlus1J,&coefOnIMinus1J,&coefOnIJP
        lus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
```

```
        &coefOnIPlus1JMinus1, &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);

        temp1 = coefOnIPlus2J + coefOnIJMinus2 + coefOnIJ;
        temp2 = -1./temp1;
        temp3 = temp2;

        //calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);

        //calculate special parts
        part5 = coefOnIPlus2J * (-2. * stepSize * exp(pl) * v);
        part8 = 0.;

        pPasi[i][j] = temp3 * sum12Parts(part1, part2, part3, part4, part5, part6,
        part7, part8, part9, part10, part11, part12);

}

/* Usage: calculate stream function on (NP - 1, NQ - 1)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[NP-1][NQ - 1]
*/
void calculateStreamFunctionOnNpMinus1NqMinus1(double *pPasi[SIZE2],
double stepSize, double u, double v, double pl, double ql, double p, double
q,const int i,const int j){
        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp1, temp2, temp3,
        part1, part2, part3, part4, part5, part6, part7, part8, part9, part10, part11,
        part12;

        //calculate those coefficients
        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
```

```
        &coefOnIPlus1JMinus1, &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);

        temp1 = coefOnIJ + coefOnIPlus2J + coefOnIJPlus2;
        temp2 = -1./temp1;
        temp3 = temp2;

        //calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);

        //calculate special parts
        part5 = coefOnIPlus2J * (-2. * stepSize * exp(pl) * v);
        part7 = coefOnIJPlus2 * (2. * stepSize * exp(ql) * u);

        pPasi[i][j] = temp3 * sum12Parts(part1, part2, part3, part4, part5, part6,
        part7, part8, part9, part10, part11, part12);
}

/* Usage: calculate stream function on (1, j)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[1][j]
*/
void calculateStreamFunctionOn1J(double *pPasi[SIZE2], double stepSize,
double p, double q, int i, int j, double T[]){
        double coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, coefOnIJ, temp1, temp2,
        temp3, part1, part2, part3, part4, part5, part6, part7, part8, part9, part10,
        part11, part12;

        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1, &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);
```

```
        temp1 = (-2. + stepSize) / (2. + stepSize) * coefOnIMinus2J;
        temp2 = coefOnIJ + temp1;
        temp3 = -1. / temp2;

        //calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);

        //calculate special parts
        part6 = coefOnIMinus2J * (stepSize / (2. + stepSize) * exp(-q) * (T[j+1] -
        T[j-1]));

        pPasi[i][j] = temp3 * sum12Parts(part1, part2, part3, part4, part5, part6,
        part7, part8, part9, part10, part11, part12);
}

/* Usage: calculate stream function on (NP - 1, j)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[NP - 1][j]
*/
void calculateStreamFunctionOnNpMinus1J(double *pPasi[SIZE2], double
stepSize, double pl, double v, double p, double q, int i, int j){
        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp1, temp2, temp3,
        part1, part2, part3, part4, part5, part6, part7, part8, part9, part10, part11,
        part12;

        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1, &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);

        temp1 = coefOnIPlus2J / coefOnIJ;
        temp2 = 1 + temp1;
        temp3 = 1. / temp2 / (-coefOnIJ);
```

```
        // calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);

        // calculate special parts
        part5 = coefOnIPlus2J * (-2. * stepSize * exp(pl) * v);

        pPasi[i][j] = temp3 * sum12Parts(part1, part2, part3, part4, part5, part6,
        part7, part8, part9, part10, part11, part12);
}

/* Usage: calculate stream function on (i, 1)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[i][1]
*/
void calculateStreamFunctionOnI1(double *pPasi[SIZE2], double stepSize,
double p, double q, int i, int j){
        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp1, temp2, temp3,
        part1, part2, part3, part4, part5, part6, part7, part8, part9, part10, part11,
        part12;

        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1, &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize,p,q);

        temp1 = coefOnIJ + coefOnIJMinus2;
        temp2 = -1. / temp1;
        temp3 = temp2;

        //calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
```

```
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);

        // calculate special part
        part8 = 0.;

        pPasi[i][j] = temp3 * sum12Parts(part1, part2, part3, part4, part5, part6,
        part7, part8, part9, part10, part11, part12);
}

/* Usage: calculate stream function on (i, NQ - 1)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[i][NQ - 1]
*/
void calculateStreamFunctionOnINqMinus1(double *pPasi[SIZE2], double
stepSize, double ql, double u, double p, double q, int i, int j){

        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp1, temp2, temp3,
        part1, part2, part3, part4, part5, part6, part7, part8, part9, part10, part11,
        part12;

        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1, &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);

        temp1 = coefOnIJ + coefOnIJPlus2;
        temp2 = -1. / temp1;
        temp3 = temp2;

        //calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
```

```
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);

        //calculate special parts
        part7 = coefOnIJPlus2 * 2. * stepSize * exp(ql) * u;
        pPasi[i][j] = temp3 * sum12Parts(part1, part2, part3, part4, part5, part6,
        part7, part8, part9, part10, part11, part12);
}

/* Usage: calculate stream function on (i, j)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[i][j]
*/
void calculateStreamFunctionOnIJ(double *pPasi[SIZE2], double stepSize,
double p, double q, int i, int j){

        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp3, part1, part2,
        part3, part4, part5, part6, part7, part8, part9, part10, part11, part12;

        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1, &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);

        temp3 = -1. / coefOnIJ;

        //calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);

        pPasi[i][j] = temp3 * sum12Parts(part1, part2, part3, part4, part5, part6,
        part7, part8, part9, part10, part11, part12);
}
```

```
/* Usage: calculate the maximum error
   Input: the array pasi, the reference array pasi0, number of rows & columns of
the 2D array, and the addresses of two integers storing the i-index and j-index
of the point with largest difference between two iteration*/
double calculateError(double *pPasi[SIZE2], double *pPasi0[SIZE2], int* pi, int
*pj){
        int i = 0, j = 0;
        double error = 0.;
        error = fabs(pPasi[1][1] - pPasi0[1][1]);
        for(i = 1; i < NP; i++)
                for(j = 1; j < NQ; j++)
                        if(error < fabs(pPasi[i][j] - pPasi0[i][j]))
                        {
                                error = fabs(pPasi[i][j] - pPasi0[i][j]);
                                *pi = i;
                                *pj = j;
                        }
        return error;
}

/* Usage: iterate to get stream functions satisfying accuracy requirement
   Input: pasi: the 2D double array used to store the stream function
          pasi0: the reference of pasi
          stepSize: the step size
          p & q: 1D double arrays used to store x & y coordinates
          T: 1D double array used to store temperature at the interface
   Output: the updated pasi array
*/
void iterateForStream(double *pasi[SIZE2], double *pasi0[SIZE2], double p[],
double q[], double T[], double u[], double v[], double stepSize, double
accuracyControl, double relax){
        int i = 0, j = 0, erri = 1, errj = 1, numOfIteration = 0;
        double error = 0.;
        do{
                i = 1; j = 1;
                calculateStreamFunctionOn11(pasi, stepSize, p[i], q[j], i, j, T);

                i = 1; j = NQ - 1;
                calculateStreamFunctionOn1NqMinus1(pasi,stepSize,QL,u[i],
                p[i],q[j],i,j,T);

                i = NP - 1; j = 1;
                calculateStreamFunctionOnNpMinus11(pasi,stepSize,PL,v[j],p[i],
                q[j],i,j);
```

```
			i = NP - 1; j = NQ - 1;
			calculateStreamFunctionOnNpMinus1NqMinus1(pasi,stepSize,u[i]
			, v[j], PL, QL, p[i], q[j], i, j);

			for(i = 1, j = NQ - 2; j >= 2; j--)
				calculateStreamFunctionOn1J(pasi, stepSize, p[i], q[j], i, j,
				T);
			for(i = NP - 1, j = NQ - 2; j >= 2; j--)
				calculateStreamFunctionOnNpMinus1J(pasi, stepSize, PL,
				v[j], p[i], q[j], i, j);
			for(j = 1, i = NP - 2; i >= 2; i--)
				calculateStreamFunctionOnI1(pasi,stepSize,p[i],q[j],i,j);
			for(j = NQ -1, i = NP - 2; i >= 2; i--)
				calculateStreamFunctionOnINqMinus1(pasi, stepSize, QL,
				u[i], p[i], q[j], i, j);
			for( j = NQ - 2 ; j >= 2; j--)
			{
				for( i = NP -2 ; i >= 2 ; i--)
					calculateStreamFunctionOnIJ(pasi,stepSize,p[i],q[j],i,
					j);
			}

			// calculate error
			error = calculateError(pasi, pasi0, &erri, &errj);
			if(fabs(error) < fabs(accuracyControl)) break;
			++ numOfIteration;
			for(i = 1; i < NP; i++)
				for(j = 1; j < NQ; j++)
					pasi[i][j] = pasi0[i][j] + relax * (pasi[i][j] - pasi0[i][j]);
			setPasi0(pasi, pasi0, NP, NQ);
			if(numOfIteration%10 == 0)
			{
				printf("i = %d j = %d\n", erri, errj);
				printf("%d  err = %15.14lf pasi = %lf\n", numOfIteration,
				error, pasi[erri][errj]);
			}
		}while(error>accuracyControl);
}

/* Main Function: start execution point*/
void main(){
		double *p = NULL, *q = NULL, **pasi = NULL, *T = NULL, *u = NULL, *v
		= NULL, **pasi0 = NULL, stepSize = 0.,  error = 0., accuracyControl =
		1.e-9, relax = 1.1, *tempP = NULL, *tempQ = NULL, **tempPasi = NULL;
		FILE* pFile = NULL;
```

```
int i = 0, j = 0, numOfIteration = 0,  fileIsClose = 0;
// determine the stepSize,NP,NQ
stepSize = calStepSize();

// Create 1D double arrays
create1DDoubleArray( &p, SIZE1 );
create1DDoubleArray( &q, SIZE2 );
create1DDoubleArray( &u, SIZE1 );
create1DDoubleArray( &v, SIZE2 );
create1DDoubleArray( &T, SIZE2 );
create1DDoubleArray( &tempP, SIZE1 );
create1DDoubleArray( &tempQ, SIZE2 );

// Create 2D double arrays
create2DDoubleArray( &pasi, SIZE1, SIZE2 );
create2DDoubleArray( &pasi0, SIZE1, SIZE2 );
create2DDoubleArray( &tempPasi, SIZE1, SIZE2 );

// Set initial values for 1D arrays
setA1DDoubleArrayIniValue(p, SIZE1, stepSize);
setA1DDoubleArrayIniValue(q, SIZE2, stepSize );
setA1DDoubleArrayIniValue(u, SIZE1, 0. );
setA1DDoubleArrayIniValue(v, SIZE2, 0. );
setA1DDoubleArrayIniValue(T, SIZE2, 0. );
setA1DDoubleArrayIniValue(tempP, SIZE1, 0. );
setA1DDoubleArrayIniValue(tempQ, SIZE2, 0. );

// Initialize pasi, pasi0
setA2DDoubleArrayIniValue(pasi, SIZE1, SIZE2, 0. );
setA2DDoubleArrayIniValue(pasi0, SIZE1, SIZE2, 0. );
setA2DDoubleArrayIniValue(tempPasi, SIZE1, SIZE2, 0. );

/* Read stream function from file with previous iteration result
   This section restricted by '/*' & '*/ is used after at least one iteration
*/
/*pFile = openFile("data\\h0_05\\streamP8Q81.dat", "r");
judgeFileOpenCorrectly(pFile, "streamP8Q81.dat");
readStreamFunctionFromFile(pFile, p, q, pasi, NP, NQ);
closeFile(pFile);
fileIsClose = closeFile(pFile)
judgeFileCloseCorrectly(fileIsClose, "streamP8Q81.dat")*/

/* Read initial stream function for stepSize h = 0.1
   Use this to interpolate to get initial stream function for stepSize h =.05
   This section will be used only for the first iteration
```

```
*/
pFile = openFile("data\\h0_05\\streamP8Q8_h0_1.dat", "r");
judgeFileOpenCorrectly(pFile, "streamP8Q8h0_1.dat");
readStreamFunctionFromFile(pFile, tempP, tempQ, tempPasi, NP/2,
NQ/2);
fileIsClose = closeFile(pFile);
judgeFileCloseCorrectly(fileIsClose, "streamP8Q8h0_1.dat");

/* Interpolate the stream function */
interpolateStreamFunction(pasi, tempPasi);

/* free tempP, tempQ, and tempPasi*/
free1DDoubleArray(tempP);
free1DDoubleArray(tempQ);
free2DDoubleArray(tempPasi, SIZE1);

/* open file for reading boundary conditions of the stream functions*/
// Top boundary
pFile = openFile("data\\h0_05\\TemperatureP8Q8.dat", "r");
judgeFileOpenCorrectly(pFile, "TemperatureP8Q8.dat");
readTopBoundary(pFile, q, T);
fileIsClose = closeFile(pFile);
judgeFileCloseCorrectly(fileIsClose, "TemperatureP8Q8.dat");

// Right boundary
pFile = openFile("data\\h0_05\\rightBoundaryP8Q8.dat", "r");
judgeFileOpenCorrectly(pFile, "rightBoundaryP8Q8.dat");
readRightBoundary(pFile, p, pasi);
fileIsClose = closeFile(pFile);
judgeFileCloseCorrectly(fileIsClose, "rightBoundaryP8Q8.dat");
pFile = openFile("data\\h0_05\\rightUP8Q8.dat", "r");
judgeFileOpenCorrectly(pFile, "rightUP8Q8.dat");
readRightU(pFile, p, u);
fileIsClose = closeFile(pFile);
judgeFileCloseCorrectly(fileIsClose, "rightUP8Q8.dat");

// Bottom boundary
pFile = openFile("data\\h0_05\\bottomBoundaryP8Q8.dat", "r");
judgeFileOpenCorrectly(pFile, "bottomBoundaryP8Q8.dat");
readBottomBoundary(pFile, q, pasi);
fileIsClose = closeFile(pFile);
judgeFileCloseCorrectly(fileIsClose, "bottomBoundaryP8Q8.dat");
pFile = openFile("data\\h0_05\\bottomVP8Q8.dat", "r");
judgeFileOpenCorrectly(pFile, "bottomVP8Q8.dat");
readBottomV(pFile, q, v);
```

```
        fileIsClose = closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose, "bottomVP8Q8.dat");

        /* Set the initial stream functions for inner points
           This is only for the first iteration and the largest step size, say h = .1
        */
        /* setIniStreamOnInnerPoints(pasi, NP, NQ);
        */

        /*Set pasi0: pasi0 is used as the reference for pasi to check accuracy */
        setPasi0(pasi, pasi0, NP, NQ);

        /* Start Iteraton */
        iterateForStream(pasi, pasi0,  p, q, T, u, v, stepSize, accuracyControl,
        relax);

        // open file to write pasi
        pFile = openFile("data\\h0_05\\StreamP8Q8.dat", "w");
        judgeFileOpenCorrectly(pFile, "StreamP8Q8.dat");
        outputThreeArrays(pFile, p, q, pasi, NP, NQ);
        fileIsClose = closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose, "StreamP8Q8.dat");

        // Free memeory
        free1DDoubleArray( p );
        free1DDoubleArray( q );
        free1DDoubleArray( u );
        free1DDoubleArray( v );
        free1DDoubleArray( T );

        free2DDoubleArray( pasi, SIZE1 );
        free2DDoubleArray( pasi0, SIZE1 );
}
```

## B.7. PROGRAM FOR COMPUTING BOUNDARY CONDITION AT THE LIQUID-VAPOR INTERFACE IN SINK FLOW (calTop.c)

```
#include <math.h>
#include "BasicOperation.h"

#define Q0 0.
#define QL 8.0
#define SIZE 161
#define GAMA .5772156649

/* Usage: calculate the inner integrand which is a function
   Input: 1. two double variables on which the integrand depends
   Output: the value of the integrand
*/
double finner(double s, double yi)
{
        double f = 0.;
        f = log(pow(s,2) + pow(yi,2)) * exp(-s);
        return f;
}

/* Usage: calculate an integration by Simpson Method
   Input: the upper & lower limit of the integration, a double variable on which
the integrand depends
   Output: the result of the integratioin
*/
double Simpson(double sLower, double sUpper, double stepSize, double yi)
{
        double integral = 0.;
        int i=0, odd = 1 , N = 0;
        N = (int)((sUpper - sLower) / stepSize);
        integral = finner(sLower, yi);
        integral += finner(sUpper, yi);
        for(i = 1; i < N; i++)
        {
                integral += finner(sLower + i * stepSize, yi) * (odd ?  4 : 2);
                odd = !odd;
        }
        return integral * stepSize / 3.;

}


/* Usage: a driver to calculate the integration
```

```
    Input: a double array used to store the value of the integration
    Output: the updated array with integration value
*/
void calculateIntegration(double *integral)
{
        int i = 0;
        double qi = 0., stepSize = 0.;
        stepSize = (QL - Q0) / (SIZE - 1);
        integral[0] = 0.;
        for(i = 1; i < SIZE; i++)
        {
                qi = i * stepSize;
                integral[i] = Simpson(0., 100., 0.001, exp(qi)-1.);
                integral[i] = -1./3.1415 * integral[i] + 2 / 3.1415 * (-GAMA);
        }
}

/* Main Function: start point of execution */
void main()
{
        FILE *pFile = NULL;
        int fileIsClose=0;
        double *integral = NULL;
        create1DDoubleArray(&integral, SIZE);
        pFile = openFile("data\\h0_05\\TopBoundaryP8Q8.dat", "w");
        judgeFileOpenCorrectly(pFile, "TopBoundaryP8Q8.dat");

        calculateIntegration(integral);

        outputOne1DDoubleArray(pFile, integral, SIZE);
        fileIsClose = closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose, "TopBoundaryP8Q8.dat");

        pFile = openFile("data\\h0_05\\Flux.dat", "w");
        judgeFileOpenCorrectly(pFile, "Flux.dat");
        writeOneDoubleValue(pFile, integral[SIZE-1]);
        free1DDoubleArray(integral);
}
```

## B.8. PROGRAM FOR COMPUTING BOUNDARY CONDITION ON $h = h_{\infty}$ IN SINK FLOW (calRight.c)

```
#include <math.h>
#include "BasicOperation.h"
#define PI 3.1415926
#define QL 8.0
#define SIZE 161

/* Usage: calculate the inner integrand
   Input: two double variables on which the integrand depend
   Output: the value of the integrand
*/
double f(double p, double flux){
        double valFunction, alfa = PI / 2., temp1, temp2, rSquare, viNq;
        temp1 = exp(p) - 1;
        temp1 = pow(temp1,2);
        temp2 = exp(QL)-1;
        temp2 = pow(temp2,2);
        rSquare = temp1 + temp2;
        viNq = 2. * flux * pow((exp(QL) - 1), 3) / alfa / pow(rSquare, 2);
        valFunction = - exp(p) * viNq;
        return valFunction;
}

/* Usage: solve the stream function by Runge-Kutta method
   Input: x coordinate, step size, flux, and the value of pasi
  Output: updated pasi
*/
double rungeKutta( double x, double y, double h, double flux){
        double k1, k2, k3, k4, valFunction;
        valFunction = f(x,  flux);
        k1 = valFunction * h;
        valFunction = f(x + h / 2., flux);
        k2 = valFunction * h;
        valFunction = f(x + h / 2., flux);
        k3 = valFunction * h;
        valFunction = f(x + h, flux);
        k4 = valFunction * h;
        y += 1. / 6. * (k1 + 2. * (k2 + k3) + k4);
        return y;
}
```

```
/* Usage: calculate the x component of velocity
    Input: x coordinate, the flux, and the alfa
   Output: the updated velocity u array
*/
void calculateRightU( double p[], double u[], double  flux, double alfa){
        int i = 0;
        double temp1 = 0., temp2 = 0.;
        for(i = 0; i < SIZE; i++){
                temp1 = pow((exp(QL) - 1.), 2);
                temp2 = pow((exp(p[i]) - 1.), 2);
        u[i] = 2. * flux / alfa * pow((exp(QL) - 1.),2)*(exp(p[i]) - 1.) / pow((temp1 +
        temp2), 2);
        }
}

/* Usage: calculate the stream function at the right boundary
   Input: x coordinate, step size, and the flux
   Output: the updated pasi array
*/
void calculateStreamFunction( double pP[], double pPasi[], double stepSize,
double flux){
        int i = 0;
        pPasi[0] = flux;
        for(i = 1; i < SIZE; i++){
                pPasi[i] = rungeKutta(pP[i - 1], pPasi[i - 1], stepSize, flux);
        }
}

/* Main Function: starting point of execution */
void main(){
        double *pP = NULL, *pPasi = NULL, *pU = NULL, stepSize = 0.05, flux =
        0., alfa = PI/2.;
        int i = 0, isSuccessful = 0,fileIsClose = 0;
        FILE*pFile = NULL;

        create1DDoubleArray(&pP, SIZE);
        create1DDoubleArray(&pPasi, SIZE);
        create1DDoubleArray(&pU, SIZE);

        pFile = openFile("data\\h0_05\\Flux.dat", "r");
        judgeFileOpenCorrectly(pFile, "Flux.dat");
        readOneDoubleValue(pFile, &flux);
        fileIsClose = closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose, "Flux.dat");
```

```
        setA1DDoubleArrayIniValue(pP, SIZE, stepSize);

        calculateRightU( pP, pU, flux, alfa);
        calculateStreamFunction( pP, pPasi, stepSize, flux);

        pFile = openFile("data\\h0_05\\RightBoundaryP8Q8.dat", "w");
        judgeFileOpenCorrectly(pFile, "RightBoundaryP8Q8.dat");
        outputTwo1DDoubleArrays(pFile, pP, pPasi, SIZE);
        fileIsClose = closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose, "RightBoundaryP8Q8.dat");
        pFile = openFile("data\\h0_05\\RightUP8Q8.dat", "w");
        judgeFileOpenCorrectly(pFile, "RightUP8Q8.dat");
        outputTwo1DDoubleArrays(pFile, pP, pU, SIZE);
        fileIsClose = closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose, "RightUP8Q8.dat");

        free1DDoubleArray ( pP );
        free1DDoubleArray ( pPasi );
        free1DDoubleArray ( pU );
}
```

## B.9. PROGRAM FOR COMPUTING BOUNDARY CONDITION ON $x = x_{\infty}$ IN SINK FLOW (calBottom.c)

```
#include <math.h>
#include "BasicOperation.h"
#define PL 8.
#define PI 3.1415926
#define SIZE 161

/* Usage: calculate the inner function
   Input: y coordinate and the flux on which the inner function depends
   Output: the value of the inner function
*/
double f(double q, double flux){
        double valFunction = 0., alfa = PI / 2., temp1, temp2, rSquare, uNpj;
        temp1 = exp(PL) - 1;
        temp1 = pow(temp1, 2);
        temp2 = exp(q) - 1;
        temp2 = pow(temp2, 2);
        rSquare = temp1 + temp2;
        uNpj = 2. * flux * (exp(PL) - 1) * pow((exp(q) - 1), 2) / alfa / pow(rSquare,
        2);
        valFunction = exp(q) * uNpj;
        return valFunction;
}

/* Usage: calculate the differential equation using Runge-Kutta method
  Input: x coordinate, step size, flux, and the value of pasi
  Output: the updated value of pasi
*/
double rungeKutta( double x, double y, double h, double flux){
        double k1, k2, k3, k4, valFunction;
        valFunction = f(x , flux);
        k1 = valFunction * h;
        valFunction = f(x + h / 2., flux);
        k2 = valFunction * h;
        valFunction = f(x + h / 2., flux);
        k3 = valFunction * h;
        valFunction = f(x + h, flux);
        k4 = valFunction * h;
        y += 1. / 6. * (k1 + 2. * (k2 + k3) + k4);
        return y;
}
```

```
/* Usage: calculate the y component of velocity
    Input: y coordinate, the flux, and the alfa
   Output: the updated velocity v array
*/
void calculateBottomV(double pQ[], double pV[], double flux, double alfa){
        int j = 0;
        double temp1 = 0., temp2 = 0.;
        for(j = 0; j < SIZE; j++){
                temp1 = pow((exp(PL) - 1.), 2);
                temp2 = pow((exp(pQ[j]) - 1.), 2);
        pV[j] = 2. * flux / alfa * pow((exp(pQ[j]) - 1.), 3) / pow((temp1 + temp2), 2);
        }
}
/* Usage: calculate the stream function
   Input: the y coordinate, the step size, and the flux
   Output: the updated pasi array
*/
void calculateStreamFunction(double q[], double pasi[], double stepSize, double
flux){
        int j = 0;
        pasi[0] = 0.;
        for(j = 1; j < SIZE; j++)
                pasi[j] = rungeKutta(q[j-1], pasi[j-1], stepSize, flux);
}

/* Main Function: starting point of execution */
void main(){
        double *pQ = NULL, *pPasi = NULL, stepSize = 0.05, *pV = NULL, flux,
        alfa = PI / 2.;
        int j = 0, fileIsClose = 0;
        FILE* pFile = NULL;

        create1DDoubleArray(&pQ, SIZE);
        create1DDoubleArray(&pPasi, SIZE);
        create1DDoubleArray(&pV, SIZE);

        pFile = openFile("data\\h0_05\\Flux.dat", "r");
        judgeFileOpenCorrectly(pFile, "Flux.dat");
        readOneDoubleValue(pFile, &flux);
        fileIsClose = closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose, "Flux.dat");
        setA1DDoubleArrayIniValue(pQ, SIZE, stepSize);

        calculateBottomV(pQ, pV, flux, alfa);
        calculateStreamFunction(pQ, pPasi, stepSize, flux);
```

```
        pFile = openFile("data\\h0_05\\BottomBoundaryP8Q8.dat", "w");
        judgeFileOpenCorrectly(pFile, "BottomBoundaryP8Q8.dat");
        outputTwo1DDoubleArrays(pFile, pQ, pPasi, SIZE);
        fileIsClose = closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose, "BottomBoundaryP8Q8.dat");
        pFile = openFile("data\\h0_05\\BottomVP8Q8.dat", "w");
        judgeFileOpenCorrectly(pFile, "BottomVP8Q8.dat");
        outputTwo1DDoubleArrays(pFile, pQ, pV, SIZE);
        fileIsClose = closeFile(pFile);
        judgeFileCloseCorrectly(fileIsClose, "BottomVP8Q8.dat");

        free1DDoubleArray ( pQ );
        free1DDoubleArray ( pPasi );
        free1DDoubleArray ( pV );
}
```

## B.10. PROGRAM FOR COMPUTING STREAM FUNCTIONS OF SINK FLOW (streamCal.c)

```
#include "basicOperation.h"
#include <math.h>
#define SIZE1 161
#define SIZE2 161
#define PL 8.
#define QL 8.
#define NP 160
#define NQ 160

/* Usage: read stream function from a specified file
   Input: pFile: the pointer pointing to the file
          p & q: 1D double arrays used to store x & ycoordinates
          pasi: 2D double array used to store the stream functions
          nx & ny: number of rows and cols for pasi
   Output: updated arrays
*/
void readStreamFunctionFromFile(FILE* pFile, double p[], double q[], double
*pasi[SIZE2], int nx, int ny){
        int i = 0, j = 0;
        for(i=0;  i <= nx; ++i)
                for(j = 0; j <= ny; ++j)
                        fscanf(pFile, "%lf %lf %lf", &p[i], &q[j], &pasi[i][j]);
}

/* Usage: interpolate stream functions got when step size h = 0.1
   Input: pasi: used to store stream functions interpolated
          TempPasi: used to store stream function to interpolate
   Output: updated stream functions stored in pasi
*/
void interpolateStreamFunction( double* pasi[SIZE2], double *tempPasi[SIZE2 /
2]){
        int i = 0, j = 0;
        for(i = 0; i < SIZE1; i += 2)
                for(j = 0; j < SIZE2; j += 2)
                        pasi[i][j] = tempPasi[I / 2][j / 2];
        for(i = 1; i < SIZE1 - 1; i += 2)
                for(j = 0; j < SIZE2; j += 2)
                    pasi[i][j] = (tempPasi[(i-1)/2][j/2] + tempPasi[(i+1)/2][j/2]) / 2.;
        for(i = 0; i < SIZE1; ++i)
                for(j = 1; j < SIZE2 - 1; j += 2)
                        pasi[i][j] = (pasi[i][j-1] + pasi[i][j+1]) / 2.;
}
```

```
/* Usage: read top boundary conditions
   Input: pFile: a pointer pointing to the file to be read
          q: a 1D double array to store y coordinate
          T: a 1D double array to store the temperature at the top boundary
   Output: the updated T array
*/
void readTopBoundary(FILE *pFile, double* pasi[SIZE2]){
        int j = 0;
        for( j = 0 ; j < SIZE2;  j++ )
                readOneDoubleValue(pFile, &pasi[0][j]);
}

/* Usage: read right boundary conditions
   Input: pFile: a pointer pointing to the file to be read
          p: 1D double array to store x coordinate
          pasi: 2D double array to store stream functions
   Output: updated pasi array
*/
void readRightBoundary(FILE* pFile, double p[], double *pasi[SIZE2]){
        int i = 0, j = 0;
        for( i = 0 ; i <= NP; i ++ )
                readTwoDoubleValues(pFile, &p[i], &pasi[i][NQ] );
}

/* Usage: read x component of velocity on the right boundary
   Input: pFile: a pointer pointing to the file to be read
          p: 1D double array to store x coordinate
          u: 1D double array to store x component velocity on the right-bound
   Output: updated u array
*/
void readRightU(FILE* pFile, double p[], double u[]){
        int i = 0;
        for( i = 0 ; i <= NP; i ++ )
                readTwoDoubleValues(pFile, &p[i], &u[i]);
}

/* Usage: read bottom boundary conditions
   Input: pFile: a pointer pointing to the file to be read
          q: 1D double array to store y coordinate
          pasi: 2D double array to store stream functions
   Output: updated pasi array
*/
void readBottomBoundary(FILE* pFile, double q[], double *pasi[SIZE2]){
        int i = 0, j = 0;
```

```
        for( i = 0 ; i <= NP; i ++ )
                readTwoDoubleValues(pFile, &q[j], &pasi[NP][j] );
}

/* Usage: read y component of velocity on the bottom boundary
   Input: pFile: a pointer pointing to the file to be read
          q: 1D double array to store x coordinate
          v: 1D double array to store x component velocity on the right-bound
   Output: updated u array
*/
void readBottomV(FILE* pFile, double q[], double v[]){
        int j = 0;
        for( j = 0 ; j <= NQ; j ++ )
                readTwoDoubleValues(pFile, &q[j], &v[j]);
}

/* Usage: set the initial value of the stream function on inner points
   Input: pasi: 2D double array for store stream functions
          nx & ny: x & y dimensions of array pasi
   Output: the updated array pasi
*/
void setIniStreamOnInnerPoints(double *pPasi[SIZE2], int nx, int ny){
        int i = 0, j = 0;
        for( i = 1 ; i < nx ; i++)
                for( j = 1 ; j < ny ; j++ )
                        pPasi[i][j] = pPasi[nx][j] / (nx) * i;
}

/* Usage: set pasi0 as a reference of pasi
   Input: two 2D double arrays: pasi and pasi0
          nx & ny are x & y dimensions of the 2D arrays
   Output: updated pasi0
*/
void setPasi0(double *pPasi[SIZE2], double *pPasi0[SIZE2], int nx, int ny){
        int i = 0, j = 0;
        for(i = 0; i <= nx; i++)
                for(j = 0; j <= ny; j++)
                        pPasi0[i][j] = pPasi[i][j];
}

/* Usage: calculate the coefficient on (i, j)
   Input: step size and x & y coordinate
   Output: coefficient on (i, j)
*/
```

```
double calculateCoefIAndJ(const double stepSize,const double p,const double
q){
        double temp1,temp2,temp3,temp4,temp5,coefOnIJ;
        temp1 = exp(-4. * p);
        temp2 = exp(-4. * q );
        temp3 = ( temp1 + temp2 ) / pow( stepSize , 4 ) * ( 6. - 22. * pow(
        stepSize , 2 ) );
        temp1 = exp( -2. * p - 2. * q );
        temp4 = temp1 * 8. / pow( stepSize , 4 );
        temp5 = 2. * exp(-2.*p-2.*q) / pow( stepSize, 2 );
        coefOnIJ = temp3 + temp4 + temp5 ;
        return coefOnIJ;
}

/* Usage: calculate the coefficient on (i+1, j)
   Input: step size and x & y coordinates
   Output: coefficient on (i+1, j)
*/
double calculateCoefIPlus1J(const double stepSize,const double p,const
double q){
        double temp1,temp2,temp3,temp4,temp5,coefOnIPlus1J;
        temp1=exp(-4.*p);
        temp2=(-3.*pow(stepSize,3)+11.*pow(stepSize,2)+6.*stepSize-
        4.)/pow(stepSize,4);
        temp3=temp1*temp2;
        temp1=exp(-2.*p-2.*q);
        temp2=2.*(stepSize-2.)/pow(stepSize,4);
        temp4=temp1*temp2;
        temp5 = 2. * exp(-2.*p-2.*q) / pow( stepSize, 2 );
        coefOnIPlus1J=temp3+temp4-temp5;
        return coefOnIPlus1J;
}

/* Usage: calculate the coefficient on (i-1, j)
   Input: step size and x & y coordinates
   Output: coefficient on point (i-1, j)
*/
double calculateCoefIMinus1J(const double stepSize,const double p,const
double q){
        double temp1,temp2,temp3,temp4,coefOnIMinus1J;
        temp1=exp(-4.*p);
        temp2=(3.*pow(stepSize,3)+11.*pow(stepSize,2)-6.*stepSize-
        4.)/pow(stepSize,4);
        temp3=temp1*temp2;
        temp1=exp(-2.*p-2.*q);
```

```
        temp2=(2.+stepSize)*2./pow(stepSize,4);
        temp4=temp1*temp2;
        coefOnIMinus1J=temp3-temp4;
        return coefOnIMinus1J;
}

/* Usage: calculate the coefficient on (i, j+1)
   Input: step size and x & y coordinates
   Output: coefficient on (i, j +1)
*/
double calculateCoefIJPlus1(const double stepSize,const double p,const
double q){
        double temp1,temp2,temp3,temp4,temp5,coefOnIJPlus1;
        temp1=exp(-4.*q);
        temp2=(-3.*pow(stepSize,3)+11.*pow(stepSize,2)+6.*stepSize-
        4.)/pow(stepSize,4);
        temp3=temp1*temp2;
        temp1=exp(-2.*p-2.*q);
        temp2=2.*(stepSize-2.)/pow(stepSize,4);
        temp4=temp1*temp2;
        temp5 = 2. * exp(-2.*p-2.*q) / pow( stepSize, 2 );
        coefOnIJPlus1=temp3+temp4-temp5;
        return coefOnIJPlus1;
}


/* Usage: calculate the coefficient on (i, j-1)
   Input: step size and x & y coordinates
   Output: coefficient on (i, j -1)
*/
double calculateCoefIJMinus1(const double stepSize,const double p,const
double q){
        double temp1,temp2,temp3,temp4,coefOnIJMinus1;
        temp1=exp(-4.*q);
        temp2=(3.*pow(stepSize,3)+11.*pow(stepSize,2)-6.*stepSize-
        4.)/pow(stepSize,4);
        temp3=temp1*temp2;
        temp1=exp(-2.*p-2.*q);
        temp2=2.*(2.+stepSize)/pow(stepSize,4);
        temp4=temp1*temp2;
        coefOnIJMinus1=temp3-temp4;
        return coefOnIJMinus1;
}

/* Usage: calculate the coefficient on (i+2, j)
```

```
   Input: step size and x coordinate
   Output: coefficient on (i + 2, j)
*/
double calculateCoefIPlus2J(const double stepSize,const double p){
        double temp1,temp2,coefOnIPlus2J;
        temp1=exp(-4.*p);
        temp2=(1.-3.*stepSize)/pow(stepSize,4);
        coefOnIPlus2J=temp1*temp2;
        return coefOnIPlus2J;
}

/* Usage: calculate the coefficient on (i-2, j)
   Input: step size and x coordinate
   Output: coefficient on (i-2, j)
*/
double calculateCoefIMinus2J(const double stepSize,const double p){
        double temp1,temp2,coefOnIMinus2J;
        temp1=exp(-4.*p);
        temp2=(1.+3.*stepSize)/pow(stepSize,4);
        coefOnIMinus2J=temp1*temp2;
        return coefOnIMinus2J;
}

/* Usage: calculate the coefficient on (i, j+2)
   Input: step size and the y coordinate
   Output: the coefficient on (i, j + 2)
*/
double calculateCoefIJPlus2(const double stepSize,const double q){
        double temp1,temp2,coefOnIJPlus2;
        temp1=exp(-4.*q);
        temp2=(1.-3.*stepSize)/pow(stepSize,4);
        coefOnIJPlus2=temp1*temp2;
        return coefOnIJPlus2;
}

/* Usage: calculate the coefficient on (i, j-2)
   Input: step size and the y coordinate
   Output: the coefficient on (i, j - 2)
*/
double calculateCoefIJMinus2(const double stepSize,const double q){
        double temp1,temp2,coefOnIJMinus2;
        temp1=exp(-4.*q);
        temp2=(1.+3.*stepSize)/pow(stepSize,4);
        coefOnIJMinus2=temp1*temp2;
        return coefOnIJMinus2;
```

```
}

/* Usage: calculate the coefficient on (i+1, j+1)
   Input: step size, x & y coordinates
   Output: the coefficient on (i+1, j+1)
*/
double calculateCoefIPlus1JPlus1(const double stepSize,const double p, const
double q){
        double temp1,temp2,temp3,coefOnIPlus1JPlus1;
        temp1=exp(-2.*p-2.*q);
        temp2=2.*(1.-stepSize)/pow(stepSize,4);
        temp3 = 2. * exp(-2.*p-2.*q) / pow( stepSize, 2 );
        coefOnIPlus1JPlus1=temp1*temp2+temp3;
        return coefOnIPlus1JPlus1;
}

/* Usage: calculate the coefficient on (i+1, j  -1)
   Input: step size and x & y coordinates
   Output: the coefficient on (i + 1, j - 1 )
*/
double calculateCoefIPlus1JMinus1(const double stepSize,const double
p,const double q){
        double temp1,temp2,coefOnIPlus1JMinus1;
        temp1=exp(-2.*p-2.*q);
        temp2=2./pow(stepSize,4);
        coefOnIPlus1JMinus1=temp1*temp2;
        return coefOnIPlus1JMinus1;
}

/* Usage: calculate the coefficient on (i-1, j+1)
   Input: step size and x & y coordinates
   Output: the coefficient on (i - 1, j + 1)
*/
double calculateCoefIMinus1JPlus1(const double stepSize,const double
p,const double q){
        double temp1,temp2,coefOnIMinus1JPlus1;
        temp1=exp(-2.*p-2.*q);
        temp2=2./pow(stepSize,4);
        coefOnIMinus1JPlus1=temp1*temp2;
        return coefOnIMinus1JPlus1;
}

/* Usage: calculate the coefficient on the (i-1,j-1)
   Input: step size, x & y coordinates
   Output: coefficient on (i - 1, j - 1)
```

```
*/
double calculateCoefIMinus1JMinus1(const double stepSize,const double
p,const double q){
        double temp1,temp2,coefOnIMinus1JMinus1;
        temp1=exp(-2.*p-2.*q);
        temp2=2.*(1.+stepSize)/pow(stepSize,4);
        coefOnIMinus1JMinus1=temp1*temp2;
        return coefOnIMinus1JMinus1;
}


/* Usage: calculate coefficients in the descretized equation
   Input: addresses of all the coefficients need to be calculated
          Step size and x and y coordinates on the point
*/
void calculateCoef(double *coefOnIJ, double *coefOnIPlus1J, double
*coefOnIMinus1J, double *coefOnIJPlus1, double* coefOnIJMinus1, double*
coefOnIPlus2J, double* coefOnIMinus2J, double* coefOnIJPlus2, double*
coefOnIJMinus2, double* coefOnIPlus1JPlus1, double *coefOnIPlus1JMinus1,
double* coefOnIMinus1JPlus1, double *coefOnIMinus1JMinus1, double
stepSize, double p, double q){
        *coefOnIJ = calculateCoefIAndJ(stepSize, p, q);
        *coefOnIPlus1J = calculateCoefIPlus1J(stepSize, p, q);
        *coefOnIMinus1J = calculateCoefIMinus1J(stepSize, p, q);
        *coefOnIJPlus1 = calculateCoefIJPlus1(stepSize, p, q);
        *coefOnIJMinus1 = calculateCoefIJMinus1(stepSize, p, q);
        *coefOnIPlus2J = calculateCoefIPlus2J(stepSize, p);
        *coefOnIMinus2J = calculateCoefIMinus2J(stepSize, p);
        *coefOnIJPlus2 = calculateCoefIJPlus2(stepSize, q);
        *coefOnIJMinus2 = calculateCoefIJMinus2(stepSize, q);
        *coefOnIPlus1JPlus1 = calculateCoefIPlus1JPlus1(stepSize, p, q);
        *coefOnIPlus1JMinus1 = calculateCoefIPlus1JMinus1(stepSize, p, q);
        *coefOnIMinus1JPlus1 = calculateCoefIMinus1JPlus1(stepSize, p, q);
        *coefOnIMinus1JMinus1 =calculateCoefIMinus1JMinus1(stepSize, p,q);
}

/* Usage: calculate coef * pasi (usual expressions)
   Input: coefficient on that point and the pasi array, i & j index
   Output: part1 ~ part 12
*/
void calculateParts(double*part1, double* part2, double* part3, double* part4,
double *part5, double *part6, double *part7, double *part8, double *part9,
double *part10, double *part11, double *part12, double coefOnIPlus1J, double
coefOnIMinus1J, double coefOnIJPlus1, double coefOnIJMinus1, double
coefOnIPlus2J, double coefOnIMinus2J, double coefOnIJPlus2, double
```

```
coefOnIJMinus2, double coefOnIPlus1JPlus1, double coefOnIPlus1JMinus1,
double coefOnIMinus1JPlus1, double coefOnIMinus1JMinus1, double
*pPasi[SIZE2], int i, int j){
        *part1 = coefOnIPlus1J * pPasi[i+1][j];
        *part2 = coefOnIMinus1J * pPasi[i-1][j];
        *part3 = coefOnIJPlus1 * pPasi[i][j+1];
        *part4 = coefOnIJMinus1 * pPasi[i][j-1];
        if( i != NP - 1 )
        *part5 = coefOnIPlus2J * pPasi[i+2][j];
        if( i != 1)
        *part6 = coefOnIMinus2J * pPasi[i-2][j];
        if( j != NQ - 1)
        *part7 = coefOnIJPlus2 * pPasi[i][j+2];
        if( j != 1)
        *part8 = coefOnIJMinus2 * pPasi[i][j-2];
        *part9 = coefOnIPlus1JPlus1 * pPasi[i+1][j+1];
        *part10 = coefOnIPlus1JMinus1* pPasi[i+1][j-1];
        *part11 = coefOnIMinus1JPlus1* pPasi[i-1][j+1];
        *part12 = coefOnIMinus1JMinus1 * pPasi[i-1][j-1];
}

/* Usage: calculate the sum of those 12 parts
   Input: twelve double variables
   Output: the sum
*/
double sum12Parts(double part1, double part2, double part3, double part4,
double part5, double part6, double part7, double part8, double part9, double
part10, double part11, double part12){
        double sum=0.;
        sum=part1+part2+part3+part4+part5+part6+part7+part8+part9+part10+p
        art11+part12;
        return sum;
}

/* Usage: calculate stream function on (1,1)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[1][1]
*/
void calculateStreamFunctionOn11(double *pPasi[SIZE2], double stepSize,
double p, double q, int i, int j){
        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
```

```
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp1, temp2, temp3,
        part1, part2, part3, part4, part5, part6, part7, part8, part9, part10, part11,
        part12;

//calculate coefficients
        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1,  &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);

        temp1=coefOnIMinus2J*(stepSize-2.)/(stepSize+2.)+coefOnIJMinus2;
        temp2=1.+temp1/coefOnIJ;
        temp3=1./temp2/(-coefOnIJ);

//calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);

        //part6(special)
        part6=coefOnIMinus2J*2./(1.+stepSize/2.)*pPasi[0][j];
        //part8(special)
        part8=0.;

        pPasi[i][j]=temp3*sum12Parts(part1,part2,part3,part4,part5,part6,part7,p
        art8,part9,part10,part11,part12);
}

/* Usage: calculate stream function on (1, NQ - 1)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[1][NQ - 1]
*/
void calculateStreamFunctionOn1NqMinus1(double *pPasi[SIZE2], double
stepSize, double ql, double u, double p, double q, int i, int j){
        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
```

```
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp1, temp2, temp3,
        part1, part2, part3, part4, part5, part6, part7, part8, part9, part10, part11,
        part12;

        //calculate coefficients
        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1,  &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);

        temp1=coefOnIMinus2J*(stepSize-2.)/(stepSize+2.)+coefOnIJPlus2;
        temp2=1.+temp1/coefOnIJ;
        temp3=1./temp2/(-coefOnIJ);

        //calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);
        //part6(special)
        part6=coefOnIMinus2J*(2./(1.+stepSize/2.))*pPasi[i-1][j];
        //part7(special)
        part7=coefOnIJPlus2*(2.*stepSize*exp(ql)*u);

        pPasi[i][j]=temp3*sum12Parts(part1,part2,part3,part4,part5,part6,part7,p
        art8,part9,part10, part11,part12);
}

/* Usage: calculate stream function on (NP - 1, 1)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[NP-1][1]
*/
void calculateStreamFunctionOnNpMinus11(double *pPasi[SIZE2], double
stepSize, double pl, double v, double p, double q, int i, int j){
        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp1, temp2, temp3,
```

```
        part1, part2, part3, part4, part5, part6, part7, part8, part9, part10, part11,
        part12;

        //calculate coefficients
        calculateCoef(&coefOnIJ,&coefOnIPlus1J,&coefOnIMinus1J,&coefOnIJP
        lus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1, &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);

        temp1=(coefOnIPlus2J+coefOnIJMinus2)/coefOnIJ;
        temp2=1.+temp1;
        temp3=1./temp2/(-coefOnIJ);

        //calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);

        //part5(special)
        part5=coefOnIPlus2J*(-2.*stepSize*exp(pl)*v);
        //part8(special)
        part8=0.;

        pPasi[i][j]=temp3*sum12Parts(part1,part2,part3,part4,part5,part6,part7,p
        art8,part9,part10, part11,part12);
}

/* Usage: calculate stream function on (NP - 1, NQ - 1)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[NP-1][NQ - 1]
*/
void calculateStreamFunctionOnNpMinus1NqMinus1(double *pPasi[SIZE2],
double stepSize, double u, double v, double pl, double ql, double p, double
q,const int i,const int j){
        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
```

```
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp1, temp2, temp3,
        part1, part2, part3, part4, part5, part6, part7, part8, part9, part10, part11,
        part12;

        //calculate those coefficients
        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1, &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);

        temp1=(coefOnIPlus2J+coefOnIJPlus2)/coefOnIJ;
        temp2=1.+temp1;
        temp3=1./temp2/(-coefOnIJ);

        //calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);
        //part5(special)
        part5=coefOnIPlus2J*(-2.*stepSize*exp(pl)*v);
        //part7(special)
        part7=coefOnIJPlus2*(2.*stepSize*exp(ql)*u);

        pPasi[i][j]=temp3*sum12Parts(part1,part2,part3,part4,part5,part6,part7,p
        art8,part9,part10, part11,part12);
}

/* Usage: calculate stream function on (1, j)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[1][j]
*/
void calculateStreamFunctionOn1J(double *pPasi[SIZE2], double stepSize,
double p, double q, int i, int j){
        double coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, coefOnIJ, temp1, temp2,
```

```
        temp3, temp4, part1, part2, part3, part4, part5, part6, part7, part8, part9,
        part10, part11, part12;
        //calculate coefficients
        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1, &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);

        temp1=(stepSize-2.)/(stepSize+2.);
        temp2=coefOnIMinus2J/coefOnIJ;
        temp3=temp1*temp2;
        temp4=1./(1.+temp3)/(-coefOnIJ);

        //calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);
        //part6(i-2,j)(special);
        part6=coefOnIMinus2J*pPasi[0][j]*2./(1.+stepSize/2.);

        pPasi[i][j]=temp4*sum12Parts(part1,part2,part3,part4,part5,part6,part7,p
        art8,part9,part10, part11,part12);
}

/* Usage: calculate stream function on (NP - 1, j)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[NP - 1][j]
*/
void calculateStreamFunctionOnNpMinus1J(double *pPasi[SIZE2], double
stepSize, double pl, double v, double p, double q, int i, int j){
        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp1, temp2, temp3,
        part1, part2, part3, part4, part5, part6, part7, part8, part9, part10, part11,
        part12;

        //calculate coefficients
```

```
        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1, &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);

        temp1=coefOnIPlus2J/coefOnIJ;
        temp2=1+temp1;
        temp3=1./temp2/(-coefOnIJ);

        // calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8  , &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);
        //part5(special)
        part5=coefOnIPlus2J*(-2.*stepSize*exp(pI)*v);

        pPasi[i][j]=temp3*sum12Parts(part1,part2,part3,part4,part5,part6,part7,p
        art8,part9,part10, part11,part12);
}

/* Usage: calculate stream function on (i, 1)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[i][1]
*/
void calculateStreamFunctionOnI1(double *pPasi[SIZE2], double stepSize,
double p, double q, int i, int j){
        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp1, temp2, temp3,
        part1, part2, part3, part4, part5, part6, part7, part8, part9, part10, part11,
        part12;
        //calculate coefficients
        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1, &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize,p,q);
```

```
        temp1=coefOnIJMinus2/coefOnIJ;
        temp2=1.+temp1;
        temp3=1./temp2/(-coefOnIJ);

        //calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);
        //part8 (special)
        part8=0.;

        pPasi[i][j]=temp3*sum12Parts(part1,part2,part3,part4,part5,part6,part7,p
        art8,part9,part10, part11,part12);
}

/* Usage: calculate stream function on (i, NQ - 1)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[i][NQ - 1]
*/
void calculateStreamFunctionOnINqMinus1(double *pPasi[SIZE2], double
stepSize, double ql, double u, double p, double q, int i, int j){
        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp1, temp2, temp3,
        part1, part2, part3, part4, part5, part6, part7, part8, part9, part10, part11,
        part12;

        //calculate those coefficients
        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1, &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize, p, q);

        temp1=coefOnIJPlus2/coefOnIJ;
        temp2=1.+temp1;
        temp3=1./temp2/(-coefOnIJ);
```

```
        //calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);
        //part7(special)
        part7=coefOnIJPlus2*2.*stepSize*exp(ql)*u;

        pPasi[i][j]=temp3*sum12Parts(part1,part2,part3,part4,part5,part6,part7,p
        art8,part9,part10, part11,part12);
}

/* Usage: calculate stream function on (i, j)
   Input: pasi: 2D double array used to store stream function
          stepSize: the step size
          p & q: x & y coordinate on that point
          i & j: x & y index on that point
   Output: pasi[i][j]
*/
void calculateStreamFunctionOnIJ(double *pPasi[SIZE2], double stepSize,
double p, double q, int i, int j){
        double coefOnIJ, coefOnIPlus1J, coefOnIMinus1J, coefOnIJPlus1,
        coefOnIJMinus1, coefOnIPlus2J, coefOnIMinus2J, coefOnIJPlus2,
        coefOnIJMinus2, coefOnIPlus1JPlus1, coefOnIPlus1JMinus1,
        coefOnIMinus1JPlus1, coefOnIMinus1JMinus1, temp3, part1, part2,
        part3, part4, part5, part6, part7, part8, part9, part10, part11, part12;

        //calculate coefficients
        calculateCoef(&coefOnIJ, &coefOnIPlus1J, &coefOnIMinus1J,
        &coefOnIJPlus1, &coefOnIJMinus1, &coefOnIPlus2J, &coefOnIMinus2J,
        &coefOnIJPlus2, &coefOnIJMinus2, &coefOnIPlus1JPlus1,
        &coefOnIPlus1JMinus1, &coefOnIMinus1JPlus1,
        &coefOnIMinus1JMinus1, stepSize,p,q);

        temp3=-1./coefOnIJ;
        //calculate those 12 parts
        calculateParts(&part1, &part2, &part3, &part4, &part5, &part6, &part7,
        &part8, &part9, &part10, &part11, &part12, coefOnIPlus1J,
        coefOnIMinus1J, coefOnIJPlus1, coefOnIJMinus1, coefOnIPlus2J,
        coefOnIMinus2J, coefOnIJPlus2, coefOnIJMinus2,coefOnIPlus1JPlus1,
        coefOnIPlus1JMinus1, coefOnIMinus1JPlus1, coefOnIMinus1JMinus1,
        pPasi, i, j);
```

```
        pPasi[i][j]=temp3*sum12Parts(part1,part2,part3,part4,part5,part6,part7,p
        art8,part9,part10, part11,part12);
}

/* Usage: calculate the maximum error
   Input: the array pasi, the reference array pasi0, number of rows & columns of
the 2D array, and the addresses of two integers storing the i-index and j-index
of the point with largest difference between two iteration*/
double calculateError(double *pPasi[SIZE2],double *pPasi0[SIZE2],int* pi,int
*pj){
        int i = 0, j = 0;
        double error = 0.;
        error = fabs(pPasi[1][1] - pPasi0[1][1]);
        for(i = 1; i < NP;i++)
                for(j = 1; j < NQ; j++)
        if(error < fabs(pPasi[i][j] - pPasi0[i][j])){
                error = fabs(pPasi[i][j] - pPasi0[i][j]);
                *pi = i;
                *pj = j;
        }
        return error;
}

/* Usage: iterate to get stream functions satisfying accuracy requirement
   Input: pasi: the 2D double array used to store the stream function
          pasi0: the reference of pasi
          stepSize: the step size
          p & q: 1D double arrays used to store x & y coordinates
          T: 1D double array used to store temperature at the interface
   Output: the updated pasi array
*/
void iterateForStream(double *pasi[SIZE2], double *pasi0[SIZE2], double p[],
double q[], double u[], double v[], double stepSize, double accuracyControl,
double relax){
        int i = 0, j = 0, erri = 1, errj = 1, numOfIteration = 0;
        double error = 0.;
        do{
                i = 1; j = 1;
                calculateStreamFunctionOn11(pasi, stepSize, p[i], q[j], i, j);

                i = 1; j = NQ -1;
                calculateStreamFunctionOn1NqMinus1(pasi,stepSize,QL,u[i],
                p[i],q[j],i,j);
```

```
            i = NP - 1; j = 1;
            calculateStreamFunctionOnNpMinus11(pasi,stepSize,PL,v[j], p[i],
            q[j],i,j);

            i = NP - 1; j = NQ - 1;
            calculateStreamFunctionOnNpMinus1NqMinus1(pasi,stepSize,
            u[i], v[j], PL, QL, p[i], q[j], i, j);

            for(i = 1, j = NQ - 2; j >= 2; j--)
                    calculateStreamFunctionOn1J(pasi, stepSize, p[i],q[j], i, j);
            for(i = NP - 1, j = NQ - 2; j >= 2; j--)
                    calculateStreamFunctionOnNpMinus1J(pasi,stepSize,PL,v[
                    j], p[i],q[j],i,j);
            for(j = 1, i = NP - 2; i >= 2; i--)
                    calculateStreamFunctionOnI1(pasi,stepSize,p[i],q[j],i,j);
            for(j = NQ -1, i = NP - 2; i >= 2; i--)
                    calculateStreamFunctionOnINqMinus1(pasi,stepSize,QL,u[i
                    ], p[i],q[j],i,j);
            for( j = NQ - 2 ; j >= 2; j--){
                    for( i = NP -2 ; i >= 2 ; i--)
                            calculateStreamFunctionOnIJ(pasi,stepSize,p[i],q[j],i,
                            j);
            }//for
            // calculate error
            error = calculateError(pasi, pasi0, &erri, &errj);
            if(fabs(error) < fabs(accuracyControl)) break;
            ++ numOfIteration;
            for(i = 1; i < NP; i++)
                    for(j = 1; j < NQ; j++)
                            pasi[i][j] = pasi0[i][j] + relax * (pasi[i][j] - pasi0[i][j]);
            setPasi0(pasi, pasi0, NP, NQ);
            if(numOfIteration%10 == 0){
            printf("i = %d j = %d\n", erri, errj);
            printf("%d  err = %15.14lf pasi = %lf\n", numOfIteration, error,
            pasi[erri][errj]);
            }
    }while(error>accuracyControl);
}


main(){
    double *p = NULL, *q = NULL,**pasi = NULL, **tempPasi =
    NULL,*tempP = NULL, *tempQ = NULL, stepSize,*u = NULL, *v = NULL,
    **pasi0 = NULL ,accuracyControl,relax;
    FILE* pFile = NULL;
```

```
int fileIsClose = 0;
accuracyControl=1.e-6;
relax=1.1;

// determine the stepSize,np,nq
stepSize = PL / (SIZE1 - 1.);

// Create 1D double arrays
create1DDoubleArray( &p, SIZE1 );
create1DDoubleArray( &q, SIZE2 );
create1DDoubleArray( &u, SIZE1 );
create1DDoubleArray( &v, SIZE2 );
create1DDoubleArray( &tempP, SIZE1 );
create1DDoubleArray( &tempQ, SIZE2 );
// Create 2D double arrays
create2DDoubleArray( &pasi, SIZE1, SIZE2 );
create2DDoubleArray( &pasi0, SIZE1, SIZE2 );
create2DDoubleArray( &tempPasi, SIZE1, SIZE2 );
// Set initial values for 1D arrays
setA1DDoubleArrayIniValue(p, SIZE1, stepSize);
setA1DDoubleArrayIniValue(q, SIZE2, stepSize );
setA1DDoubleArrayIniValue(u, SIZE1, 0. );
setA1DDoubleArrayIniValue(v, SIZE2, 0. );
setA1DDoubleArrayIniValue(tempP, SIZE1, 0. );
setA1DDoubleArrayIniValue(tempQ, SIZE2, 0. );
// Initialize pasi, pasi0
setA2DDoubleArrayIniValue(pasi, SIZE1, SIZE2, 0. );
setA2DDoubleArrayIniValue(pasi0, SIZE1, SIZE2, 0. );
setA2DDoubleArrayIniValue(tempPasi, SIZE1, SIZE2, 0. );
/* Read stream function from file with previous iteration result
  This section restricted by '/*' & '*/ //is used after at least one iteration
//*/
/*pFile = openFile("data\\h0_05\\streamP8Q81.dat", "r");
judgeFileOpenCorrectly(pFile, "streamP8Q81.dat");
readStreamFunctionFromFile(pFile, p, q, pasi, NP, NQ);
closeFile(pFile);
fileIsClose = closeFile(pFile)
judgeFileCloseCorrectly(fileIsClose, "streamP8Q81.dat")
*/

/* Read initial stream function for stepSize h = 0.1
  Use this to interpolate to get initial stream function for stepSize h =.05
  This section will be used only for the first iteration
*/
pFile = openFile("data\\h0_05\\streamP8Q8_h0_1.dat", "r");
```

```
judgeFileOpenCorrectly(pFile, "streamP8Q8h0_1.dat");
readStreamFunctionFromFile(pFile, tempP, tempQ, tempPasi, NP/2,
NQ/2);
fileIsClose = closeFile(pFile);
judgeFileCloseCorrectly(fileIsClose, "streamP8Q8h0_05.dat");

/* Interpolate the stream function */
interpolateStreamFunction(pasi, tempPasi);
/* free tempP, tempQ, and tempPasi*/
free1DDoubleArray(tempP);
free1DDoubleArray(tempQ);
free2DDoubleArray(tempPasi, SIZE1);

/* open file for reading boundary conditions of the stream functions*/
// Top boundary
pFile = openFile("data\\h0_05\\TopBoundaryP8Q8.dat", "r");
judgeFileOpenCorrectly(pFile, "TopBoundaryP8Q8.dat");
readTopBoundary(pFile, pasi);
fileIsClose = closeFile(pFile);
judgeFileCloseCorrectly(fileIsClose, "TopBoundaryP8Q8.dat");
// Right boundary
pFile = openFile("data\\h0_05\\RightBoundaryP8Q8.dat", "r");
judgeFileOpenCorrectly(pFile, "RightBoundaryP8Q8.dat");
readRightBoundary(pFile, p, pasi);
fileIsClose = closeFile(pFile);
judgeFileCloseCorrectly(fileIsClose, "RightBoundaryP8Q8.dat");
pFile = openFile("data\\h0_05\\RightUP8Q8.dat", "r");
judgeFileOpenCorrectly(pFile, "RightUP8Q8.dat");
readRightU(pFile, p, u);
fileIsClose = closeFile(pFile);
judgeFileCloseCorrectly(fileIsClose, "RightUP8Q8.dat");

// Bottom boundary
pFile = openFile("data\\h0_05\\BottomBoundaryP8Q8.dat", "r");
judgeFileOpenCorrectly(pFile, "BottomBoundaryP8Q8.dat");
readBottomBoundary(pFile, q, pasi);
fileIsClose = closeFile(pFile);
judgeFileCloseCorrectly(fileIsClose, "BottomBoundaryP8Q8.dat");
pFile = openFile("data\\h0_05\\BottomVP8Q8.dat", "r");
judgeFileOpenCorrectly(pFile, "BottomVP8Q8.dat");
readBottomV(pFile, q, v);
fileIsClose = closeFile(pFile);
judgeFileCloseCorrectly(fileIsClose, "BottomVP8Q8.dat");

/* Set the initial stream functions for inner points
```

```
    This is only for the first iteration and the largest step size, say h = .1
*/
 setIniStreamOnInnerPoints(pasi, NP, NQ);


/*Set pasi0: pasi0 is used as the reference for pasi to check accuracy */
/*setPasi0(pasi, pasi0, NP, NQ);
*/

/* Start Iteraton */
iterateForStream(pasi, pasi0,  p, q, u, v, stepSize, accuracyControl,
relax);

// open file to write pasi
pFile = openFile("data\\h0_05\\StreamP8Q8.dat", "w");
judgeFileOpenCorrectly(pFile, "StreamP8Q8.dat");
outputThreeArrays(pFile, p, q, pasi, NP, NQ);
fileIsClose = closeFile(pFile);
judgeFileCloseCorrectly(fileIsClose, "StreamP8Q8.dat");

// Free memeory
free1DDoubleArray( p );
free1DDoubleArray( q );
free1DDoubleArray( u );
free1DDoubleArray( v );

free2DDoubleArray( pasi, SIZE1 );
free2DDoubleArray( pasi0, SIZE1 );
}
```

# VITA

The author, Jin Zhang, was born in February 1975, in Zhejiang, People's Republic of China. She graduated from No.14 Hangzhou High School in July 1993.

In September 1993, the author attended Shanghai Jiao Tong University (SJTU) with a major in thermal energy engineering and a minor in accounting. She obtained a bachelor of science degree in thermal energy engineering in July 1997. She continued her graduate study in SJTU and obtained her master of science degree in thermal energy engineering in March 2000. After that, she worked as a software engineer in Kingpo Electronic Company, Shanghai, for three months.

In August 2000, the author furthered her graduate study in the Department of Mechanical Engineering, Louisiana State University. And in December 2002, the degree of Master of Science in Mechanical Engineering will be conferred.